TYPESCRIPT

A brief history of typescript

in 2010, anders hejlsberg (the creator of typescript) started working on typescript at Microsoft and in 2012 the first version of typescript was released to the public (typescript 0.8). Although the release of typescript was praised by many people around the world, due to the lack of support by major ides, it was not majorly adopted by the JavaScript community.

The first version of typescript (typescript 0.8) released to the public October 2012.

The latest version of typescript (typescript 3.0) was released to the public in July 2018.

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. 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.

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 Type

- Number
- String
- Boolean
- Array
- Tuple
- Enum
- Union
- Any
- Void
- Never

Variable Declaration

Variables can be declared using :

var, let, const

TypeScript Data Type - NUMBER

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
```

TypeScript Data Type - STRING

Example: TypeScript String Type Variable

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

TypeScript Data Type - BOOLEAN

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

Note that, boolean with an upper case B is different from boolean with a lower case b. Upper case boolean is an object type whereas lower case boolean is a primitive type. It is always recommended to use boolean, the primitive type in your programs. This is because, while JavaScript coerces an object to its primitive type, the typescript type system does not. Typescript treats it like an object type.

TypeScript Data Type - Array

1. Using square brackets. This method is similar to how you would declare arrays in JavaScript.

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

2. Using a generic array type, array<element type>.

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

Of course, you can always initialize an array like shown below, but you will not get the advantage of typescript's type system.

```
let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false];
```

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'];
```

TYPESCRIPT DATA TYPE - TUPLE

Typescript introduced a new data type called tuple. There are other data types such as number, string, boolean etc. In typescript which only store a value of that particular data type. Tuple is a new data type which includes two set of values of different data types.

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
```

You can declare an array of tuple also.

Example: Tuple Array

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

Add Elements into Tuple

Example: push()

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

TYPESCRIPT DATA TYPE - ENUM

Enums or enumerations are a new data type supported in typescript. Most object-oriented languages like java and C# use enums. This is now available in typescript too.

There are three types of enums:

Numeric enum

String enum

Heterogeneous enum

Numeric Enum

Numeric enums are number-based enums i.e. they store string values as numbers.

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
```

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.

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
```

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.

STRING ENUM

String enums are similar to numeric enums, except that the enum values are initialized with string values rather than numeric values.

Example: String Enum

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

The difference between numeric and string enums is that numeric enum values are auto-incremented, while string enum values need to be individually initialized.

HETEROGENEOUS ENUM

Heterogeneous enums are enums that contain both string and numeric values.

Example: Heterogeneous Enum

```
enum Status {
    Active = 'ACTIVE',
    Deactivate = 1,
    Pending
}
```

```
"use strict";
var Status;
(function (Status) {
    Status["Active"] = "ACTIVE";
    Status[Status["Deactivate"] = 1] = "Deactivate";
    Status[Status["Pending"] = 2] = "Pending";
})(Status || (Status = {}));
```

TYPESCRIPT - 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
```

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' is not assignable to a parameter of type string | number
```

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.

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' ]
```

TYPESCRIPT DATA TYPE - 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.

Example: void

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

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

*** There is no meaning to assign void to a variable, as only null or undefined is assignable to void.**

TYPESCRIPT DATA TYPE - 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.')
     }
}
```

DIFFERENCE BETWEEN NEVER AND VOID

The void type can have undefined or null as a value where as never cannot have any value.

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.')
     }
}
```

In typescript, a function that does not return a value, actually returns undefined.

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

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

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

For example:

```
var a = "some text";
var b = 123;
a = b; // Compiler Error: Type 'number' is not assignable to type 'string'
```

The above code shows an error because while inferring types, typescript inferred the type of variable a as string and variable b as number. When we try to assign b to a, the compiler complains saying a number type cannot be assigned to a string type.

TYPE INFERENCE IN COMPLEX OBJECTS

For example:

```
var arr = [ 10, null, 30, 40 ];
```

In the above example, we have an array that has the values 10, null, 30, and, 40 . typescript looks for the most common type to infer the type of the object. In this case, it picks the one that's is compatible with all types i.e. Number, as well as null.

Consider another example:

```
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

There are two ways to do type assertion in typescript:

- Using the angular bracket <> syntax.

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

- Using as keyword

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

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. However, unlike C# and Java, there is no runtime effect of type assertion in Typescript. It is merely a way to let the Typescript compiler know the type of a variable.

TYPESCRIPT - FUNCTION

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
```

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.

