TypeScript

TypeScript

Elite
IT Academy

Why typescript?

- Typescript is open source.
- Typescript simplifies JavaScript code, making it easier to read and debug.
- Typescript is a superset of ES3, ES5, and ES6.
- Typescript will save developers time.
- Typescript code can be compiled as per ES5 and ES6 standards to support the latest browser.
- Typescript can help us to avoid painful bugs that developers commonly run into when writing JavaScript by type checking the code.
- Typescript is nothing but JavaScript with some additional features.

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, mac os 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.
- Additionally, using the type inference mechanism, if a variable is declared without a type, it will be inferred based on its value.

Typescript features

- Optional static typing: typescript also allows optional static typing if you would rather use JavaScript's dynamic typing.
- DOM manipulation: just like JavaScript, typescript can be used to manipulate the DOM for adding or removing elements.
- ES 6 features: typescript includes most features of planned ECMAScript 2015 (ES 6, 7) such as class, interface, arrow functions etc.

Setup Development Environment

- Install typescript using node.Js package manager (npm).
- Install the typescript plug-in in your IDE (integrated development environment).

```
npm install -g typescript
tsc -v
Version blah.blah.blah
```

Typescript Playground

- Typescript provides an online playground https://www.Typescriptlang.Org/play to write and test your code on the fly without the need to download or install anything.

Typescript Data Types

- Number - Store numbers in different number systems
- String - Store string values
- Boolean - true or false, Boolean is an object, boolean is a primitive
- Array - collection of data
- Tuple - Stores two sets of values of different types
- Enum - enumerations
- Union - more than one datatype for a variable
- Any - dynamic variables
- Void - no data type
- Never - value will never occur

Variable Declaration

- Variables can be declared using :
  - var, let, const

Examples

Elite IT Academy

Example: TypeScript Number Type Variables

```
let first:number = 123; // number
let second: number = 0x37CF;  // hexadecimal
let third:number=0o377 ;      // octal
let fourth: number = 0b111001;// binary

console.log(first);  // 123
console.log(second); // 14287
console.log(third);  // 255
console.log(fourth); // 57
```

Examples

Example: TypeScript String Type Variable

```
let employeeName:string = 'John Smith';
//OR
let employeeName:string = "John Smith";
```

```
let isPresent:boolean = true;
```

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

Examples

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

Example: Array Declaration and Initialization

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

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

Example: Multi Type Array

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

Examples - Tuple

Example: Tuple vs Other Data Types

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

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

Example: Tuple

```
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
```

Examples - Tuple

Example: Tuple Array

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

Example: push()

```
var employee: [number, string] = [1, "Steve"];
employee.push(2, "Bill");
console.log(employee); //Output: [1, 'Steve', 2, 'Bill']
```

Examples - Enums

Example: Numeric Enum

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

Example: Enum as Return Type

```
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
```

Examples - Enums

- Numeric enums can include members with computed numeric value. The value of an enum member can be either a constant or computed. The following enum includes members with computed values.
- When the enum includes computed and constant members, then uninitiated enum members either must come first or must come after other initialized members with numeric constants.

Example: Computed Enum

```
enum PrintMedia {
    Newspaper = 1,
    Newsletter = getPrintMediaCode('newsletter'),
    Magazine = Newsletter * 3,
    Book = 10
}

function getPrintMediaCode(mediaName: string): number {
    if (mediaName === 'newsletter') {
        return 5;
    }
}

PrintMedia.Newsetter; // returns 5
PrintMedia.Magazine; // returns 15
```

Examples - String Enums

- String enums are similar to numeric enums, except that the enum values are initialized with string values rather than numeric values.
- The difference between numeric and string enums is that numeric enum values are auto-incremented, while string enum values need to be individually initialized.

