Typescript

About TypeScript

- Implementing a JavaScript-based application is more error-prone as it supports dynamic typing and supports non-strict mode with global variables.
- TypeScript makes such implementations easy, as it supports static typing and structured code with the help of modules and object-oriented concepts.
- This course introduces various features and programming constructs of TypeScript which enables you to develop a highly structured, less error-prone JavaScript code using TypeScript features.

Need For TypeScript

- JavaScript is the language used for client-side scripting to do client-side validations, DOM manipulation, Ajax calls, etc. using JavaScript. Also, JavaScript frameworks can be used for writing complex business logic that runs at the client-side.
- As the complexity of the JavaScript code increases, it gradually becomes difficult in coding and maintaining. This is because of the limitations of the JavaScript language. There is no option to change the language for the client-side scripting as the browser understands only JavaScript.
- The solution is to choose a language that is rich in features and the code can be converted to JavaScript. This process of converting the code written in one language into another language is called Transpilation.

TypeScript is one such language where its code can get transpiled to JavaScript.

Pitfalls of JavaScript

- **Dynamic Typing**: It decides the data type of the variable dynamically at run time.
- **Interpreted Language**: It is a language in which the code instructions are executed directly without prior compilation to machine-language instructions.
- **Minimal Object Oriented support**: Object-Oriented Programming (OOP) is a programming methodology based on the concept of objects. Object-Oriented concepts like classes, encapsulation, inheritance help in the readability and reusability of the code.
- **Minimal IDE support**: Integrated Development Environment (IDE) is a software application that provides all necessary options like code refactoring, intelliSense support, debugging support to software programmers for software development.

Dynamic Typing

Consider the below JavaScript function:

```
function calculateTotalPrice(quantity, unitPrice) {
        return quantity * unitPrice;
 }
```

We can invoke this function using the below code:

```
console.log(calculateTotalPrice(3, "500"));
```

Since JavaScript is dynamically typed, we can invoke the above function using the below code as well.

```
console.log(calculateTotalPrice('three', "500"));
```

Even though the above code will not throw any error, it will return **NaN** as output, since the expected number type value is not passed to the quantity argument of the calculateTotalPrice function.

To avoid the above runtime error we can use static typing in TypeScript, wherein we will add data type to the function argument while defining it. Consider the same JavaScript function written using TypeScript as below:

```
function calculateTotalPrice(quantity:number, unitPrice:number) {
    return quantity * unitPrice;
}
```

In the above code, we are adding a **number** as the data type to both the arguments of the function. Hence when we invoke the function using the below code:

```
console.log(calculateTotalPrice('three', "500"));
```

We will get a compilation error since we are invoking the function with the first parameter as a **string** type. Hence we can detect error early at compilation time itself.

Interpreted Language

JavaScript is an interpreted language. The advantages are:

- It takes less amount of time to analyze the source code
- Memory efficient as no intermediate object code will get generated.

The disadvantage is most of the errors can be identified only at run time.

- This can be overcome by using TypeScript which will be transpiled to JavaScript and most of the errors will be fixed during the transpilation time itself.
- Therefore, TypeScript saves the application development time for a JavaScript developer.

Minimal Object Oriented Support

The equivalent Typescript code is as below:

```
class Product{
    protected productId:number;
}
class Gadget extends Product{
    getProduct():void{
    }
}
```

You can observe from the above code that:

- The readability of the JavaScript code is minimal.
- Abstraction is done using closures or self-invoking functions wherein the number of lines of code is more compared to public, private, protected access modifiers.
- Classes are created using a constructor function, which leads to confusion compared to using the class keyword.
- Javascript supports OOP through prototypal inheritance which is complex for people with classical inheritance background to understand.

In addition to that TypeScript also supports interface and generic concepts which is not supported by JavaScript.

Minimal IDE Support

A few IDEs have code refactoring, IntelliSense support for JavaScript application development.

- IntelliSense support helps in writing the code quickly.
- Refactoring support helps in changing the variable or function names throughout the application quickly.

Most of the IDEs has good support for TypeScript, some are listed below:

- Visual Studio with versions 2015, 2013, and so on
- Sublime Text
- Atom
- Eclipse
- Emacs

- WebStorm
- Vim

Why TypeScript?

JavaScript application development has become easier with the help of the following tools:

- **npm** can be used to download packages
- **webpack** can be used to manage the complexity of applications.
- **Babel** can be used to fetch the latest features of the language.
- Tools like **rollup** and **uglifyjs** can be used to optimize application payloads.
- **prettier** and **eslint** can be used to uphold code with consistent style as well as quality.
- IDE like **Visual Studio Code** with **Node.js** environment can be used to run JavaScript code everywhere.

There is browser support challenge for the latest ES6 version of JavaScript. You can use ES6 transpilers like Babel to address this.

TypeScript can be another preferred option which is a superset of JavaScript and transpiles to the preferred version of JavaScript.

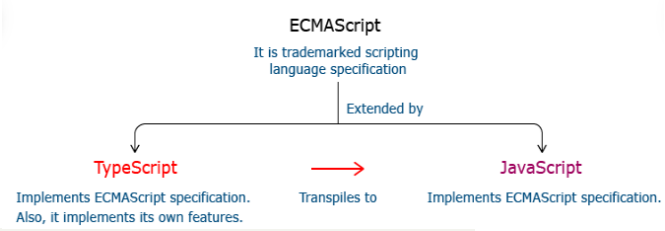
What is TypeScript?

TypeScript can be considered as a typed superset of JavaScript, that transpiles to JavaScript.

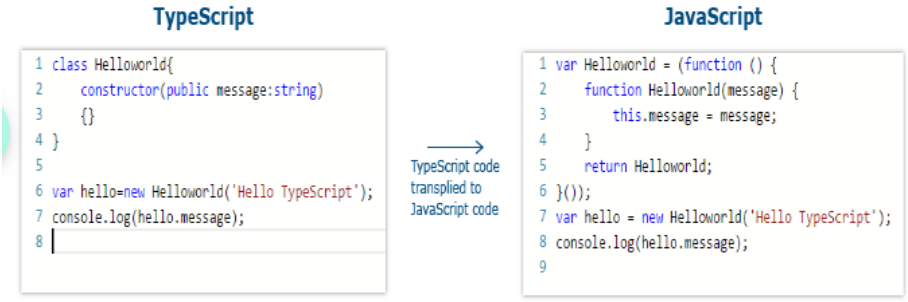
- Transplier converts the source code of one programming language to the source code of another programming language.
- TypeScript makes the development of JavaScript more of a traditional object-oriented experience.
- TypeScript is based on ECMAScript 6 and 7 proposals.
- Any valid JavaScript is TypeScript.

Relationship between TypeScript and JavaScript

- TypeScript implements EcmaScript specification. Apart from the EcmaScript specification, TypeScript has its own features as well.
- JavaScript also implements EcmaScript.
- TypeScript code must be transpiled to JavaScript code to use it in an application.



Relationship between TypeScript and JavaScript



From the above code, the TypeScript class Helloworld is converted to a self-invoking function in JavaScript when transpiled.

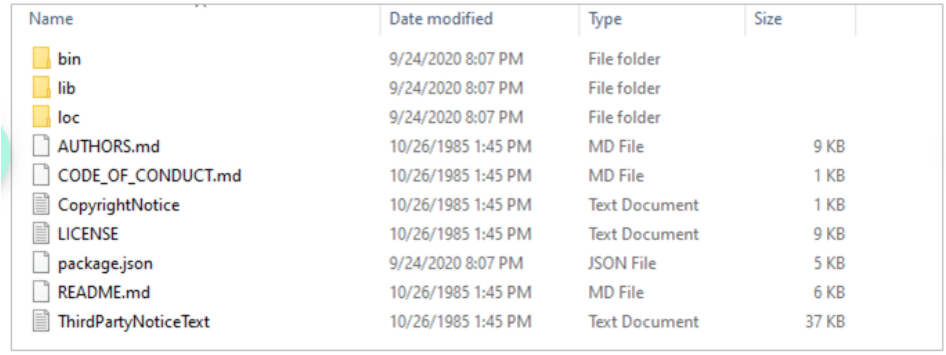
You can use TypeScript's online playground editor to see how TypeScript gets converted into JavaScript.

Features of TypeScript

Static Typing: It adds static typing to JavaScript, due to which the readability of the code improves and helps in finding more early compilation errors than run time errors.
Modules support: TypeScript provides an option to create modules to modularize the code for easy maintenance. Modules help in making the application scalable.
Object-Oriented Programming: TypeScript supports Object-Oriented Programming features such as class, encapsulation, interface, inheritance and so on which helps in creating highly structured and reusable code.
Open Source: TypeScript is open source. The source code of TypeScript can be downloaded from Github.
Cross-Platform: It works across the platform.
Tooling Support: TypeScript works extremely well with Sublime Text, Eclipse, and almost all major IDEs compared to JavaScript.