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.
```

OPTIONAL PARAMETERS IN FUNCTIONS

All optional parameters must follow required parameters and should be at the end.

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.
```

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.

TYPESCRIPT - 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.

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
```

Function overloading with different number of parameters and types with same name is not supported.

TYPESCRIPT - 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 ....

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 !"
```

Remember, rest parameters must come last in the function definition, otherwise the Typescript compiler will show an error. The following is not valid.

Example: Wrong Rest Parameters

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

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.

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:
```

In the above 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. The typescript compiler will show an error if there is any change in the name of the properties or the data type is different than key pair. Another variable kv2 is also declared as key pair type but the assigned value is Val instead of value, so this will cause an error. In the same way, kv3 assigns a number to the value property, so the compiler will show an error. Thus, typescript uses an interface to ensure the proper structure of an object.

INTERFACE AS FUNCTION TYPE

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 FOR ARRAY TYPE

An interface can also define the type of an array where you can define the type of index as well as values.

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";
```

In the above example, 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.

OPTIONAL PROPERTY

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.

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"
}
```

In the above example, empDept is marked with ?, so objects of IEmployee may or may not include this property.

READ ONLY 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
```

In the above example, the SSN property is read only. We define the personobj object of type citizen and assign values to the two interface properties. Next, we try to change the values assigned to both the properties-name and SSN. The typescript compiler will show an error when we try to change the read only SSN property.

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"
}
```

In the above example, the iemployee interface extends the iperson interface. So, objects of iemployee must include all the properties and methods of the iperson interface otherwise, the compiler will show an error.

IMPLEMENTING AN INTERFACE

Similar to languages like java and C#, interfaces in typescript can be implemented with a class. The class implementing the interface needs to strictly conform to the structure of the interface.

In the example, the iemployee interface is implemented in the employee class using the implement keyword. The implementing class should strictly define the properties and the function with the same name and data type. If the implementing class does not follow the structure, then the compiler will show an error.

Of course, the implementing class can define extra properties and methods, but at least it must define all the members of an interface.

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");
```

TYPESCRIPT - CLASS

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.

A class can include the following:

- Constructor
- Properties
- Methods

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.

Example: Class without Constructor

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

CREATING AN OBJECT OF CLASS

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();
```

Here, we create an object called emp of type employee using let emp = new employee();. The above class does not include any parameterized constructor so we cannot pass values while creating an object. If the class includes a parameterized constructor, then we can pass the values while creating the object.

In the example, we pass values to the object to initialize the member variables. When we instantiate a new object, the class constructor is called with the values passed and the member variables empCode and empName are initialized with these values.

```
class Employee {

    empCode: number;
    empName: string;

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

let emp = new Employee(100,"Steve");
```

INHERITANCE

Just like object-oriented languages such as java and C#, typescript classes can be extended to create new classes with inheritance, using the keyword extends.

Example: 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
```

In the last example, the employee class extends the person class using extends keyword. This means that the employee class now includes all the members of the person class. The constructor of the employee class initializes its own members as well as the parent class's properties using a special keyword 'super'. The super keyword is used to call the parent constructor and passes the property values.

Note:

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.

Example: Implement Interface

```
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);
    }
}

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'
```

In the last example, the employee class implements two interfaces - iperson and iemployee. So, an instance of the employee class can be assigned to a variable of iperson or iemployee type. However, an object of type iemployee cannot call the display() method because iemployee does not include it. You can only use properties and methods specific to the object type.

INTERFACE EXTENDS CLASS

An interface can also extend a class to represent a type.

Example: Interface Extends Class

```
class Person {
    name: string;
}

interface IEmployee extends Person {
    empCode: number;
}

let emp: IEmployee = { empCode  : 1, name:"James Bond" }
```

In the above example, iemployee is an interface that extends the person class. So, we can declare a variable of type iemployee with two properties. So now, we must declare and initialize values at the same time.

TYPESCRIPT - 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.