Example: String Enum

```
enum PrintMedia {
    Newspaper = "NEWSPAPER",
    Newsletter = "NEWSLETTER",
    Magazine = "MAGAZINE",
    Book = "BOOK"
}
// Access String Enum
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

Union

- Typescript allows us to use more than one data type for a variable or a function parameter. This is called union type.

Example: Union

```
let code: (string | number);
code = 123;   // OK
code = "ABC"; // OK
code = false; // Compiler Error

let empId: string | number;
empId = 111; // OK
empId = "E111"; // OK
empId = true; // Compiler Error
```

Union

Elite IT Academy

Example: Function Parameter as Union Type

```
function displayType(code: (string | number))
{
    if(typeof(code) === "number")
        console.log('Code is number.')
    else if(typeof(code) === "string")
        console.log('Code is string.')
}

displayType(123); // Output: Code is number.
displayType("ABC"); // Output: Code is string.
displayType(true); //Compiler Error: Argument of type 'true'
```

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.

Example: Any

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

Example: Any type Array

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

void

- Similar to languages like java, void is used where there is no data type. For example, in return type of functions that do not return any value.
- The void type can have undefined or null as a value where as never cannot have any value.

Example: void

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

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

Never

- Typescript introduced a new type never, which indicates the values that will never occur.
- The never type is used when you are sure that something is never going to occur. For example, you write a function which will not return to its end point or always throws an exception.

Example: never

```
function throwError(errorMsg: string): never {
            throw new Error(errorMsg);
}

function keepProcessing(): never {
            while (true) {
          console.log('I always does something and never ends.')
      }
}
```

Type inference in Typescript

- It is not mandatory to annotate type. TypeScript infers types of variables when there is no explicit information available in the form of type annotations.
- **Types are inferred by TypeScript compiler when:**
  - Variables are initialized, Default values are set for parameters
  - Function return types are determined

```
var arr = [0, 1, "test"];
```

- Here, the array has values of type number as well as type string. In such cases, the typescript compiler looks for the most common type to infer the type of the object but does not find any super type that can encompass all the types present in the array.
- In such cases, the compiler treats the type as a union of all types present in the array. Here, the type would be (string | number) which means that the array can hold either string values or number values. This is called union type.

Type assertion in Typescript

- Type assertion allows you to set the type of a value and tell the compiler not to infer it.
- This is when you, as a programmer, might have a better understanding of the type of a variable than what Typescript can infer on its own.
- Such a situation can occur when you might be porting over code from JavaScript and you may know a more accurate type of the variable than what is currently assigned.
- It is similar to type casting in other languages like C# and Java.
- It is merely a way to let the Typescript compiler know the type of a variable.

```
let code: any = 123;
let employeeCode = <number> code;
```

```
let code: any = 123;
let employeeCode = code as number;
```

Functions in Typescript

- Functions can also include parameter types and return type.

Example: Function with Paramter and Return Types

```
let Sum = function(x: number, y: number) : number
{
    return x + y;
}

Sum(2,3); // returns 5
```

Functions in Typescript

Example: Function Parameters

```
function Greet(greeting: string, name: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // Compiler Error: Expected 2 arguments, but got 1.
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

- Parameters are values or arguments passed to a function.
- In Typescript, the compiler expects a function to receive the exact number and type of arguments as defined in the function signature.
- If the function expects three parameters, the compiler checks that the user has passed values for all three parameters i.e. it checks for exact matches.

Optional Parameters

- All optional parameters must follow required parameters and should be at the end.
  - In the above example, the second parameter name is marked as optional with a question mark appended at the end.
  - Hence, the function greet() accepts either 1 or 2 parameters and returns a greeting string. If we do not specify the second parameter then its value will be undefined.

Example: Optional Parameter

```
function Greet(greeting: string, name?: string ) : string {
    return greeting + ' ' + name + '!';
}

Greet('Hello','Steve');//OK, returns "Hello Steve!"
Greet('Hi'); // OK, returns "Hi undefined!".
Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.
```

Function Overloading

- Typescript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type.
- However, the number of parameters should be the same.
- Function overloading with different number of parameters and types with same name is not supported

Example: Function Overloading

```
function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {
    return a + b;
}

add("Hello ", "Steve"); // returns "Hello Steve"
add(10, 20); // returns 30
```

Rest Parameters

- Typescript introduced rest parameters to accommodate n number of parameters easily.
- When the number of parameters that a function will receive is not known or can vary, we can use rest parameters.
- In JavaScript, this is achieved with the "arguments" variable. However, with typescript, we can use the rest parameter denoted by ellipsis ....
- Remember, rest parameters must come last in the function definition, otherwise the Typescript compiler will show an error.