Installing TypeScript

- To install TypeScript, go to the official site of TypeScript (http://www.typescriptlang.org/) and follow the steps mentioned there to download TypeScript.
- As mentioned on the official site, you need to install Node.js.
- Install Node.js from the official site of Node.js (https://nodejs.org/en/) or Software Center.
- Open a Node.js command prompt and check whether node and npm are installed in your machine by using **"node -v "** and **"npm -v"** commands.
- npm is a command-line tool that comes along with Node.js installation with which you can download node modules. TypeScript is also such a node module that can be installed using npm.
- In the same Node.js command prompt, type the **"npm i –g typescript"** command to download the TypeScript module from the repository.
- On successful execution of above command, the TypeScript module will get downloaded under folder C:\Users\<<username>>\AppData\Roaming\npm\node_modules\typescript as shown below.



Name	Date modified	Type	Size
bin	9/24/2020 8:07 PM	File folder	
lib	9/24/2020 8:07 PM	File folder	
loc	9/24/2020 8:07 PM	File folder	
AUTHORS.md	10/26/1985 1:45 PM	MD File	9 KB
CODE_OF_CONDUCT.md	10/26/1985 1:45 PM	MD File	1 KB
CopyrightNotice	10/26/1985 1:45 PM	Text Document	1 KB
LICENSE	10/26/1985 1:45 PM	Text Document	9 KB
package.json	9/24/2020 8:07 PM	JSON File	5 KB
README.md	10/26/1985 1:45 PM	MD File	6 KB
ThirdPartyNoticeText	10/26/1985 1:45 PM	Text Document	37 KB

In the same command prompt check for TypeScript installation as below:

```
tsc -v
//or
tsc --version
```

- Output showing version number indicates the successful installation of the TypeScript module.
- Alternatively, you can also use TypeScript online playground editor in this case, you should be always connected to good Internet.

To configure TypeScript with different IDEs. Here are a few links for IDE configuration for TypeScript:

- **Visual Studio Code**: https://code.visualstudio.com/Docs/languages/typescript
- **Eclipse IDE**: https://github.com/palantir/eclipse-typescript
- **Visual Studio 2015**: https://angular.io/guide/visual-studio-2015

To configure TypeScript with different IDEs. Here are some of the popular IDEs for working with TypeScript:

- Visual Studio code
- Eclipse IDE
- Visual Studio 2015

First TypeScript Application

To start with the first application in TypeScript, create a file hello_typescript.ts under a folder. In the ts file give a console.log statement and save it.

hello_typescript.ts

```
1 console.log("Hello welcome to TypeScript");
2
```

From the Node.js command prompt, navigate to the directory in which the ts file resides and compile the ts file using the **tsc** command.

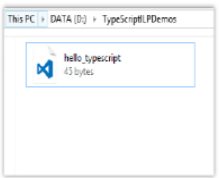
```
D:\>cd D:\TypeScriptILPDemos

D:\TypeScriptILPDemos>tsc hello_typescript.ts

D:\TypeScriptILPDemos>
```

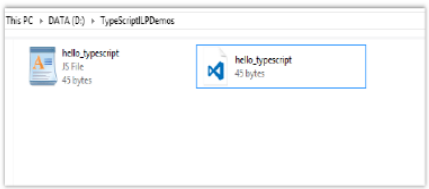
After compilation of the TypeScript file, the corresponding JavaScript file gets generated.

Before compilation:



Folder contains .ts file

After compilation:



Folder contains .ts and .js files

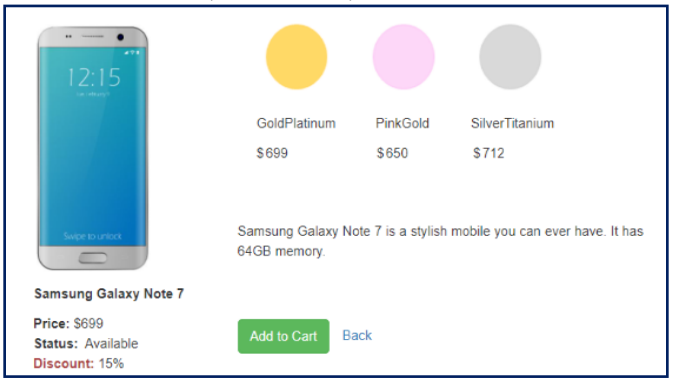
To run the generated JavaScript file, use the node command from the command line or include it in an HTML page using the script tag and render it in the browser.

```
D:\TypeScriptILPDemos>node hello_typescript.js
Hello welcome to TypeScript

D:\TypeScriptILPDemos>
```

TypeScript Basics

Consider the below page of the Mobile Cart application. The information such as mobile phone name, price, status on the availability of the mobile, discounts is displayed on the page. It also displays different price ranges as GoldPlatinum, PinkGold, SilverTitanium.



Each of the information getting displayed has a specific type. For example, the price can only be numeric and the mobile name can only be a string.
There should be a mechanism using which you can restrict the values being rendered on the page.
Sometimes it is preferred to have a text instead of numeric values to represent some information. For example, on the above page instead of categorizing the mobiles based on their prices, you can prefer to remember them as some text like GoldPlatinum, PinkGold, etc.
TypeScript is a static typed language that allows us to declare variables of various data types so that you ensure only the desired type of data being used in the application.
Let us discuss declaring variables and basic data types supported by TypeScript.

Declaring Variables

Declaration of a variable is used to assign a data type and an identifier to a variable. Declaration of variables in TypeScript is like JavaScript variable declaration.
Below are the three declaration types supported in TypeScript.

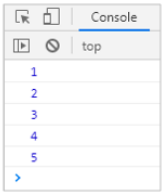
Data Type	Explanation
var	• Variable declared with this type would have function scope. • They can be re-assigned and re-defined. • When a variable declared outside the function, it would have global scope and automatically attaches itself to the window object.
let	• Variable declared with this type would have a block-level scope. • They can be re-assigned and cannot be redefined.
const	• Variable declared with this type would have a block-level scope. • They can be neither re-assigned nor re-defined.

Problem with var declaration:

In the below code, you will observe a strange behavior of the variable count. You can access this variable even outside the loop although it is defined inside the loop. This is due to the global scope nature of the **var** data type declaration.

```
for (var count = 1; count < 5; count++) {
    console.log(count)
}
// count is accessible outside for loop
console.log(count)
```

Output:



Some of the other problems with global scope variable are:

- When the var declared variable is defined outside a function it is attached with the window object which has only one instance.
- This is a bad practice and causes an overridden of a variable when you try to use a third-party library working with the same variable.
- Generally, var variables have a function-scope and if they are not declared within a function, they have a global scope.

To overcome this issue, in ES2015. JavaScript has introduced **let** and **const** as two new declaration types and it is considered as standard because declaring variables by these two data types are safer than declaring a variable using the var keyword.
Let us learn about these declarations.

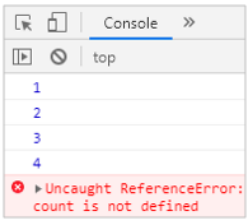
Declaring variables using let keyword

When a developer wants to declare variables to live only within the block, they can achieve a block-scoped variable using **let** declaration.

Let us replace the var with the let keyword in the previous for loop code-snippet and observe the difference.

```
for (let count = 1; count < 5; count++) {
    console.log(count)
}
// count is not accessible outside for loop
console.log(count)
```

Output:



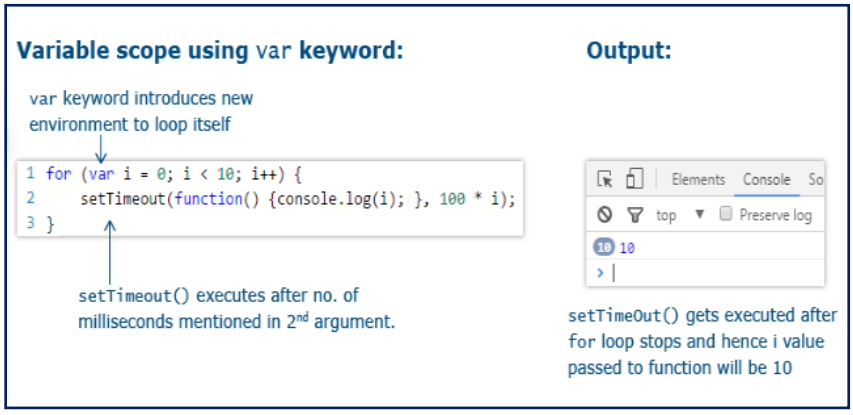
Since the count variable is a **block-scoped**, it is not accessible outside the for loop hence results in the error as not defined.

Difference between var and let Keyword

Capturing Variable in the loop

Variable declared using **var** keyword will have function scope. So, even if you declare a variable using **var** inside a loop, it will have function scope.