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;
}

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');
```

In the last example, person is an abstract class which includes one property and two methods, one of which is declared as abstract. The find() method is an abstract method and so must be defined in the derived class. The employee class derives from the person class and so it must define the find() method as abstract. The employee class must implement all the abstract methods of the person class, otherwise the compiler will show an error.

Note:

The class which implements an abstract class must call super() in the constructor.

The abstract class can also include an abstract property, as shown below.

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

class Employee extends Person {
 name: string;
 empCode: number;
 constructor(name: string, code: number) {
  super(); // must call super()

  this.empCode = code;
  this.name = name;
 }
}

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

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**

Public

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";
```

In the last example, empcode and empname are declared as public. So, they can be accessible outside of the class using an object of the class.

Please notice that there is not any modifier applied before empname, as typescript treats properties and methods as public by default if no modifier is applied to them.

private

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

Example: private

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

let emp = new Employee();
emp.empCode = 123; // Compiler Error
emp.empName = "Swati";//OK
```

In the above example, we have marked the member empcode as private. Hence, when we create an object emp and try to access the emp.Empcode member, it will give an error.

protected

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

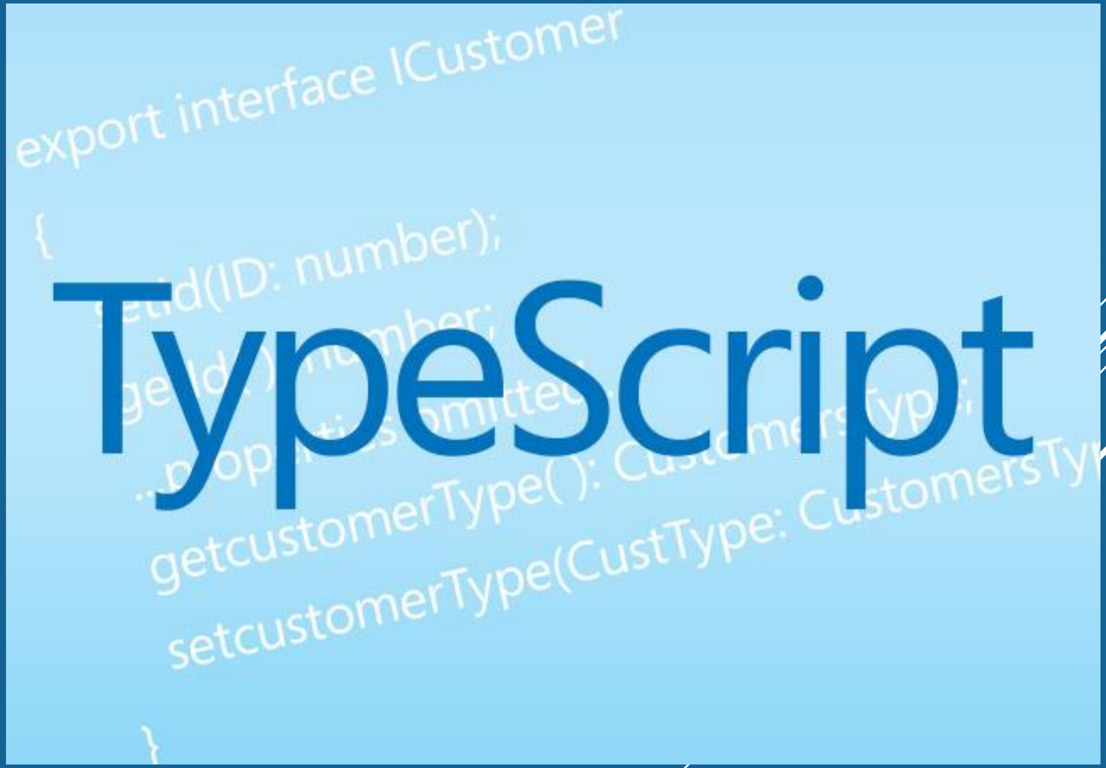
Example: protected

```
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
```

In the last example, we have a class employee with two members, public empname and protected property empcode. We create a subclass salesemployee that extends from the parent class employee. If we try to access the protected member from outside the class, as emp.Empcode, we get the following compilation error:

Error ts2445: property 'empcode' is protected and only accessible within class 'employee' and its subclasses.

In addition to the access modifiers, typescript provides two more keywords: **read-only** and **static**.



TYPESCRIPT - READONLY

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
```

In the last example, we have the employee class with two properties- empname and empcode. Since empcode is read only, it can be initialized at the time of declaration or in the constructor.