Example: Rest Parameters

```
function Greet(greeting: string, ...names: string[]) {
    return greeting + " " + names.join(", ") + "!";
}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"
```

Type Alias

- type aliases create type definitions that can be reused throughout the code.
- This is unlike type unions and intersections, where the explicit type information is used repetitively.

```
type myType = {
  memberOne: string;
  memberTwo: number;
}
let favoriteNum: myType = {"my favorite number is ", 42}

type myType = string | number;

let favoriteNum: myType = '42';
```

Type Alias

- type aliases create type definitions that can be reused throughout the code.
- This is unlike type unions and intersections, where the explicit type information is used repetitively.

```
type myType = {
  memberOne: string;
  memberTwo: number;
}
let favoriteNum: myType = {memberOne:"my favorite number is ", memberTwo: 42}

type myType = string | number;

let favoriteNum: myType = '42';
```

Typescript Interface

- 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.

Example: Interface

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

Interface as type

- Interface in typescript can be used to define a type and also to implement it in the class.
- In the example, an interface key pair includes two properties key and value.
- A variable kv1 is declared as key pair type. So, it must follow the same structure as key pair. It means only an object with properties key of number type and value of string type can be assigned to a variable kv1.

Example: Interface as Type

```
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:
```

Interface as function

- In the example, an interface KeyValueProcessor includes a method signature. This defines the function type.
- Now, we can define a variable of type KeyValueProcessor which can only point to functions with the same signature as defined in the KeyValueProcessor interface.
- So, addKeyValue or updateKeyValue function is assigned to kvp. So, kvp can be called like a function.

Example: Function Type

```
interface KeyValueProcessor
{
    (key: number, value: string): void;
};

function addKeyValue(key:number, value:string):void {
    console.log('addKeyValue: key = ' + key + ', value = ' + value)
}

function updateKeyValue(key: number, value:string):void {
    console.log('updateKeyValue: key = '+ key + ', value = ' + value)
}

let kvp: KeyValueProcessor = addKeyValue;
kvp(1, 'Bill'); //Output: addKeyValue: key = 1, value = Bill

kvp = updateKeyValue;
kvp(2, 'Steve'); //Output: updateKeyValue: key = 2, value = Steve
```

Interface as an array type

- An interface can also define the type of an array where you can define the type of index as well as values
- interface numlist defines a type of array with index as number and value as number type. In the same way, istringlist defines a string array with index as string and value as string.

Example: Type of Array

```
interface NumList {
    [index:number]:number
}

let numArr: NumList = [1, 2, 3];
numArr[0];
numArr[1];

interface IStringList {
    [index:string]:string
}

let strArr : IStringList;
strArr["TS"] = "TypeScript";
strArr["JS"] = "JavaScript";
```

Interface optional properties

- Sometimes, we may declare an interface with excess properties but may not expect all objects to define all the given interface properties.
- We can have optional properties, marked with a "?". in such cases, objects of the interface may or may not define these properties.
- empDept is marked with ?, so objects of IEmployee may or may not include this property.

Example: 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"
}
```

Interface readonly properties

- Typescript provides a way to mark a property as read only.
- This means that once a property is assigned a value, it cannot be changed!

Example: Readonly Property

```
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
```

Extending interfaces

- Interfaces can extend one or more interfaces. This makes writing interfaces flexible and reusable.

Example: Extend Interface

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

interface IEmployee extends IPerson {
    empCode: number;
}

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

Implementing interfaces

Example: Interface Implementation

```
interface IEmployee {
    empCode: number;
    name: string;
    getSalary:(number)=>number;
}

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

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

    getSalary(empCode:number):number {
        return 20000;
    }
}

let emp = new Employee(1, "Steve");
```

Class in typescript

- 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.

Intersection Type in TypeScript

- In Typescript, Although intersection and union types are similar, they are employed in completely different ways.
- An intersection type is a type that merges several kinds into one.
- This allows you to combine many types to create a single type with all of the properties that you require.
- An object of this type will have members from all of the types given.
- The ‘&’ operator is used to create the intersection type.