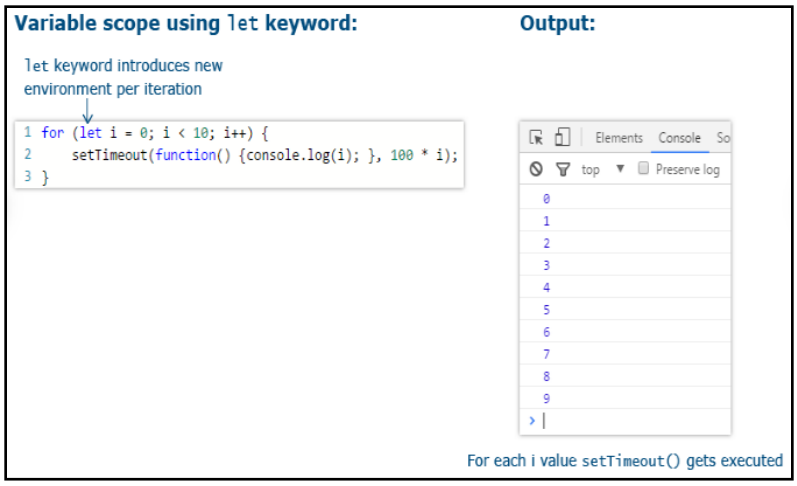
In the below example, variable **i** has been declared inside the for loop using the **var** keyword, but it will have the scope of the function. Thus, by the time **setTimeout** function executes, the value of **i** has already reached 10.



Therefore, every invocation of **setTimeout** function gets the same value of i as 10.

Variable declared using **let** keyword will have block scope. Therefore, once the block is terminated, the scope also is lost.

In the below example, variable **i** has been declared using **let** inside the for loop. So, its scope will be limited to one iteration of the loop.

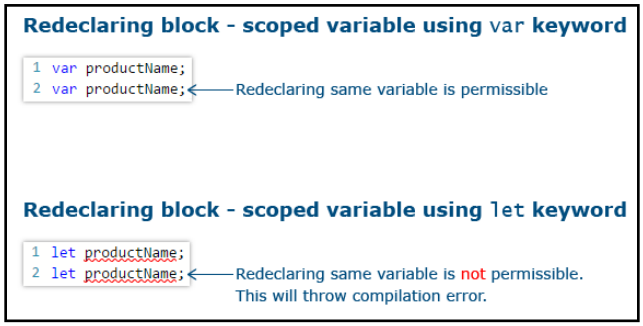


In this scenario, every invocation of the **setTimeout** function will get the value of **i** from that iteration scope.

Difference between var and let Keyword

Redeclaring block-scoped variable:

- The let declared variable cannot be redeclared within the same block.
- The var declared variable can be redeclared within the same block.



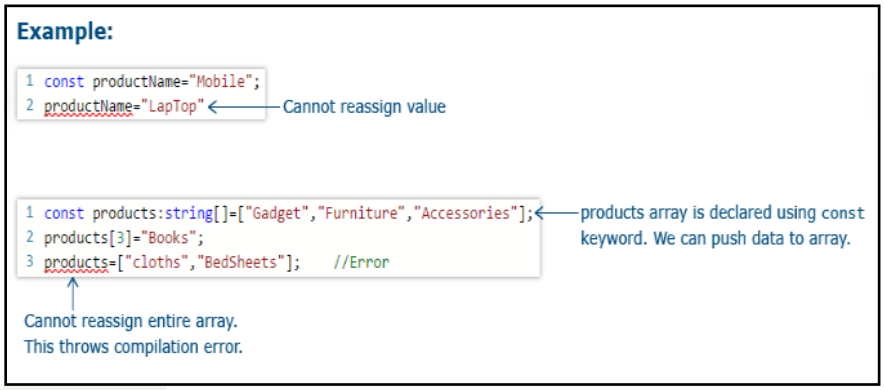
Due to these reasons let declaration is preferred over var declaration.

Now let us see what is const declaration meant for.

Declaring variables using const Keyword

The const declaration is similar to the let declaration except that the value cannot be re-assigned to the variable.

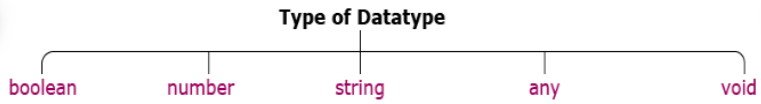
- const declared variables are mutable if declared as an array or as an object literal.
- this declaration should be used if the value of the variable should not be reinitialized.



Basic Types

Data type mentions the type of value assigned to a variable. It is used for variable declarations.

Since TypeScript supports static typing, the data type of the variable can be determined at the compilation time itself.



There are variables of different types in TypeScript code based on the data type used while declaring the variable.

boolean:

boolean is a data type that accepts only true or false as a value for the variable it is been declared.

In a shopping cart application, you can use a boolean type variable to check for the product availability, to show/hide the product image, etc.

Example:
```
let showImage: boolean = true;
```

number:

number type represents numeric values. All TypeScript numbers are floating-point values.

In a shopping cart application, you can use number type to declare the Id of a product, price of a product, etc.

Example:
```
1 let productId:number = 1045;
```

string:

A string is used to assign textual values or template strings. String values are written in quotes – single or double.

In a shopping cart application, you can use string type to declare variables like productName, productType, etc.

Example:
```
let productName:string="Samsung Galaxy J7";
```

Template strings are types of string value that can have multiple lines and embedded expressions. They are surrounded by the backquote\backtick (`) character and embedded expressions of the form ${ expr }.

Example:
```
let productName:string="Television";
let message:string=`The product name is ${productName}`;

let catalog = `The products in the catalog are
               TV
               Refrigerator
               Airconditioner
               Geyser`;
console.log(catalog);
```
Expression in template string used to get value of variable productName

Prints multi-line message in console

any:

any type is used to declare a variable whose type can be determined dynamically at runtime with the value supplied to it. If no type annotation is mentioned while declaring the variable, any type will be assigned by default.

In a shopping cart application, you can use any type to declare a variable like screenSize of a Tablet.

Example:
```
let screenSize:any;
screenSize=13.97;
screenSize="5.5-inch";
```
screenSize variable is assigned numeric value

screenSize variable is assigned string value, which is acceptable since datatype is any

void:

void type represents undefined as the value. the undefined value represents "no value". A variable can be declared with void type as below:

Example:
```
let product:void = undefined;
```
Variable declared with void datatype. It is not preferred as we can assign only undefined or null as value.

void type is commonly used to declare function return type. The function which does not return a value can be declared with a void return type. If you don't mention any return type for the function, the void return type will be assigned by default.

Function declared with void return type

Example:
```
function displayProductDetails(): void{
console.log("Product category is Gadget");
}
```

Type Annotations

Type Annotation is a way to enforce type restriction to a specific variable or a function. If a variable is declared with a specific data type and another type of value is assigned to the variable, a compilation error will be thrown.

Example:
```
let productId:number=1045;
productId="Mobile";//Error
```
productId variable is declared with type number. Using this type annotation we are ensuring that only numeric values can be assigned to it.

If we assign string value, we will get compilation error saying Type string is not assignable to type number

Function is defined with parameter of number type and string as return type

```
function getProductDetails(productId:number):string{
    return "Product ID"+productId;
}

getProductDetails(" Mobile"); //Error
```

Invoking function using string parameter type, throws compilation error saying "Argument of type 'string' is not assignable to parameter of type 'number'"

Enum:

Enum in TypeScript is used to give more friendly names to a set of numeric values.

For example, if you need to store a set of size variables such as Small, Medium, Large with different numeric values, group those variables under a common enum called Size.

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 EnumName{property1, property2, property3};
```

In a shopping cart application, you can use a MobilePrice enum to store different prices of the mobile depending on the mobile color.

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

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

Syntax: `EnumName.item` | `EnumName["item"]`

Example: `MobilePrice.Black` | `MobilePrice["Black"]`

Enum items can also be initialized with different values than the default value. If you set a different value for one of the variables, the subsequent values will be incremented by 1.

Example:
```
enum MobilePrice{Black=25000,Gold,White};
```
Initial value is set as 25000, so subsequent values will be 25001 and 25002 respectively

You can even set different values to different enum items

Example:
```
enum MobilePrice{Black=25000,Gold=28000,White=30000};
```
All 3 enum items are assigned different values

Arrays

Consider the below page of the Mobile Cart application that displays the list of mobile phones available. For each mobile in the list, you can see its image, name, price, and availability status.

This requirement is implemented using the concept of **arrays** of TypeScript. Let us discuss arrays in TypeScript.

An Array is used to store multiple values in a single variable. You can easily access the values stored in an array using the index position of the data stored in the array.

TypeScript array is an object to store multiple values in a variable with a type annotation. They are like JavaScript arrays.

Arrays can be created using one of the below:

Using datatype[] declaration:

String array is created using string[] declaration

```
let manufacturers: string[] = ["Samsung", "Apple", "Sony"];
```

Using Array<type> declaration:

String array is created using Array<type>declaration

```
let manufacturers: Array<string> = ["Samsung", "Apple", "Sony"];
```

Using any[] declaration:

It accepts any type of data

```
let products:any[]=["Mobile",12500,true];
```

A TypeScript array defined with a specific type does not accept data of different types. TypeScript arrays can be accessed and used much like JavaScript arrays.

JavaScript Arrays has several useful properties and methods to access or modify the given array. The same is supported in TypeScript.

To add a dynamic value to an array, you can either use the push function or use the index reference.