If we try to change the value of empcode after the object has been initialized, the compiler shows the following compilation error:

Error TS2540: cannot assign to empcode' because it is a constant or a read-only property.

An interface can also have read-only member properties.

Example: ReadOnly Interface

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

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

empObj.empCode = 100; // Compiler Error: Cannot change readonly 'empCode'
```

As you can see above, empcode is read-only, so we can assign a value at the time of creating an object but not after wards.

In the same way you can use read-only<t> to create a read-only type, as shown below.

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
```

In the above example, emp1 is declared as read-only<iemployee> and so values cannot be changed once initialized.

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.

Example: Static Property

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

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

The following example defines a class with static property and method and how to access it.

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
```

The above circle class includes a static property and a static method. Inside the static method calculatearea, the static property can be accessed using this keyword or using the class name circle.Pi.

Now, consider the following example with static and non-static members.

Example: Static and Non-static Members

```
class Circle {
    static pi = 3.14;
    pi = 3;
}

Circle.pi; // returns 3.14

let circleObj = new Circle();
circleObj.pi; // returns 3
```

As you can see, static and non-static fields with the same name can exists without any error. The static field will be accessed using dot notation and the non-static field can be accessed using an object.

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.

For example, consider the following Typescript files: file1.ts and file2.ts

file1.ts

```
var greeting : string = "Hello World!";
```

file2.ts

```
console.log(greeting); //Prints Hello World!

greeting = "Hello TypeScript"; // allowed
```

In file1.Ts, we used the keyword export before the variable. Now, accessing a variable in file2.Ts will give an error. This is because greeting is no longer in the global scope. In order to access greeting in file2.Ts, we must import the file1 module into file2 using the import keyword.

Let's learn **export** and **import** in detail.

Export

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.

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";
```

In the above example, 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.

IMPORT

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

Let's see different ways of importing a module export.

*** Single export from a Module**

EmployeeProcessor.ts

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

*** Entire Module into a Variable**

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 an Export from a Module

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
```

Compiling a Typescript Module

We cannot use typescript modules directly in our application. We need to use the JavaScript for typescript modules. To get the JavaScript files for the typescript modules, we need to compile modules using typescript compiler.

Compilation of a module depends on the target environment you are aiming for. The typescript compiler generates the JavaScript code based on the module target option specified during compilation.

Use the following command to compile a typescript module and generate the JavaScript code.

```
Tsc --module <target> <file path>
```

TYPESCRIPT - NAMESPACE

The namespace is used for logical grouping of functionalities. A namespace can include interfaces, classes, functions and variables to support a single or a group of related functionalities.

A namespace can be created using the namespace keyword followed by the namespace name. All the interfaces, classes etc. Can be defined in the curly brackets { }.

Consider the following example of different string functions in the stringutilities namespace.

StringUtility.ts

```
namespace StringUtility
{
    function ToCapital(str: string): string {
        return str.toUpperCase();
    }

    function SubString(str: string, from: number, length: number = 0): string {
        return str.substr(from, length);
    }
}
```

The above StringUtility.ts file includes the namespace StringUtility which includes two simple string functions.

The StringUtility namespace makes a logical grouping of the important string functions for our application.

By default, namespace components cannot be used in other modules or namespaces. You must export each component to make it accessible outside, using the export keyword as shown below.

Example: Use export in Namespace

```
namespace StringUtility {

    export function ToCapital(str: string): string {
        return str.toUpperCase();
    }

    export function SubString(str: string, from: number, length: number = 0): string {
        return str.substr(from, length);
    }
}
```

Now, we can use the stringutility namespace elsewhere. The following JavaScript code will be generated for the above namespace.

JavaScript: StringUtility.js

```
var StringUtility;
(function (StringUtility) {
    function ToCapital(str){
        return str.toUpperCase();
    }
    StringUtility.ToCapital = ToCapital;
    function SubString(str, from, length) {
        if (length === void 0) { length = 0; }
        return str.substr(from, length);
    }
    StringUtility.SubString = SubString;
})(StringUtility || (StringUtility = {}));
```

As you can see, the above generated JavaScript code for the namespace uses the IIFE pattern to stop polluting the global scope.

Let's use the above stringutility namespace in the employee module, as shown below.

Employee.ts

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

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

In order to use namespace components at other places, first we need to include the namespace using the triple slash reference syntax /// <reference path="path to namespace file" />. after including the namespace file using the reference tag, we can access all the functionalities using the namespace. Above, we used the tocapital() function like this: stringutility.Tocapital()

TYPESCRIPT - GENERIC

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. Generics in typescript is almost similar to C# 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]
```

In the last example, the getarray() function accepts an array of type any. It creates a new array of type any, concats 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.