```
interface Student {
   student_id: number;
   name: string;
}

interface Teacher {
   Teacher_Id: number;
   teacher_name: string;
}

type intersected_type = Student & Teacher;

let obj1: intersected_type = {
    student_id: 3232,
    name: "rita",
    Teacher_Id: 7873,
    teacher_name: "seema",
};

console.log(obj1.Teacher_Id);
console.log(obj1.name);
```

Class in typescript

Elite IT Academy

Example: Class

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

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

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

- It is not necessary for a class to have a constructor.
- An object of the class can be created using the new keyword.

Example: Create an Object

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

let emp = new Employee();
```

Inheritance in typescript

- typescript classes can be extended to create new classes with inheritance, using the keyword extends.
- We must call super() method first before assigning values to properties in the constructor of the derived class.
- A class can implement single or multiple interfaces.

```
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
```

Abstract Class

- Typescript allows us to define an abstract class using keyword abstract.
- 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.

```
abstract class Person {
 name: string;
 constructor(name: string) {
  this.name = name;
 }

 display(): void {
  console.log(this.name);
 }
 abstract find(string): Person;
}
```

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');
```

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: private, public and protected
- By default, all members of a class in Typescript are public.
- All the public members can be accessed anywhere without any restrictions.

Example: public

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

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

Data Modifiers

- The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.
- In addition to the access modifiers, typescript provides two more keywords: read-only and static.

```
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
```

Readonly members

- Typescript introduced the keyword read-only, which makes a property as read-only in the class, type or interface.
- Prefix read-only 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.

Example: ReadOnly Class Properties

```
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'; //Compiler Error
```

Readonly types

Example: ReadOnly Type

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

let emp1: Readonly<IEmployee> = {
    empCode:1,
    empName:"Steve"
}
```

```
emp1.empCode = 100; // Compiler Error: Cannot change readonly 'empCode'
emp1.empName = 'Bill'; // Compiler Error: Cannot change readonly 'empName'

let emp2: IEmployee = {
    empCode:1,
    empName:"Steve"
}

emp2.empCode = 100; // OK
emp2.empName = 'Bill'; // OK
```

Static members

- 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.
- The circle class includes a static property pi. This can be accessed using circle.Pi

Example: Static Property

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

Example: Static Members

```
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 Module

- The typescript code we write is in the global scope by default.
- If we have multiple files in a project, the variables, functions, etc. Written in one file are accessible in all the other files.
- A module can be defined in a separate .ts file which can contain functions, variables, interfaces and classes.
- Use the prefix export with all the definitions you want to include in a module and want to access from other modules.

Typescript Module

- employee.Ts is a module which contains two variables and a class definition.
- The age variable and the employee class are prefixed with the export keyword, whereas companyname variable is not.
- Thus, employee.Ts is a module which exports the age variable and the employee class to be used in other modules by importing the employee module using the import keyword.
- The companyname variable cannot be accessed outside this employee module, as it is not exported.

Typescript Module

Elite IT Academy

Employee.ts

```
export let age : number = 20;
export class Employee {
    empCode: number;
    empName: string;
    constructor(name: string, code: number) {
        this.empName = name;
        this.empCode = code;
    }
    displayEmployee() {
        console.log ("Employee Code: " + this.empCode + ", Employee Name: " + this.empName );
    }
}
let companyName:string = "XYZ";
```

Importing a Module

- A module can be used in another module using an import statement.

EmployeeProcessor.ts

```
import { Employee } from "./Employee";
let empObj = new Employee("Steve Jobs", 1);
empObj.displayEmployee(); //Output: Employee Code: 1, Employee Name: Steve Jobs
```

EmployeeProcessor.ts

```
import * as Emp from "./Employee"
console.log(Emp.age); // 20

let empObj = new Emp.Employee("Bill Gates" , 2);
empObj.displayEmployee(); //Output: Employee Code: 2, Employee Name: Bill Gates
```

Renaming a Module

- A module can be renamed in another module using an import statement.

EmployeeProcessor.ts