```
let products:string[]=["Mobiles","Tablets"];
products.push("Television");
products.push("Air Conditioners");
```

Adding data using push function. Make sure that the type of pushed data is same as array type, or it will generate compilation error.

or

```
let products:string[]=["Mobiles","Tablets"];
products[2]="Television";
products[3]="Air Conditioners";
```

Added data using index reference

Data can be removed from an array using the pop function or splice function

Example: Using pop() function:

```
let products:string[]=["Mobiles","Tablets","Television"];
products.pop()
```

Original array:

Mobiles	Tablets	Television

After using pop() function:

Mobiles	Tablets

or

Using splice() function:

```
let products:string[]=["Mobiles","Tablets","Television"];
products.splice(1,2)
```

Original array:

Mobiles	Tablets	Television

After using splice() function:

Mobiles

The splice function removes the item from a specific index position.

Tuple

Tuple type is a kind of array which accepts more than one predefined type of data. Arrays are used to represent a collection of similar objects, whereas tuples are used to represent a collection of different objects.

Let us consider an example, where customerCreditId, Customer object, and customerCreditLimit must be represented in a data type.
The choice could be to define a class, with these properties. If it is represented using the class, then there will be a requirement to instantiate the class and then the properties can be accessed.
Tuples provides an easy way to implement the same scenario with an array-like data structure, which is easy to access and manipulate.

Example:

```
//customerCreditInfo tuple with 3 different types of data
var customerCreditInfo: [string, Customer, number];
customerCreditInfo = ["I342", new Customer("I342"), 3000];
```

In order to access the customerCreditId, you can use customerCreditInfo[0].
In order to access the customer object, you can use customerCreditInfo[1].
In a shopping cart application, you can use tuples to store product details that should require more than one type of data.

- The order of the first set of data entries while initializing a tuple variable should be the same as the order in which the type is declared in a tuple.
- A developer can initialize only one entry as per TypeScript data restriction length policy.
- A compilation error will be thrown in below two cases:
  - if you are trying to assign multiple entries in the first initialization.
  - if you try to initialize different datatypes directly to the tuple declared variable.
- In order to overcome the above-mentioned compilation errors, you can use the push() method.

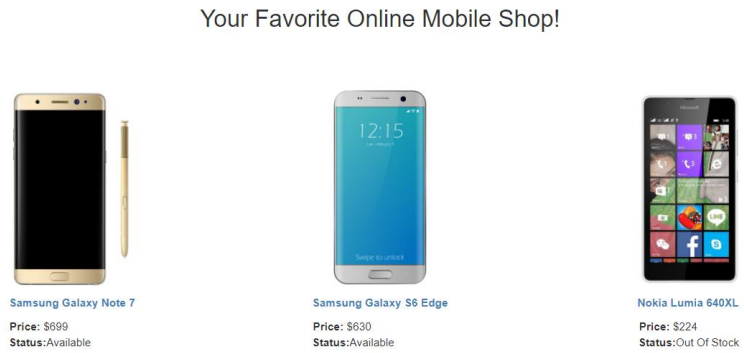
Example:

```
let productAvailable: [string, boolean];
productAvailable = ["Samsung Galaxy J7",true];
productAvailable = ["Samsung Galaxy J7", false, "Samsung Galaxy J7", false];
productAvailable.push("Samsung Galaxy J5",false);
productAvailable.push(false, "Samsung Galaxy J8");
productAvailable.push(false, "Samsung Galaxy J8",false,"Samsung Galaxy J8");
```

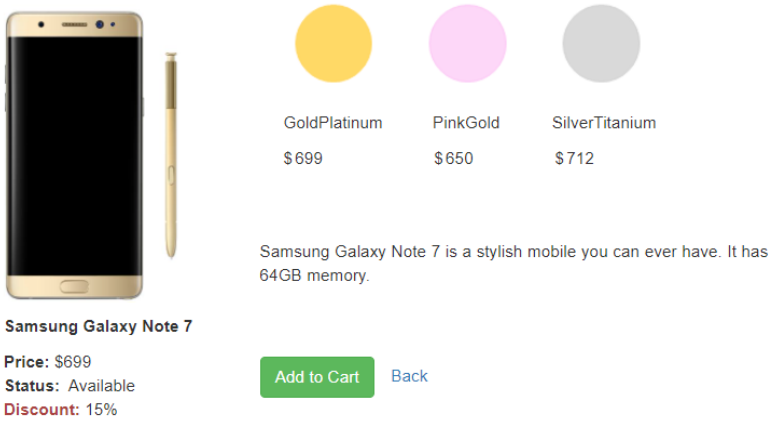
In the above example, the underlined error is due to multiple declarations in the first initialization which violates the length restriction policy. To avoid this, the push method can be used as shown in the code.

Why Function?

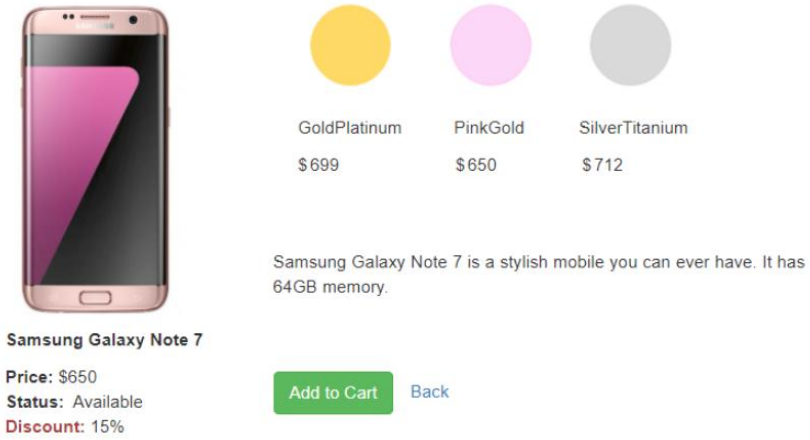
Consider the below page that displays the list of mobile phones available. For each mobile in the list, you can see its image, name, price, and availability status. The property 'name' of the mobile is clickable.



When the mobile name link is clicked, the user is navigated to the next screen which shows the details specific to the phone selected. The details such as different colors in which the phone is available, price of mobile according to the color selected, description of the phone, availability status, and discounts if any.



Users can click on each color to view the mobile image for that color. The Price corresponding to the color selected can also be shown. For example: On click of PinkGold the below screen should be rendered:



This requirement is implemented using the **function** concept in TypeScript that helps us to implement business logic and event handlers. Let us discuss functions in TypeScript.

Function in TypeScript Vs JavaScript:

- A function is a block of statements to perform a particular task.
- A sequence of statements written within function forms function body.
- Functions are executed when it is invoked by a function call. Values can be passed to a function as function parameters and the function returns a value.
- Functions in TypeScript are like functions in JavaScript with some additional features.

	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

Parameter Types and Return Types

The parameter type is the data type of the parameter and the return type is the data type of the returned value from the function.

With the TypeScript function, you can add types to the parameter passed to the function and the function return types.
While defining the function, return the data with the same type as the function return type. While invoking the function you need to pass the data with the same data type as the function parameter type, or else it will throw a compilation error.

Example:

```
function getProductDetails(productId:number):string{
    return "Product ID"+productId;
}

getProductDetails("Mobile");//Error
```

Function is defined with parameter of number type and string as return type

Invoking function using string parameter type, throws compilation error saying "Argument of type 'string' is not assignable to parameter of type 'number'"

Arrow Function

Arrow function is a concise way of writing a function. Whenever you need a function to be written within a loop, the arrow function will be the opt choice.
Do not use the function keyword to define an arrow function.
In a shopping cart application, you can use the arrow function to perform filtering, sorting, searching operations, and so on.

Syntax: `(parameter)=>function body`

Example:

```
var getProductDetails = (productId: number): string => { return "Product ID" + productId; }
```

Arrow function with number parameter, string return type and function body

Handling 'this' keyword in JavaScript

Example:

```
class Product{
    productName:string="Mobile";
    getProductDetails():string{
        return "Product: "+this.productName;
    }
    testThisFunction(){
        setTimeout(function(){
            console.log(this.productName);
        },100);  //Error
    }
}
```

It has Product class scope, so we can access productName

It has current function scope as it is used within call back function. Hence we can't access productName declared within Product class. This will log in console as undefined output.

- In a class, if a method wants to access the property of the class it should use this keyword.
- For a particular object, this keyword will help to access the properties of the current object. This is possible because all the methods and properties are within the same scope.
- In the above example, when you use this.productName inside the getProductDetails method, getProductDetails method, and productName variable are in the same scope. Also, you get the desired result.
- But when you use this.productName inside the setTimeout function, instead of directly using it in testThisFunction method, the scope of this.productName will be inside the

setTimeout's callback function and not the testThisFunction method. That is the reason you are unable to access the value of productName for that particular object.

- If you need to access the class scope with this keyword inside the callback function then use the arrow function.
- Arrow function lexically captures the meaning of this keyword.

Rewrite the same logic using the arrow function as below:

Example:

```
class Product{
     productName:string="Mobile";
     getProductDetails():string{
         return "Product: "+this.productName;
     }
     testThisFunction(){
        setTimeout(
            ()=>{ console.log(this.productName);},100);
     }
}
```

It has Product class scope, so we can access productName

It has Product class scope as it is defined using arrow function. Hence we can access productName declared with in Product class.

In the above code, this.productName is written inside an arrow function. Since the callback function of setTimeout is implemented using the arrow function, it does not create a new scope and it will be in the same scope as the testThisFunction method.

Function Types

Function types are a combination of parameter types and return type. Functions can be assigned to variables.

While assigning a function to a variable make sure that the variable declaration is the same as the assigned function's return type.

Example:

```
function getProductDetails(productName:string):string{
         return "Product: "+ productName;
     }
//correct way of assigning
let productName:string=getProductDetails("Mobile");

//Incorrect way of assigning
let productName:number=getProductDetails("Mobile");//Error
```

Assigning function with string return type to number type variable. This throws compilation error saying "Type 'string' is not assignable to type 'number'"

Optional and Default Parameters

TypeScript treats every function parameter as mandatory. So when a function is compiled, the compiler will check whether there is a value supplied to all the parameters of the function, or else it will throw a compilation error.

Consider the code below:

Example:

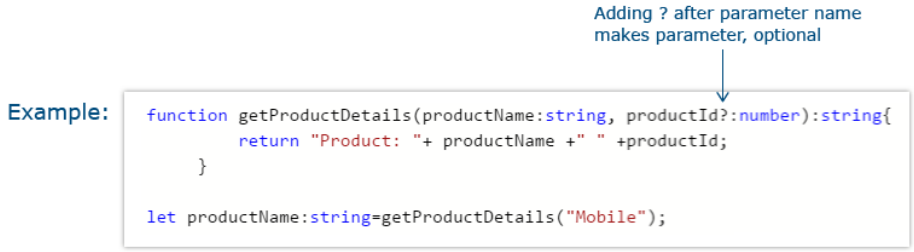
```
function getProductDetails(productName:string,productId:number):string{
         return "Product: "+ productName +" " +productId;
     }

let productName:string=getProductDetails("Mobile"); //Error
```

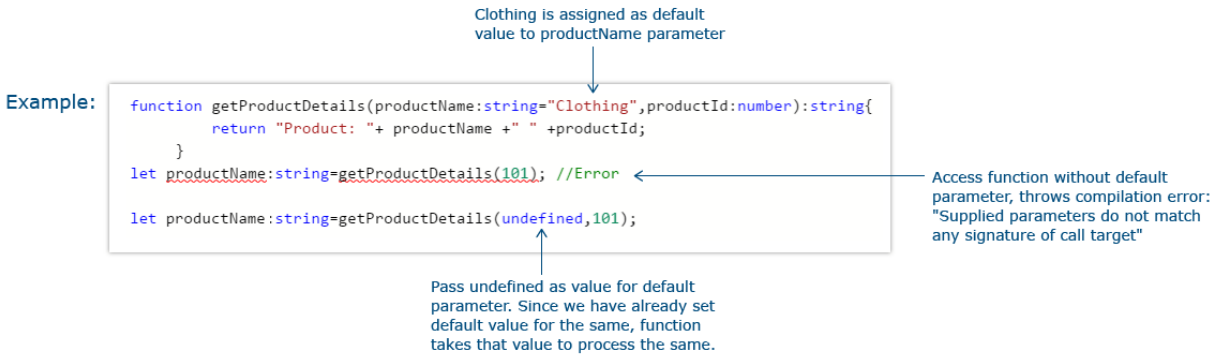
Invoking getProductDetails function with single parameter. This throws compilation error "Supplied parameters do not match any signature of call target"

In the above example, you have tried to invoke a function with only a single parameter, whereas the

definition of the function accepts two parameters. Hence, it will throw a compilation error. Also, optional parameter can be used to tackle this issue.
The Optional parameter is used to make a parameter, optional in a function while invoking the function.
If you rewrite the previous code using an optional parameter, it looks like the below:



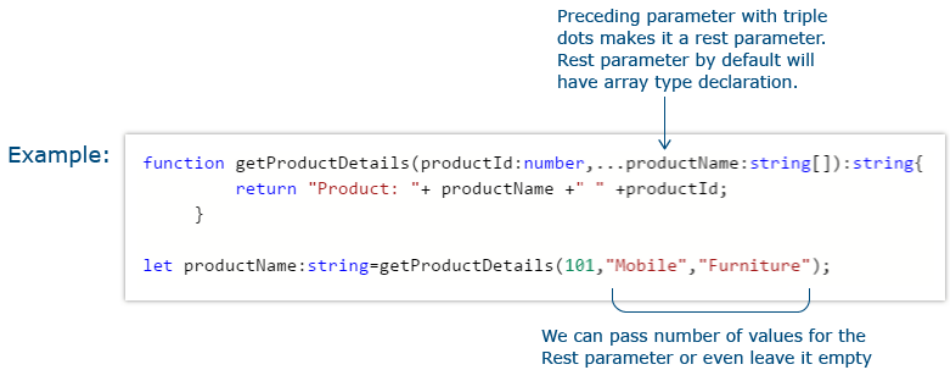
- An Optional parameter must appear after all the mandatory parameters of a function.
- Default parameter is used to assign the default value to a function parameter.
- If the user does not provide a value for a function parameter or provide the value as undefined for it while invoking the function, the default value assigned will be considered.
- If the default parameter comes before a required parameter, you need to explicitly pass undefined as the value to get the default value.



Rest Parameter

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.
- Precede the parameter to be made as rest parameter with triple dots.
- Rest parameter should be the last parameter in the function parameter list.

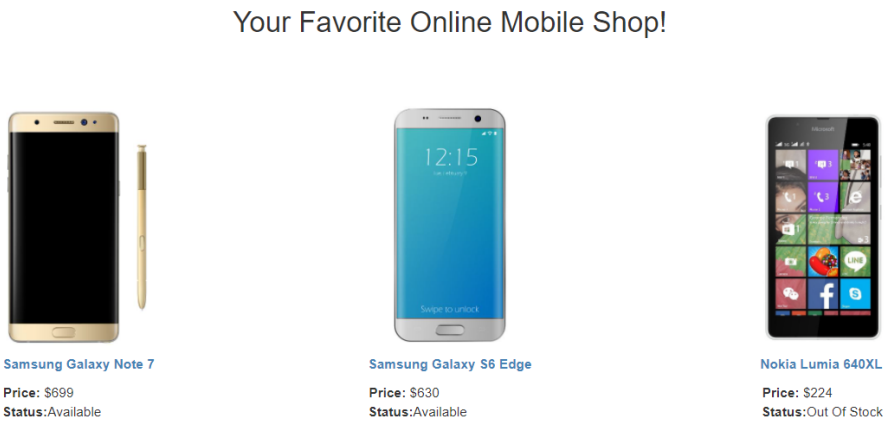


Why Interface?

- Interfaces can be used to impose consistency among various TypeScript classes.

- Any class which implements an interface should implement all the required members of that interface.
- Interfaces can be used to ensure that proper values are being passed into functions, properties as well as constructors.
- Interfaces can be used to achieve additional flexibility as well as loosely coupling in an application.
- Any object which implements an interface can be passed as a parameter to a function whose parameter type is declared the same as the interface.

Consider the below screen of the Mobile Cart application:



Here an array is used to store the product information. But not restricted the type of object to be stored in the array. You can restrict the array which contains only a particular type of object. For this, use Interface.

Let us discuss more on Interface in TypeScript.

What is an Interface?

An interface in TypeScript is used to define contracts within the code.

- Interfaces are used to enforce type checking by defining contracts.
- It is a collection of properties and method declarations.
- Interfaces are not supported in JavaScript and will get removed from the generated JavaScript.
- Interfaces are mainly used to identify issues upfront as we proceed with coding.

How to create an Interface?

Syntax:
```
interface Interfacename{
    properties;
    method declarations;
}
```

Example:
```
interface Product{
    productId:number;
    productName:string;
}

function getProductDetails(productobj:Product):string{
}

//Correct way of using the interface type
let productobj={productId:1001,productName:'Mobile'};