To solve this, typescript introduced 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.

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

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
```

METHODS AND PROPERTIES OF GENERIC TYPE

When using type variables to create generic components, typescript forces us to use only general methods which are available for every type.

Example: Generic Type Methods and Properties

```
function displayType<T, U>(id:T, name:U): void {

    id.toString(); // OK
    name.toString(); // OK

    id.toFixed();  // Compiler Error: 'toFixed' does not exists on type 'T'
    name.toUpperCase(); // Compiler Error: 'toUpperCase' does not exists on type 'U'

    console.log(typeof(id) + ", " + typeof(name));
}
```

In the above example, id.Tostring() and name.Tostring() method calls are correct because the tostring() method is available for all types. However, type specific methods such as tofixed() for number type or touppercase() for string type cannot be called. The compiler will give an error.

COMPILE TYPESCRIPT PROJECT

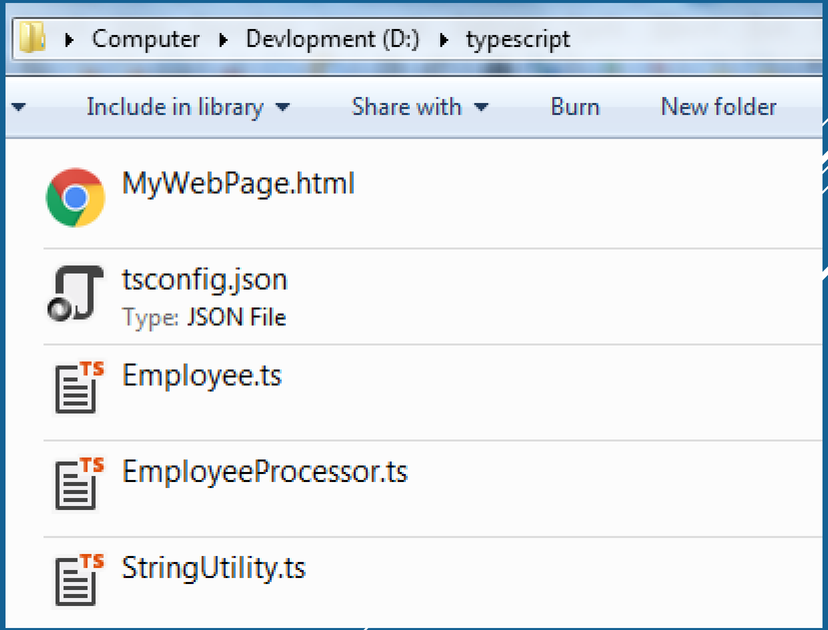
As you know, typescript files can be compiled using the tsc <file name>.ts command. It will be tedious to compile multiple .ts files in a large project. So, typescript provides another option to compile all or certain .Ts files of the project.

“ tsconfig.json “

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.

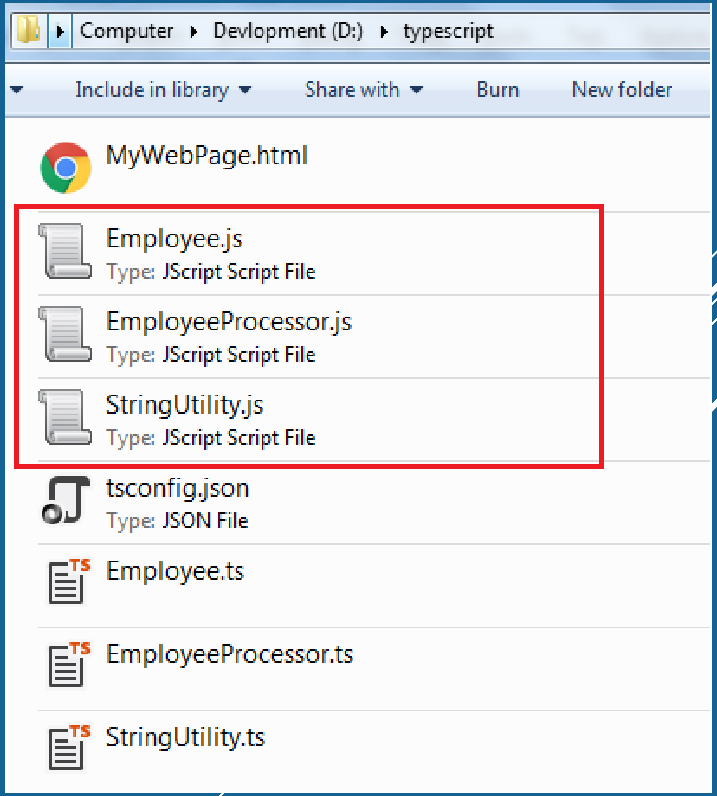
Consider the following simple project which includes two module files, one namespace file, tsconfig.Json and an html file.



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
```