```
import { Employee as Associate } from "./Employee"
let obj = new Associate("James Bond" , 3);
obj.displayEmployee();//Output: Employee Code: 3, Employee Name: James Bond
```

Typescript Generics

- Generics uses the type variable <T>, a special kind of variable that denotes types. The type variable remembers the type that the user provides and works with that particular type only.
- This is called preserving the type information.

Example: Generic Function

```
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
```

Multiple Type Variables

- We can specify multiple type variables with different names as shown below.

Example: Multiple Type Variables

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

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

- Generic type can also be used with other non-generic types.

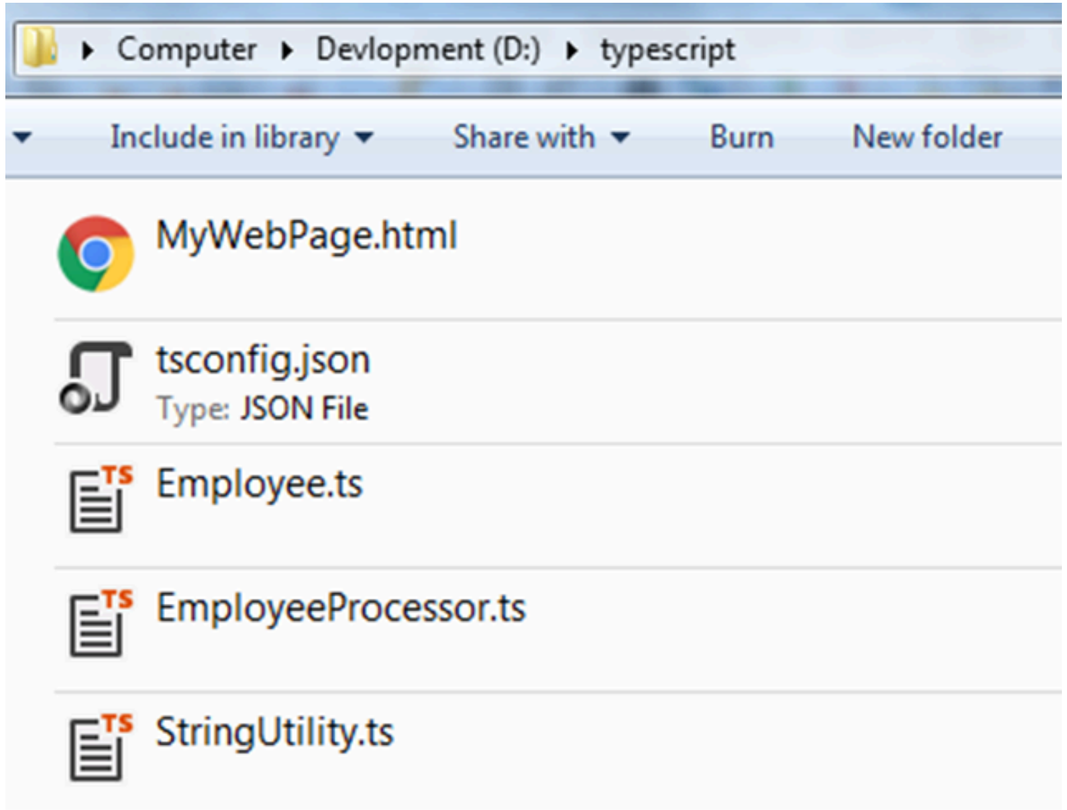
Example: Generic with Non-generic Type

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

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

Compiling a Typescript project

- typescript files can be compiled using the tsc <file name>.ts command.
- It will be tedious to compile multiple .ts files in a large project. So, typescript provides another option to compile all or certain .Ts files of the project.
- Typescript supports compiling a whole project at once by including the tsconfig.Json file in the root directory.
