

Agenda

- Overview
- Why TypeScript?
- How to use TypeScript?
- TypeScript Features.
- TypeScript Environment setup
- First TypeScript Program: Compile and Run
- TypeScript - Type Annotations
- TypeScript Data Type
- Interface
- Class
- Generics

Overview

- **TypeScript** is an open-source, object-oriented language developed and maintained by **Microsoft**.
- **TypeScript** extends JavaScript by adding data types, classes, and other object-oriented features with type - checking.
- It is a typed superset of JavaScript that compiles to plain JavaScript.
- Official website: <https://www.typescriptlang.org>

Overview (cont.)

TypeScript Version History

Version	Released Date
TypeScript 0.8	October 2012
TypeScript 0.9	June 2013
TypeScript 1.0	October 2014
TypeScript 2.0	September 2016
TypeScript 3.0	July 2018
TypeScript 4.0 - latest release	August 2020

TypeScript 5.0 is now available

Why TypeScript?

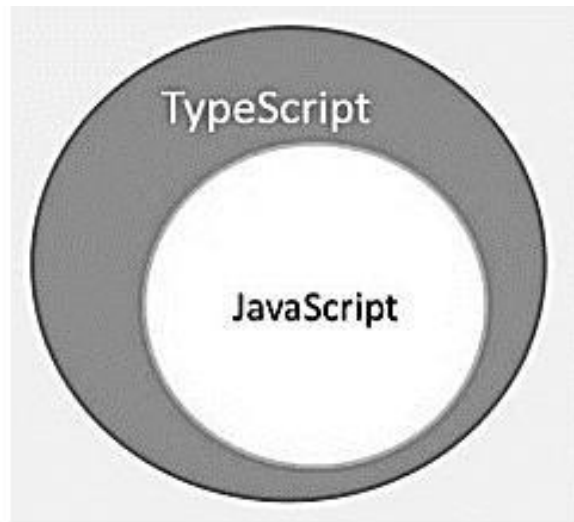
- **JavaScript** is a dynamic programming language with **no type system**.
- JavaScript provides primitive types like **string, number, object, etc.**, but it doesn't check assigned values.
- **JavaScript variables** are declared using the **var** keyword, and it can point to any value.
- JavaScript doesn't support classes and other object-oriented features.
- **So, without the type system**, it is not easy to use JavaScript to build complex applications with large teams working on the same code.

Why TypeScript?

- **The type system** increases the code quality, readability and makes it easy to maintain and refactor codebase.
- More importantly, **errors** can be caught at compile time rather than at runtime.
- Hence, the reason to use TypeScript is that it **catches errors** at compile-time, so that you can fix it before you run code.
- **It supports** object-oriented programming features like data types, classes, enums, etc., allowing JavaScript to be used at scale.

Why TypeScript?

- **TypeScript** compiles into simple JavaScript.
- **The TypeScript compiler** is also implemented in TypeScript and can be used with any browser or JavaScript engines like **Node.js**.



How to use TypeScript?

- **TypeScript code** is written in a file with **.ts** extension and then compiled into JavaScript using the TypeScript compiler.
- **A TypeScript file** can be written in any code editor.
- **A TypeScript compiler** needs to be installed on your platform.
- Once installed, the command **tsc <filename>.ts** compiles the TypeScript code into a plain JavaScript file.
- JavaScript files can then be included in the HTML and run on any browser.



Compile TypeScript to JavaScript

TypeScript Features

- **Cross-Platform:** TypeScript runs on any platform that JavaScript runs on. The TypeScript compiler can be installed on any Operating System such as Windows, macOS, and Linux.
- **Object-Oriented Language:** TypeScript provides powerful features such as Classes, Interfaces, and Modules. You can write pure object-oriented code for client-side as well as server-side development.
- **Static type-checking:** TypeScript uses static typing. This is done using type annotations. It helps type checking at compile time. Thus, you can find errors while typing the code without running your script each time. Additionally, using the type inference mechanism, if a variable is declared without a type, it will be inferred based on its value.


•

TypeScript Features(cont.)

- **Optional Static Typing:** TypeScript static typing is optional, if you prefer to use JavaScript's dynamic typing.
- **DOM Manipulation:** Like JavaScript, TypeScript can be used to manipulate the DOM.
- **ES 6 Features:** TypeScript includes most features of planned [ECMAScript](#) 2015 (ES 6, 7) such as class, interface, Arrow functions etc.