//InCorrect way of using the interface type
let productobj={productCategory:'Gadget',productName:'Mobile'};

getProductDetails(productobj);
```

Declaring interface with name Product. Function or object which is going to use this interface should contain properties declared in interface

Passing object with type Product interface as parameter to function. If the object passed does not have any of the properties declared in the Interface it throws compilation error

As this declaration does not have productId property of interface, it throws compilation error

Duck Typing

Duck-Typing is a rule for checking the type compatibility for more complex variable types.

TypeScript compiler uses the duck-typing method to compare one object with the other by comparing that both the objects have the same properties with the same data types.
TypeScript interface uses the same duck typing method to enforce type restriction. If an object that has been assigned as an interface contains more properties than the interface mentioned properties, it will be accepted as an interface type and additional properties will be ignored for type checking
Let us rewrite the previous example to a pass additional parameter.

Example:

```
interface Product{
    productId:number;
    productName:string;
}

function getProductDetails(productobj:Product):string{


}
//InCorrect way of using the interface duck type
let prodObject={productName:'Mobile',productCategory:'Gadget'};

//Correct way of using the interface duck type
let prodObject={productId:1001,productName:'Mobile',productCategory:'Gadget'};

getProductDetails(prodObject);
```

Adding an additional property productCategory to demonstrate Duck Typing

Defining an Interface

- Interface keyword is used to declare an interface.
- An interface should have properties and method declarations.
- Properties or methods in an interface should not have any access modifiers.
- Properties cannot be initialized in a TypeScript interface.

Example:

```
interface Product{
    productId:number;
    productName:string;
    getProductDetails(productId :number):string;
    displayProductName:(prouctId:number)=>string;
}
```

Function declared using arrow function

Defining an Interface – Optional Property

In a TypeScript interface, if certain properties need to be made optional, you can make them optional by adding '?' after the property name.
Let us rewrite the duck typing example with the optional property.

Example:

```
interface Product{
    productId?:number;
    productName:string;
}

function getProductDetails(productobj:Product):string{

    return "The product name is "+productobj.productName;
}

let prodObject={productName:'Mobile',productCategory:'Gadget'};

getProductDetails(prodObject);
```

Making productId property optional by adding ? after property name

Does not throw compilation error as productId property is optional

Function Types

Interfaces can be used to define the structure of functions like defining structure of objects.
Once the interface for a function type is declared, you can declare variables of that type and assign functions to the variable if the function matches the signature defined in the interface.
Function type interface is used to enforce the same number and type of parameters to any function which is been declared with the function type interface.

Example:

```
function CreateCustomerID(name: string, id: number): string{
}

interface StringGenerator{
(chars: string, nums: number): string;
}

let IdGenerator: StringGenerator;

IdGenerator= CreateCustomerID;

IdGenerator("Infy",101);
```

Defining interface with function declaration. Only parameter list and return type is given.

Declaring variable with interface type

Assigning function to interface typed variable to enforce type on function

Invoking function by passing parameter to function type variable. If we are not passing parameters with declared type then compilation error will be thrown.

Extending Interfaces

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

In the code below, extend the productList interface from both the Category interface and Product interface.

Example:

```
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']
}
```

Extending productList interface from two other interfaces Category and Product

Creating object literal productDetails with productList interface type. Throw compilation error, if we do not define properties declared in super interfaces.

Class Types

Make use of the interface to define class types to explicitly enforce that a class meets a particular contract. Use implements keyword to implement interface inside a class.

To enforce interface type on a class, while instantiating an object of a class declare it using the interface type.

The only interface declared functions and properties will be available with the instantiated object.

Example:

```
interface Product{
    getProductDetails(productId :number):string;
    displayProductName:(prouctId:number)=>string;
}

class Gadget implements Product{
    getProductDetails(): string[]{
    }
    displayProductName(productId:number):string{
    }
}

var g:Product=new Gadget ();
```

Implementing interface Product with class Gadget. We should define all the functions declared within the interface inside class, otherwise we will get compilation error.

Enforcing type on class while declaring class variable. Here we are declaring class variable using interface type and enforcing interface type on the object instantiated.

Why Classes

Classes are used to create reusable components. Till the ES5 version of JavaScript, you do not have a class concept as such. For implementing reusable components, use functions and prototype-based inheritance. TypeScript provides an option for the developers to use object-oriented programming with the help of classes.

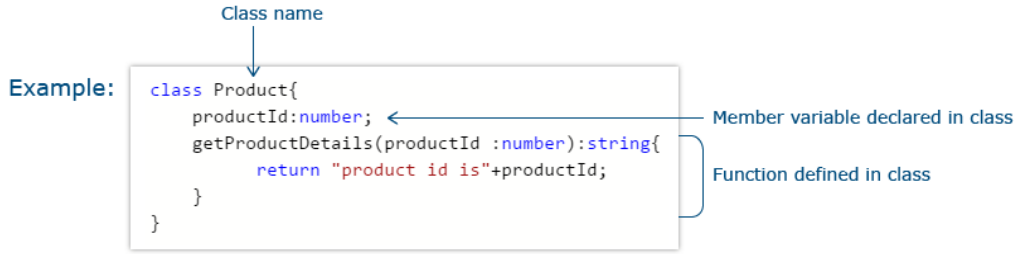
In the Mobile Cart application, you can use a class to define the product and create various objects of different products. In the below screen, creating different objects of product and rendering the details

Let us discuss more on classes in TypeScript.

What is a Class?

- Class is a template from which objects can be created.
- It provides behavior and state storage.
- It is meant for implementing reusable functionality.
- Use a class keyword to create a class.

Example:

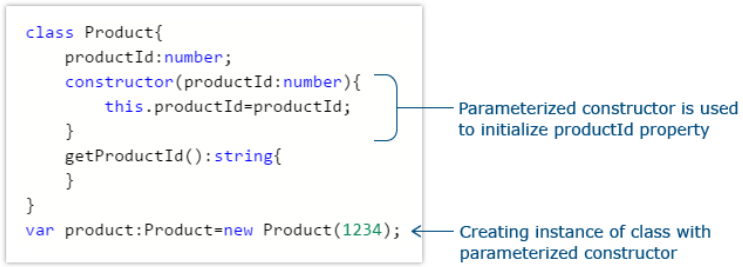


Constructor

A constructor is a function that gets executed automatically whenever an instance of a class is created using a new keyword.

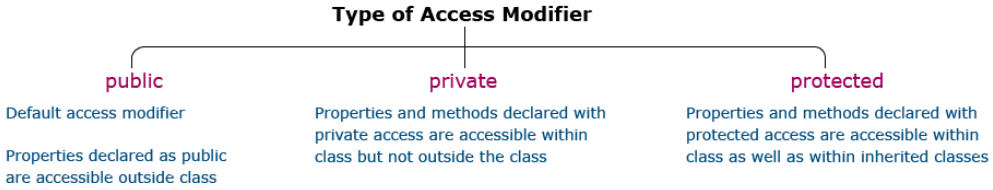
- To create a constructor, a function with the name as a "constructor" is used.
- A class can hold a maximum of one constructor method per class.
- You can use optional parameters with a constructor function as well.

Example:



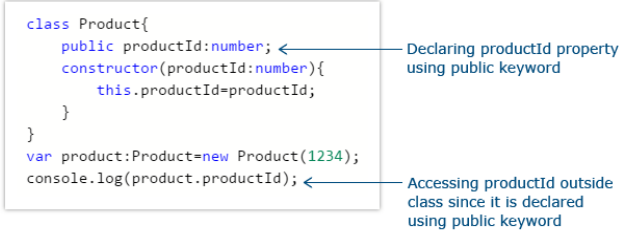
Access Modifiers

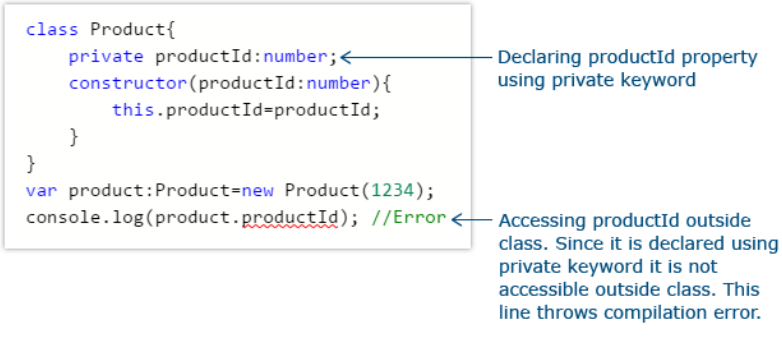
Access modifiers are used to provide certain restriction of accessing the properties and methods outside the class.



Public and Private Access Modifiers

Example:





Protected Access Modifier

Example:

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

Declaring productId property using protected keyword

Accessing productId inside inherited class since it is declared using protected keyword

Static Access Modifier

TypeScript provides an option to add a static keyword. This keyword can be used to declare a class variable or method.

- A static variable belongs to the class and not to the instance of the class.
- A variable or function declared with a static keyword can be accessed using the class name instead of the instance of the class.
- Static variables are initialized only once, at the start of the execution.
- A single copy of the static variables would be shared by all instances of the class.
- A static variable can be accessible inside the static function as well as the non-static function.
- A static function can access only static variables and other static functions.

Example:

```
class Product{
    static productName:string="Mobile";

    static getProductDetails():string{
        return "Product Name is"+Product.productName;
    }

    getProduct():string{
        return "Product Name is"+Product.productName;
    }
}
Product.productName="Tablet";
console.log(Product.productName);
console.log(Product.getProductDetails());
```

Declaring productName property using static keyword

Declaring getProductDetails function using static keyword and accessing productName property within it

Accessing static property within non-static function

Setting value for static property and accessing the same using property name and function name

Properties and Methods – Parameter Properties

Instead of declaring instance variables and then passing parameters to the constructor and initializing it, you can reduce the code by adding access specifiers to the parameters passed to the constructor.
Consider the below code:

Example:

```
class Product{
    private productId:number;
    constructor(productId:number){
        this.productId=productId;
    }
}
```

Passing parameter to constructor and setting private property with value passed to constructor

Instead of this declare the parameter itself with any of the access modifiers and reduce the lines of code used for the initialization.
Let us rewrite the above code:

Example:

```
class Product{
    constructor(private productId:number){
    }
}
```

Initial line of declaration and setting of value inside constructor is removed and parameter is declared with private keyword. The same is applicable for public or protected keywords as well.

Properties and Methods- Accessors

TypeScript provides an option to add getters/setters to control accessing the members outside the class or modifying the same.

There is no direct option available to access private members outside the class but use accessors in TypeScript for the same.

If you need to access private properties outside the class, you can use the getter and if you need to update the value of the private property, you can use the setter.

Example:

```
class Product {
     productName: string;
}
let product = new Product();
product.productName = "Fridge";
if (product.productName) {
    console.log(product.productName);
}
```

Setting value of productName property outside class. This allows modifying the productName property outside the class. However, if the productName property is declared as private, then it cannot be accessed outside the class.

```
class Product {
    private _productName: string;

    get productName(): string {
        return this._productName;
    }

    set productName(newName: string) {

            this._productName= newName;
    }

let product = new Product();
product.productName = "Fridge";
if (product.productName) {
    console.log(product.productName);
}
```

Adding getter and setter using get and set keyword before the function name. From the get function we are returning the _productName private property. Inside the set function we are setting the value of _productName.

set function will get executed automatically whenever we try to assign new value to property with same name as set function name.

get function will get executed automatically whenever we try to access property with same name as get function name

Extending Classes with Inheritance

Inheritance is the process of extending a class from an already existing class and reuse the functions and properties of the inherited class in the subclass.

TypeScript supports inheritance with class, wherein the superclass functionalities can be reused within the subclass.

The subclass constructor function definition should invoke the superclass constructor using the super function.

Example:

```
class Product{
    protected productId:number;
    constructor(productId:number){
        this.productId=productId;
    }
}
class Gadget extends Product{
    constructor(public productName:string,productId:number){
        super(productId);
    }
    getProduct():void{
    }
}
```

Class Gadget is extended from the class Product using extends keyword

super function is used to invoke superclass constructor inside the subclass constructor. If the subclass has a constructor defined, then it is mandatory to invoke the super class constructor using the super function.

Use the super keyword to access the methods of the super class inside the subclass methods.
Override the superclass methods inside the subclass by specifying the same function signature.

Example:

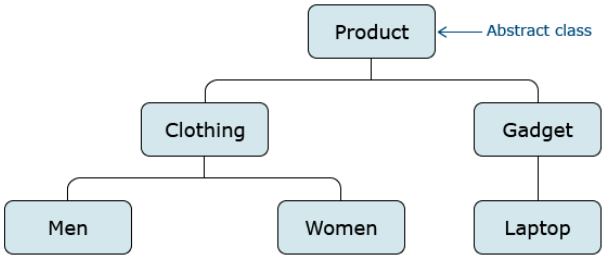
```
class Product{
    protected productId:number;
    constructor(productId:number){
        this.productId=productId;
    }
    getProduct():void{
    }
}
class Gadget extends Product{

    getProduct():void{
        super.getProduct();
    }
}
```

Using super keyword to access the superclass getProduct function inside the subclass overridden function

Abstract Class

- Abstract classes are base classes that may not be instantiated.
- An abstract class can be created using abstract keyword.
- Abstract methods within an abstract class are methods declared with abstract keyword and does not contain implementation.
- They must be implemented inside the derived classes.
- Abstract classes can contain both abstract methods and its implementations.



Example:

```
abstract class Product{
    getFeatures():void{
    }
    abstract getProductName():string;
}

class Gadget extends Product{
getProductName():string{
    }
}

class Clothing extends Product{
    getProductName():string{
    }
}

var g=new Gadget();
g.getProductName();

var c=new Clothing();
c.getProductName();
```

Creating abstract class Product with abstract method getProductname and non-abstract method getFeatures

Extending two classes Gadget and Clothing from the Product abstract class and implementing abstract method getProductname

Creating instance for the classes Gadget and Clothing and trying to access getProductName function implemented within each class

Modules and Namespaces

- Modules and Namespaces are useful in grouping functionalities under a common name.
- The main use is reusability.
- Code can be reused by importing modules or namespaces in other files.
- Namespaces are used for namespace resolution and are suitable for the development of a smaller application.
- In larger-scale applications, they can be used to achieve modularity.

TypeScript provides native support in terms of module loaders using modules concept which takes care of all the concerns with respect to modularity.

In the MobileCart application, create a product utility namespace or module and place the code related to the product in it. Reuse the code related to the product by importing it into other files.

What is Namespace?

A Namespace is used to group functions, classes, or interfaces under a common name.

- The content of namespaces is hidden by default unless they are exported.
- Use nested namespaces if required.
- The function or any construct which is not exported cannot be accessible outside the namespace.

Creates namespace

Syntax:

```
namespace namespacename{
    export namespace namespacename{
        export function functionname {
        }
    }

    export function functionname{
    }
}
```

We can have a namespace nested within another namespace and export the inner namespace if required

Precede the function or interface or class which you need to export with an export keyword

Creating and using Namespaces

In the below example, create a namespace called Utility and group a function MaxDiscountAllowed and a nested namespace called payment inside it.

Example:

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

    export function MaxDiscountAllowed(noOfProduct: number): number {
    }

    function privateFunc(): void {
    }
}
```

Nested namespace

Private function which is not been exported

To import the namespace and use it, make use of the triple slash reference tag.

Syntax: `/// <reference path="./namespacefilename.ts" />`

Importing namespace using triple slash reference tag

Path name represents filename of namespace definition

Example:

```
/// <reference path="./namespace_demo.ts" />

import util = Utility.Payment;

let paymentAmount = util.CalculateAmount(1200,6);
console.log(`Amount to be paid: ${paymentAmount}`);

let discount=Utility.MaxDiscountAllowed(6);
console.log(`Maximum discount allowed is: ${discount}`);

Utility.privateFunc(); //Error
```

Importing nested namespace

Invoking exported namespace function

Invoking non-exported function will throw compilation error

The file in which the namespace is declared and the file which uses the namespace to be compiled together. It is preferable to group the output together in a single file. You have an option to do that by using the --outFile keyword.

Syntax: `tsc --outFile Finalfilename.js namespacefilename1.ts namespacefilename2.ts`

Example:

```
D:\courseware\Typescript\ILP\demos\modules>tsc --outFile out_file.js namespace_d
emo.ts 1.namespace_util.ts

D:\courseware\Typescript\ILP\demos\modules>
```

Before compilation

After compilation

What is a Module?

Modules help us in grouping a set of functionalities under a common name. The content of modules cannot be accessible outside unless exported.

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

Filename will be used as modulename when it is been imported in other file

Syntax: **Modulename.ts**

Preceding function name with export keyword

```
export function functionname {}
export class classname{}
export interface interfacename{}
export const constname:type=value;
```

Exporting from a Module

The constructs of a module can be exported by one of the below approaches:

1. Adding an export keyword in front of a function or a class or an interface
2. Adding an export keyword to a block of statements

Preceding function name with export keyword

Example:

```
export function MaxDiscountAllowed(noOfProduct:number): number {
}
class Utility {
    CalculateAmount(price: number, quantity: number): number {
    }
}
export interface Category {
}
export const productName: string = "";

export {Utility};
```

Exporting interface

Exporting constant

Exporting class using export block. We can export all constructs together using export block instead of adding export keyword individually.

Importing a Module

Using the import keyword, you can import a module within another module or another TypeScript file. Provide the file name as the module name while importing.

Once the module is imported, make use of the classes and other constructs exported from the module.

import keyword is used to import constructs which is exported from module

module filename is considered to be the modulename

Example:

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

let util =new Utility();
let price=util.CalculateAmount(1300,4);

let discount=MaxDiscountAllowed(2);

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

Once module is imported we can use Classes and other constructs exported from that module

Also, alias names can be used while importing/exporting a module.