The above tsc command will generate .js files for all .Ts files, as shown below.



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.

If the tsconfig.json file is not found in the root directory, then you can specify the path using the --project or -p option, as shown below.

```
tsc -p <path to tsconfig.json>
```

Until now, we used an empty tsconfig.Json file and so, the typescript compiler used default settings to compile the typescript files. You can set different compiler options in the "compileroptions" property in the tsconfig.Json file, as shown below.

Example: Set compilerOptions in tsconfig.json

```json
{
    "compilerOptions": {
        "module": "amd",
        "noImplicitAny": true,
        "removeComments": true,
        "preserveConstEnums": true,
        "sourceMap": true
    }
}
```

In the above sample tsconfig.Json file, the compileroptions specifies the custom options for the typescript compiler to use when compiling a project.

You can also specify specific files to be compiled by using the "files" option. The files property provides a list of all files to be compiled.

Example: files in tsconfig.json

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

The above files option includes the file names to be compiled. Here, the compiler will only compile the employee.Ts file.

There are two additional properties that can be used to include or omit certain files: include and exclude.

All files specified in include will be compiled, except the ones specified in the exclude property.

All files specified in the exclude option are excluded by the compiler. Note that if a file in include has a dependency on another file, that file cannot be specified in the exclude property.

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"
    ]
}
```

Thus, the tsconfig.Json file includes all the options to indicate the compiler how to compile a project. Learn more about tsconfig.Json here.

https://www.typescriptlang.org/docs/handbook/tsconfig-json.html

TYPESCRIPT - BUILD TOOLS

Build tools are utilities that help automate the transformation and bundling of your code into a single file. Most javascript projects use these build tools to automate the build process.

There are several common build tools available that can be integrated with typescript. We will take a look at how to integrate typescript with some of these tools:

- **Browserify**
- **Grunt**
- **Gulp**
- **Webpack**

Browserify

```
npm install tsify
browserify main.ts -p [ tsify --noImplicitAny ] > bundle.js
```

Grunt

```
npm install grunt-ts
```

You will need to include the grunt config file:

```
module.exports = function(grunt) {
    grunt.initConfig({
        ts: {
                default : {
                src: ["**/*.ts", "!node_modules/**/*.ts"]
            }
        }
    });
    grunt.loadNpmTasks("grunt-ts");
    grunt.registerTask("default", ["ts"]);
};
```

Gulp

```
npm install gulp-typescript
```

You will need to include the gulp config file:

```
var gulp = require("gulp");
var ts = require("gulp-typescript");

gulp.task("default", function () {
                var tsResult = gulp.src("src/*.ts")
        .pipe(ts({
              noImplicitAny: true,
                out: "output.js"
        }));
                return tsResult.js.pipe(gulp.dest("built/local"));
});
```

Webpack

```
npm install ts-loader --save-dev
```

OR

```
npm install awesome-typescript-loader
```

You will need to include the webpack.Config.Js config file:

```
module.exports = {
    entry: "./src/index.tsx",
    output: {
        filename: "bundle.js"
    },
    resolve: {
            // Add '.ts' and '.tsx' as a resolvable extension.
        extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
    },
    module: {
        loaders: [
            // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
            { test: /\.tsx?$/, loader: "ts-loader" } // replace with your plugin of choice
        ]
    }
```