TypeScript Environment setup

- **Local Environment Setup**
- 1-Install any A Text Editor
- 2-Install Node.JS
- 3-Install TypeScript using

NPM ([Node.js package manager](#)) is used to install the [TypeScript package](#)  on your local machine or a project. Make sure you have [Node.js](#) install on your local machine. If you are using JavaScript frameworks for your application, then it is highly recommended to [install Node.js](#).

To install or update the latest version of TypeScript, open command prompt/terminal and type the following command:


```
npm install -g typescript
```

The above command will install TypeScript globally so that you can use it in any project. Check the installed version of TypeScript using the following command:

```
tsc -v
```

TypeScript Environment setup (cont.)

- **Local Environment Setup**
- 1-Install any A Text Editor
- 2-Install Node.JS
- 3-Install TypeScript using NPM

NPM ([Node.js package manager](#)) is used to install the [TypeScript package](#)  on your local machine or a project. Make sure you have [Node.js](#) install on your local machine. If you are using JavaScript frameworks for your application, then it is highly recommended to [install Node.js](#).

To install or update the latest version of TypeScript, open command prompt/terminal and type the following command:

```
npm install -g typescript
```

First TypeScript Program: Compile and Run

- Here, you will learn to write a simple program in TypeScript, compile it and use it in the web page.
- Create a new file in your code editor and name it **add.ts** and write the following code in it.

- ```
function addNumbers(a: number, b: number) {
 return a + b;
}

var sum: number = addNumbers(10, 15)

console.log('Sum of the two numbers is: ' +sum);
```

# First TypeScript Program: Compile and Run

- The above TypeScript code defines the `addNumbers()` function, call it, and log the result in the browser's console.
- Now, open the command prompt on Windows (or a terminal on your platform), navigate to the path where you saved `add.ts`, and compile the program using the following command:

```
n\Type Script\test> tsc add.ts
```

- The above command will compile the TypeScript file `add.ts` and create the Javascript file named **`add.js`** at the same location. The `add.js` file contains the following code.

# First TypeScript Program: Compile and Run

- 

```
function addNumbers(a, b) {
 return a + b;
}

var sum = addNumbers(10, 15);
console.log('Sum of the two numbers is: ' + sum);
```

-

# TypeScript - Type Annotations

- TypeScript is a typed language, where we can specify the type of the variables, function parameters and object properties.
- We can specify the type using `:Type` after the name of the variable, parameter or property.
- There can be a space after the colon.
- TypeScript includes all the primitive types of JavaScript- number, string and boolean.

```
var age: number = 32; // number variable
var firstName: string = "John"; // string variable
var isUpdated: boolean = true; // Boolean variable
```

# TypeScript - Type Annotations

- The following example demonstrates the type annotation of parameters.