Using alias name for class while exporting

Example:
```
export { Utility as mainUtility };
```

Importing exported alias class name

```
import { mainUtility } from "./module_demo";
```

Giving alias name while importing

```
import { Utility as mainutil} from "./module_demo";
```

Using * to assign it to an alias name to import all the constructs of a module

```
import * as mainutil from "./module_demo";
```

Compiling Module

To compile the modules and the file which is using the module, compile them together using the tsc command.

Syntax: `tsc modulefile1.ts modulefile2.ts moduleusingfile.ts`

Example:
```
D:\courseware\Typescript\ILP\demos\modules>tsc module_demo.ts 2.module_import.ts
```

Module Formats and Loaders

Module format is the syntax that is used to define a module in JavaScript.
Modules in TypeScript can be compiled to one of the module formats given below:

Supported Formats:
- AMD – Asynchronous Module Definition
- CommonJS
- ES2015 – EcmaScript 2015
- System
- UMD – Universal Module Definition

Commonly we compile the modular code in TypeScript to ES2015 format by using the --module ES2015 keyword while performing the compilation.

Syntax: `tsc filename.ts --module ES2015`

If none of the formats are mentioned in the compiler option, by default CommonJS module format will be the one that gets generated while compiling modules in TypeScript.
A module loader interprets and loads a module written in a certain module format as well as helps in importing all the dependent modules into developer working environment. At the runtime, they play a very vital role in loading and configuring all needed dependencies modules before executing any linked module.
The most commonly used module loaders in JavaScript are:
SystemJS module loader for modules in AMD, CommonJS, UMD, or System.register format

Require.js module loader for modules in AMD format.

On-demand functionalities can be loaded by Modules, which is known as lazy loading. By using this feature while executing our application all the declared modules are not loaded at that moment, it only loads needed modules that are needed by the user to render the initial look of the application on the first load. Due to this concept performance of the application can be enhanced as the initial startup time of the application automatically decreases.

Default Exports

- Default export is used to export any one of the constructs such as class, interface, function, and so on from the current module as a default one.
- Default exports are handy in case you need to export a class that has almost all functionalities attached, such as javascript library object jQuery.
- Name for the default exported constructs are optional. You can assign a name to the default construct while importing it into the other file.
- As you cannot have more than one default export per module.

To make a default export add default keyword along with export keyword. We can add this in front of any one of the module construct.

Syntax:
```
export default class{}

export class classname{}
```
← Non default export class

Example:
```
import Product from './default_export';
```
To import default export give any name along with the import keyword. Instead of enclosing it within a curly block like other named exports, give name directly.

Example:
```
export default class{
    let productName: string="Tablet";
    getProductDetails(productId:number):string{
    }
}
export class utility{
}
```

While importing, non-default export class name should be enclosed within curly braces

```
import Product, {Utility} from './default_export';
let product = new Product();
let details=product.getProductDetails(1001);
```
← Once default class is exported we can use it by creating instance of the class

Module Vs Namespace:

Module	Namespace
Organizes code	Organizes code
Have native support with Node.js module loader. All modern browsers are supported with a module	No special loader required

loader.	
Supports ES2015 module syntax	ES2015 does not have a Namespace concept. It is mainly used to prevent global namespace pollution.
Suited for large-scale applications	Suited for small-scale applications

Why Generics?

Generics helps us to avoid writing the same code again for different data types.

In TypeScript, generics are used along :

- with function to invoke the same code with different type arguments
- with Array to access the same array declaration with a different set of typed values
- with Interface to implement the same interface declaration by different classes with different types
- with a class to access the same class with different types while instantiating it

What is Generics?

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

It accepts type parameters for each invocation of a function or a class or an interface so that the same code can be used for multiple types.

Consider the below code where you implement a similar function for two different types of data:

```
function printString(stringData:string):string{
return stringData;
}
```

```
function printNumber(numberData:number):number{
return numberData;
}
```

Avoid writing the same code again for different types using generics. Let us rewrite the above code using generics:

```
function printData<T>(data:T):T{
return data;
}
```

<T> represents type parameter

It generalizing type of parameter and function return type.

Same code works for number or string or any other type of parameter.

What are Type Parameters?

Type Parameters are used to specify the type, a generic will work over.

They are listed using an angle bracket<>.

The actual type will be provided while invoking function or instance creation.

Consider a generic function given below:

```
function printData<T>(data:T):T{
return data;
}
```

To access this function with different types you will use the below code:

```
let data = printData<string>('Hello Generics');
console.log(data); //string

let data = printData('Hello Generics');
console.log(data); //string

let data1 = printData(123);
console.log(data1); //number
```

Invoking function using type parameter by passing type inside the <> bracket. In this case, type of parameter and function return type will be dependent on type parameter

Invoking generic function without type parameter. Here type is decided depending on parameter type.

Invoking generic function with number type parameter

Generic Array:

Using Array<T>

Array<T> provides us the option to use the same array declaration for different types of data when used along with a function.

<T> here represents the type parameter.

Creating generic function to accept any type of array as parameter and return type

Syntax:
```
function functionname<T>(arg: Array<T>): Array<T> {}
```

Creating generic function to accept any type of array as parameter and return type

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

```
let orderid:Array<number>=[101,102,103,104]; //number array
let idList=orderDetails(orderid);

let ordername:Array<string>=['footwear','dress','cds','toys'];
let nameList=orderDetails(ordername); //string array
```

Declaring an array with type restriction as number and passing the same as parameter to the generic function

Declaring a string type array and passing the same as parameter to the generic function

Generic Functions

Generic functions are functions declared with generic types.

Declaring generic function is done using the type parameter and using the same type variable for the parameter and the return type.

Parameter and return type are marked with generic type

Syntax:
```
function functionName<T>(arg:T):T
```

Type parameter

Example:
```
function printData<T>(data:T):T{
return data;
}
let data:string=printData<string>('Hello Generics');

class Product{
    productName:string;
}

let productData:Product={productName:'Tablet'};
let data2:Product=printData<Product>(productData);
```

Invoking generic function using string type or any other primitive type

Invoking generic function using user defined object

Generic Interface

Generic interface is an interface that works with multiple types.

```
Syntax: interface InterfaceName<T> {
            functionname(arg T):T;
            variablename:T;
        }
```

Example:

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

This interface can be implemented by different classes with their own types.

Implementing interface with string type

Example:

```
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();
```

Implementing interface with number type

```
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();
```

Generic Class

Generic class is a class that works with multiple types.

```
Syntax: class className<T> {
            functionname(arg T):T;
            variablename:T;
        }
```

Example:

```
class Gadget <T>{
    productList:Array<T>;
addItems(newItemList:Array<T>):void{
    this.productList=newItemList;
    console.log("Item added");
}
getProductList():Array<T>{
return this.productList;
}
}
```

Generic type annotation added to property and function declarations

Consider the generic class given below:

Example:

```
let product=new Gadget<string>();
let productList:Array<string>=["Mobile","Tablet","Ipod"];
product.addItems(productList);
let allProducts:Array<string>=product.getProductList();
console.log(allProducts);
```

Instantiating generic class with string type and using same type in all generic constructs of class

```
let product2=new Gadget<number>();
let shippingList:Array<number>=[123,234,543];
product2.addItems(shippingList);
let allItems:Array<number>=product2.getProductList();
console.log(allItems);
```

Instantiating generic class with number type and using same type in all generic constructs of class

The same class instance can be instantiated and invoked with different type parameters.

Generic Constraints

Generic constraints are used to add constraints to the generic type.

Generic constraints are added using the 'extends' keyword.

Generic constraints are added with extends keyword

```
Syntax: class className<T extends constraint type> {
            functionname(arg T):T;
            variablename:T;
        }
```

Consider the below code:

Here you are trying to access the length property of the function type parameter.

Since the parameter type will get resolved only at run time, this line will throw compilation

Example:

```
function orderDetails<T>(arg:T): T{
    console.log(arg.length);
    return arg;
}
```

Creating generic function and accessing length property of parameter

Since parameter type will get resolved only at run time, this line throws compilation error "Property length does not exist on type T"

To resolve this, you can add a constraint with the type parameter.

If you need to access a property on the type parameter, add those properties in an interface or class and extend the type parameter from the declared interface or class.

Let us rewrite the previous code:

```
interface AddLength{
    length:number;
}

function orderDetails<T extends AddLength>(arg:T): T{
    console.log(arg.length);
    return arg;
}
```

Adding constraint by extending from AddLength interface having length property

To invoke this generic function, you can pass any parameter which has a length property.

Implementing interface in class and passing class instance as function parameter. Since class has length property it executes successfully.

```
class Product implements AddLength{
    length:number=10;
}

let product:Product=new Product();
let product1=orderDetails(product);

let ordername:Array<string>=['footwear','dress','cds','toys'];
let nameList = orderDetails(ordername):
```

Passing an array as parameter. Since array has length property by default this code works fine.