- The tsconfig.Json file is a simple file in JSON format where we can specify various options to tell the compiler how to compile the current project.
- Consider the following simple project which includes two module files, one namespace file, tsconfig.Json and an html file.

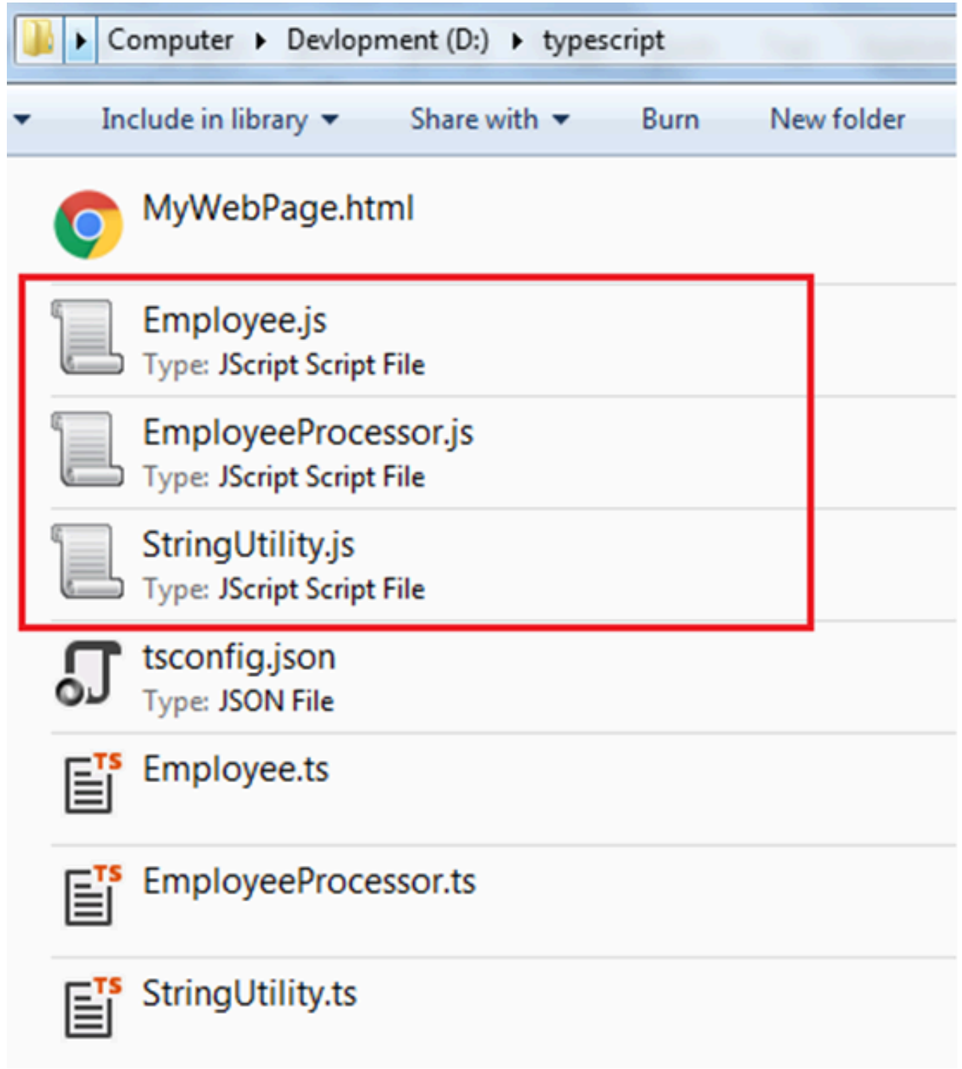


Compiling a Typescript project

- The above tsconfig.Json file includes empty curly brackets { } and does not include any options.
- In this case, the tsc command will consider the default values for the compiler options and compile all the .Ts files in a root directory and its sub-directories.

```
D:\typescript>tsc
```

- When using the tsc command to compile files, if a path to tsconfig.json is not specified, the compiler will look for the file in the current directory.
- If not found in the current directory, it will search for the tsconfig.json file in the parent directory.
- The compiler will not compile a project if a tsconfig file is absent.



Sample tsconfig.json

Elite IT Academy

Example: files in tsconfig.json

```
{
    "compilerOptions": {
        "module": "amd",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "sourceMap": true
    },
    "files": {
        "Employee.ts"
    }
}
```

Example: tsconfig.json

```
{
    "compilerOptions": {
        "module": "amd",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "outFile": "../../built/local/tsc.js",
        "sourceMap": true
    },
    "include": [
        "src/**/*"
    ],
    "exclude": [
        "node_modules",
        "**/*.spec.ts"
    ]
}
```