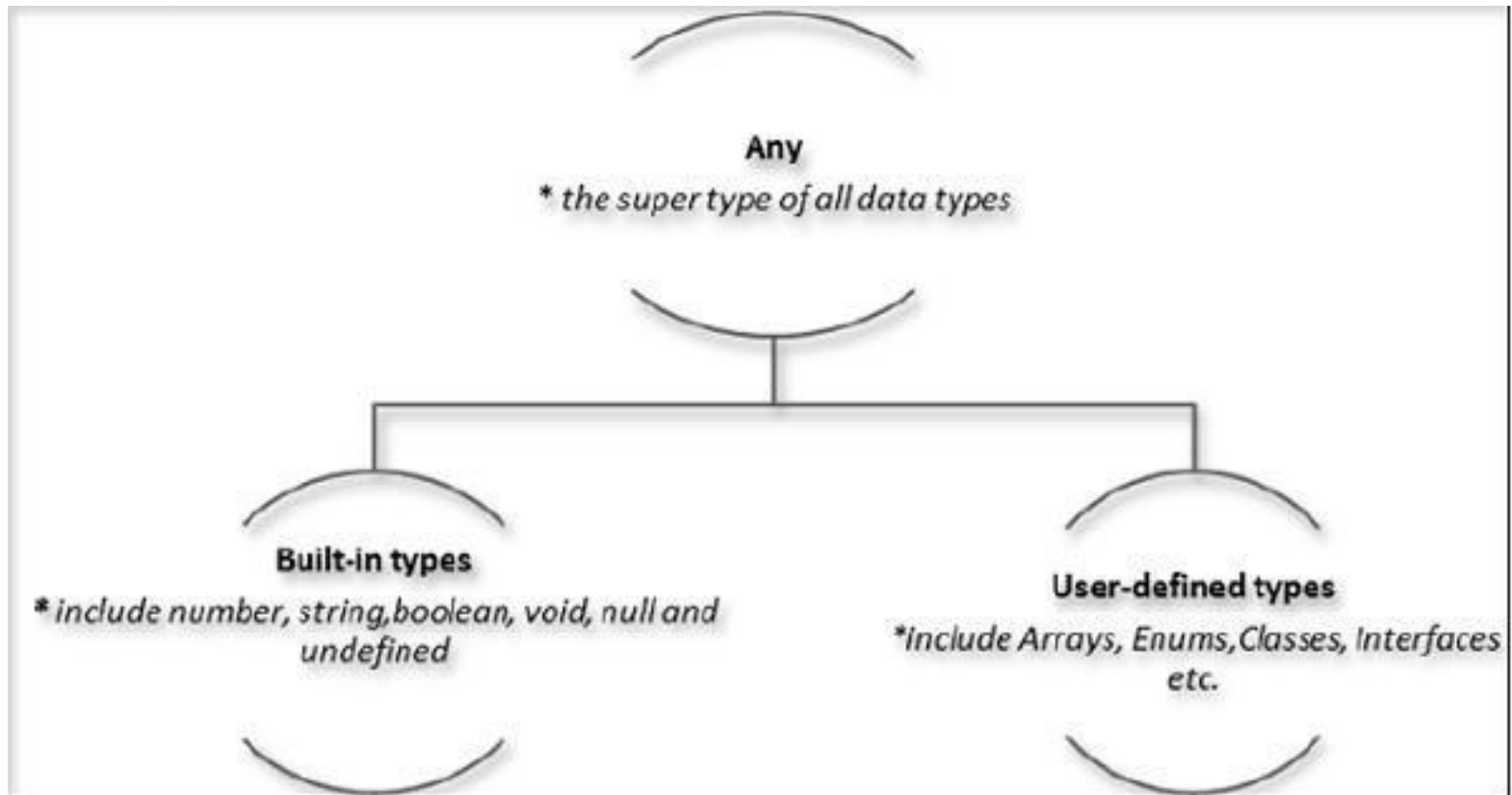
```
function displayEmployeeData(id:number, name:string)
{
 console.log("Id = " + id + ", Name = " + name);
}
```



# TypeScript Data Type

- The Type System represents the different types of values supported by the language.
- The Type System checks the validity of the supplied values, before they are stored or manipulated by the program.
- This ensures that the code behaves as expected.
- The Type System further allows for richer code hinting and automated documentation too.
- TypeScript provides data types as a part of its optional Type System.

# TypeScript Data Type



# TypeScript Data Type

## Variable Declaration in TypeScript

The type syntax for declaring a variable in TypeScript is to include a colon (:) after the variable name, followed by its type. Just as in JavaScript, we use the **var** keyword to declare a variable.

When you declare a variable, you have four options –

- Declare its type and value in one statement.

```
var [identifier] : [type-annotation] = value ;
```

- Declare its type but no value. In this case, the variable will be set to undefined.

```
var [identifier] : [type-annotation] ;
```

- Declare its value but no type. The variable type will be set to the data type of the assigned value.

```
var [identifier] = value ;
```

- Declare neither value nor type. In this case, the data type of the variable will be any and will be initialized to undefined.

```
var [identifier] ;
```

# TypeScript Data Type

- Number

```
let age:number=29;
```

- String

```
let firstName:string="Abanoub";
```

- boolean

```
let isActive=true;
```

# TypeScript Data Type

- Arrays

```
let fruits: Array<string>;
fruits = ['Apple', 'Orange', 'Banana'];

let ids: Array<number>;
ids = [23, 34, 100, 124, 44];
```

- Multi Type Array

```
let fruits: Array<string>;
let values1: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
// or
let values2: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];
```

# TypeScript Data Type

- **TypeScript - Tuples**
- TypeScript introduced a new data type called Tuple.
- **Tuple** can contain two values of different data types.
- Consider the following example of number, string and tuple type variables.

```
var empId: number = 1;
var empName: string = "Steve";

// Tuple type variable
var employee: [number, string] = [1, "Steve"];
```

# TypeScript Data Type

- TypeScript -  
Tuples

```
var employee: [number, string] = [1, "Steve"];
var person: [number, string, boolean] = [1, "Steve", true];

var user: [number, string, boolean, number, string]; // declare tuple variable
user = [1, "Steve", true, 20, "Admin"]; // initialize tuple variable
```

# TypeScript Data Type

- **TypeScript - Tuple Array**

```
var employee: [number, string][];
employee = [[1, "Steve"], [2, "Bill"], [3, "Jeff"]];
```

- **Accessing Tuple Elements**

- We can access tuple elements using index, the same way as an array. An index starts from zero.

```
var employee: [number, string] = [1, "Steve"];
employee[0]; // returns 1
employee[1]; // returns "Steve"
```



# TypeScript Data Type

## Enums

- Enums are one of the few features TypeScript has which is not a type- level extension of JavaScript.
- Enums allow a developer to define a set of named constants.
- In simple words, enums allow us to declare a set of named constants i.e. a collection of related values that can be numeric or string values.

### There are three types of enums:

- 1 Numeric enum
- 2 String enum
- 3 Heterogeneous enum

# TypeScript Data Type

## Numeric Enum

- Numeric enums are number-based enums i.e. *they store string values as numbers.*
- Enums can be defined using the keyword enum.
- Let's say we want to store a set of print media types. The corresponding enum in TypeScript would be:

```
enum PrintMedia {
 Newspaper,
 Newsletter,
 Magazine,
 Book
}
```

# TypeScript Data Type

## Numeric Enum

- In the above example, we have an enum named PrintMedia.
- The enum has four values: Newspaper, Newsletter, Magazine, and Book.
- Here, enum values start from zero and increment by 1 for each member. It would be represented as:

```
Newspaper = 0
Newsletter = 1
Magazine = 2
Book = 3
```

# TypeScript Data Type

## Numeric Enum

- Enums are always assigned numeric values when they are stored. The first value always takes the numeric value of 0, while the other values in the enum are incremented by 1.
- We also have the option to initialize the first numeric value ourselves.

For example, we can write the same enum as:

```
enum PrintMedia {
 Newspaper = 1,
 Newsletter,
 Magazine,
 Book
}
```

# TypeScript Data Type

## Numeric Enum

- The enum can be used as a function parameter or return type, as shown below:

```
enum PrintMedia {
 Newspaper = 1,
 Newsletter,
 Magazine,
 Book
}

function getMedia(mediaName: string): PrintMedia {
 if (mediaName === 'Forbes' || mediaName === 'Outlook') {
 return PrintMedia.Magazine;
 }
}

let mediaType: PrintMedia = getMedia('Forbes'); // returns Magazine
```

# TypeScript Data Type

## String Enum

- String enums are similar to numeric enums, except that the enum values are initialized with string values rather than numeric values.
- The benefits of using string enums is that string enums offer better readability. If we were to debug a program, it is easier to read string values rather than numeric values.
- Consider the same example of a numeric enum, but represented as a string enum.

# TypeScript Data Type

- String Enum

```
enum PrintMedia {
 Newspaper = "NEWSPAPER",
 Newsletter = "NEWSLETTER",
 Magazine = "MAGAZINE",
 Book = "BOOK"
}

// Access String Enum
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

# TypeScript Data Type

## Any

- TypeScript has type-checking and compile-time checks. However, we do not always have prior knowledge about the type of some variables, especially when there are user-entered values from third party libraries.
- In such cases, we need a provision that can deal with dynamic content. The Any type comes in handy here.

```
let something: any = "Hello World!";
something = 23;
something = true;

let arr: any[] = ["John", 212, true];
arr.push("Smith");
console.log(arr); //Output: ['John', 212, true, 'Smith']
```



# TypeScript Data Type

## Void

- Similar to languages like Java, void is used where there is no data. For example, if a function does not return any value then you can specify void as return type.

```
function sayHi(): void {
 console.log('Hi!')
}

let speech: void = sayHi();
console.log(speech); //Output: undefined
```

# Interfaces

- What is an interface? - *Simply interface is a contract.*

You were asked to build this:



But Built this:



# Interfaces

- Interface is a structure that defines the contract in your application.
- It defines the syntax for classes to follow.
- Classes that are derived from an interface must follow the structure provided by their interface.
- **The TypeScript compiler** does not convert interface to JavaScript.
- It uses interface for type checking. This is also known as "duck typing" or "structural subtyping".
- An interface is defined with the keyword `interface` and it can include
  - properties
  - and method declarations using a function or an arrow function.

# Interfaces

- First example

```
interface IEmployee {
 empCode: number;
 empName: string;
 getSalary: (number) => number; // arrow function
 getManagerName(number): string;
}
```

# Interfaces

- Second example

```
interface KeyPair {
 key: number;
 value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK

let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val' doesn't exist in type 'KeyPair'

let kv3: KeyPair = { key:1, value:100 }; // Compiler Error:
```

# Interfaces

- Optional Property

```
interface IEmployee {
 empCode: number;
 empName: string;
 empDept?: string;
}

let empObj1: IEmployee = { // OK
 empCode: 1,
 empName: "Steve"
}

let empObj2: IEmployee = { // OK
 empCode: 1,
 empName: "Bill",
 empDept: "IT"
}
```

# Interfaces

- Read only Properties

```
interface Citizen {
 name: string;
 readonly SSN: number;
}

let personObj: Citizen = { SSN: 110555444, name: 'James Bond' }

personObj.name = 'Steve Smith'; // OK
personObj.SSN = '333666888'; // Compiler Error
```

# Interfaces

- Extending Interfaces

```
interface IPerson {
 name: string;
 gender: string;
}

interface IEmployee extends IPerson {
 empCode: number;
}

let empObj: IEmployee = {
 empCode: 1,
 name: "Bill",
 gender: "Male"
}
```



# Classes

In object-oriented programming languages like Java and C#, classes are the fundamental entities used to create reusable components. Functionalities are passed down to classes and objects are created from classes. However, until ECMAScript 6 (also known as ECMAScript 2015), this was not the case with JavaScript. JavaScript has been primarily a functional programming language where inheritance is prototype-based. Functions are used to build reusable components. In ECMAScript 6, object-oriented class based approach was introduced. TypeScript introduced classes to avail the benefit of object-oriented techniques like encapsulation and abstraction. The class in TypeScript is compiled to plain JavaScript functions by the TypeScript compiler to work across platforms and browsers.

# Classes (cont.)

- A class can include the following:
  - Constructor
  - Properties
  - Methods
- The following is an example of a class in TypeScript

```
class Employee {
 empCode: number;
 empName: string;

 constructor(code: number, name: string) {
 this.empName = name;
 this.empCode = code;
 }

 getSalary(): number {
 return 10000;
 }
}
```

# Classes (cont.)

- Creating an Object of Class

```
class Employee {
 empCode: number;
 empName: string;
}

let emp = new Employee();
let emp2:Employee= new Employee();|
```

# Classes (cont.)

- Inheritance

```
class Person {
 name: string;

 constructor(name: string) {
 this.name = name;
 }
}

class Employee extends Person {
 empCode: number;

 constructor(empcode: number, name:string) {
 super(name);
 this.empCode = empcode;
 }

 displayName():void {
 console.log("Name = " + this.name + ", Employee Code = " + this.empCode);
 }
}

let emp = new Employee(100, "Bill");
emp.displayName(); // Name = Bill, Employee Code = 100
```

# Classes (cont.)

- **Class Implements Interface**
- A class can implement single or multiple interfaces.

```
interface IPerson {
 name: string;
 display():void;
}

interface IEmployee {
 empCode: number;
}

class Employee implements IPerson, IEmployee {
 empCode: number;
 name: string;

 constructor(empcode: number, name:string) {
 this.empCode = empcode;
 this.name = name;
 }

 display(): void {
 console.log("Name = " + this.name + ", Employee Code = " + this.empCode);
 }
}
```

# Classes (cont.)

- **Class Implements Interface**
- A class can implement single or multiple interfaces.

```
let per:IPerson = new Employee(100, "Bill");
per.display(); // Name = Bill, Employee Code = 100

let emp:IEmployee = new Employee(100, "Bill");
emp.display(); //Compiler Error: Property 'display' does not exist on type 'IEmployee'
```

# Classes (cont.)

- Method Overriding

```
class Car {
 name: string;

 constructor(name: string) {
 this.name = name;
 }

 run(speed: number = 0) {
 console.log("A " + this.name + " is moving at " + speed + " mph!");
 }
}

class Mercedes extends Car {

 constructor(name: string) {
 super(name);
 }

 run(speed = 150) {
 console.log('A Mercedes started')
 super.run(speed);
 }
}
```

# Classes (cont.)

- Method Overriding

```
class Honda extends Car {
 constructor(name: string) {
 super(name);
 }

 run(speed = 100) {
 console.log('A Honda started')
 super.run(speed);
 }
}

let mercObj = new Mercedes("Mercedes-Benz GLA");
let hondaObj = new Honda("Honda City")

mercObj.run(); // A Mercedes started A Mercedes-Benz GLA is moving at 150 mph!
hondaObj.run(); // A Honda started A Honda City is moving at 100 mph!
```



# TypeScript - Abstract Class

- Define an abstract class in Typescript using the abstract keyword.
- Abstract classes are mainly for inheritance where other classes may derive from them. We cannot create an instance of an abstract class.
- An abstract class typically includes one or more abstract methods or property declarations.
- The class which extends the abstract class **must** define all the abstract methods.

# TypeScript - Abstract Class

The following abstract class declares one abstract method `find` and also includes a normal method `display`.

```
abstract class Person {
 name: string;

 constructor(name: string) {
 this.name = name;
 }

 display(): void {
 console.log(this.name);
 }

 abstract find(string): Person;
}
```

# TypeScript - Abstract Class

```
class Employee extends Person {
 empCode: number;

 constructor(name: string, code: number) {
 super(name); // must call super()
 this.empCode = code;
 }

 find(name:string): Person {
 // execute AJAX request to find an employee from a db
 return new Employee(name, 1);
 }
}

let emp: Person = new Employee("James", 100);
emp.display(); //James

let emp2: Person = emp.find('Steve');
```

# TypeScript - Data Modifiers

- In object-oriented programming, the concept of '**Encapsulation**' is used to make class members public or private i.e. a class can control the visibility of its data members.
- This is done using access modifiers.
- There are three types of access modifiers in TypeScript: **public, private and protected**

# TypeScript - Data Modifiers

- **public**
- By default, all members of a class in TypeScript are public. All the public members can be accessed anywhere without any restrictions.

```
class Employee {
 public empCode: string;
 empName: string;
}

let emp = new Employee();
emp.empCode = "123";
emp.empName = "Ahmed";
```

# TypeScript - Data Modifiers

- **private**
- The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

```
class Employee {
 private empCode: string;
 empName: string;
}
```

```
let emp = new Employee();
emp.empCode = "123";
emp.empName = "Ahmed";
```

# TypeScript - Data Modifiers

- **protected**
- The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.

```
class Employee {
 public empName: string;
 protected empCode: number;

 constructor(name: string, code: number){
 this.empName = name;
 this.empCode = code;
 }
}

class SalesEmployee extends Employee{
 private department: string;

 constructor(name: string, code: number, department: string) {
 super(name, code);
 this.department = department;
 }
}

let emp = new SalesEmployee("John Smith", 123, "Sales");
empObj.empCode; //Compiler Error
```

# TypeScript - ReadOnly

- TypeScript includes the *readonly* keyword that makes a property as read-only in the class, type or interface.
- Prefix `readonly` is used to make a property as read-only.
- Read-only members can be accessed outside the class, but their value cannot be changed. Since read-only members cannot be changed outside the class, they either need to be initialized at declaration or initialized inside the class constructor.



# TypeScript - ReadOnly

```
class Employee {
 readonly empCode: number;
 empName: string;

 constructor(code: number, name: string) {
 this.empCode = code;
 this.empName = name;
 }
}

let emp = new Employee(10, "John");
emp.empCode = 20; //Compiler Error
emp.empName = 'Bill';
```

# TypeScript - Static

- ES6 includes static members and so does TypeScript. The static members of a class are accessed using the class name and dot notation, without creating an object  
e.g. `<ClassName>.<StaticMember>`.
- The static members can be defined by using the keyword *static*. Consider the following example of a class with static property.

```
class Circle {
 static pi: number = 3.14;

 static calculateArea(radius: number) {
 return this.pi * radius * radius;
 }
}

Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5
```

# TypeScript - Static

```
class Circle {
 static pi = 3.14;

 static calculateArea(radius:number) {
 return this.pi * radius * radius;
 }

 calculateCircumference(radius:number):number {
 return 2 * Circle.pi * radius;
 }
}

Circle.calculateArea(5); // returns 78.5

let circleObj = new Circle();
circleObj.calculateCircumference(5) // returns 31.4000000
//circleObj.calculateArea(); <-- cannot call this
```

# TypeScript - Generics

- In this section, we will learn about **generics** in TypeScript.
- When writing programs, one of the most important aspects is to build **reusable components**.
- This ensures that the program is flexible as well as scalable in the long-term.
- **Generics** offer a way to create reusable components. Generics provide a way to make components work with any data type and not restrict to one data type.
- So, components can be called or used with a variety of data types.

# TypeScript - Generics

- Let's see why we need Generics using the following example.

```
function getArray(items : any[]) : any[] {
 return new Array().concat(items);
}

let myNumArr = getArray([100, 200, 300]);
let myStrArr = getArray(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // OK
myStrArr.push(500); // OK

console.log(myNumArr); // [100, 200, 300, 400, "Hi"]
console.log(myStrArr); // ["Hello", "World", "Hello TypeScript", 500]
```

# TypeScript - Generics

- In the above example, the **getArray()** function accepts an array of type any.
- It creates a new array of type any, concatenates items to it and returns the new array.
- **Since we have used type any** for our arguments, we can pass any type of array to the function.
- However, this may not be the desired behavior.
- We may **want to add the numbers to number array** or the **strings to the string array** but not numbers to the string array or vice-versa.

# TypeScript - Generics

- The above function can be rewritten as a generic function as below.

```
function getArray<T>(items : T[]) : T[] {
 return new Array<T>().concat(items);
}

let myNumArr = getArray<number>([100, 200, 300]);
let myStrArr = getArray<string>(["Hello", "World"]);

myNumArr.push(400); // OK
myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // Compiler Error
myStrArr.push(500); // Compiler Error
```

# TypeScript - Generics

- **Multiple Type Variables**
- We can specify multiple type variables with different names as shown below.

```
function displayType<T, U>(id:T, name:U): void {
 console.log(typeof(id) + ", " + typeof(name));
}

displayType<number, string>(1, "Steve"); // number, string
```



# TypeScript - Generics

- **TypeScript - Generic Interface**

- Here, you will learn about the generic interface in TypeScript.
- The generic type can also be used with the interface. The following is a generic interface.

```
interface KeyPair<T, U> {
 key: T;
 value: U;
}

let kv1: KeyPair<number, string> = { key:1, value:"Steve" }; // OK
let kv2: KeyPair<number, number> = { key:1, value:12345 }; // OK
```

# TypeScript - Generics

- TypeScript - Generic Interface

```
interface IKeyValueProcessor<T, U>
{
 process(key: T, val: U): void;
};

class kvProcessor implements IKeyValueProcessor<number, string>
{
 process(key:number, val:string):void {
 console.log(`Key = ${key}, val = ${val}`);
 }
}

let proc: IKeyValueProcessor<number, string> = new kvProcessor();
proc.process(1, 'Bill'); //Output: processKeyPairs: key = 1, value = Bill
```

# TypeScript - Generics

- **TypeScript - Generic Classes**
- TypeScript supports generic classes. The generic type parameter is specified in angular brackets after the name of the class. A generic class can have generic fields (member variables) or methods.

# TypeScript - Generics

```
class KeyValuePair<T,U>
{
 private key: T;
 private val: U;

 setKeyValue(key: T, val: U): void {
 this.key = key;
 this.val = val;
 }

 display():void {
 console.log(`Key = ${this.key}, val = ${this.val}`);
 }
}

let kvp1 = new KeyValuePair<number, string>();
kvp1.setKeyValue(1, "Steve");
kvp1.display(); //Output: Key = 1, Val = Steve

let kvp2 = new KeyValuePair<string, string>();
kvp2.setKeyValue("CEO", "Bill");
kvp2.display(); //Output: Key = CEO, Val = Bill
```