TypeScript

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Agenda

- Introduction to TypeScript
- Benefits of TypeScript
- Setting up the Environment
- TypeScript Compiler
- Hello world Example
- Type Annotations
- Union Type
- Variable Declaration
- Functions
  - Default Parameters
  - Optional Parameters
  - Rest Parameters
  - Arrow-functions
- Classes
- Method
- Access Modifiers
- Static properties
- Accessors
- Inheritance
- Abstract class
- Interfaces
- Generics
- Modules
  - Export
  - Import

Introduction of TypeScript

- TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It's not a replacement for JavaScript, nor does it add any new features to JavaScript code.
- TypeScript allows programmers to use **object-oriented concepts in their code, which is then translated to JavaScript**. It also includes handy features like type safety and compile-time type checking.
- TypeScript are bringing true object oriented web development to the mainstream.
- TypeScript supports definition files that can contain type information of existing JavaScript libraries.
- It's completely free and open-source.

Benefits of TypeScript

- **Static Typing** - TypeScript introduces some datatypes to JavaScript variables like number, string, Boolean, any, object type. These datatypes helps you to use Static Typing feature of TypeScript and gives error when any wrong type is assigned to a variable.
- **Compilation** - TypeScript compiles your code before execution and let you know about compilation errors like syntax errors and mismatch datatypes etc. Compilation is very useful when we have to write large JavaScript programs.
- **Object Oriented programming features** - TypeScript allows you to use OOPS features like classes, interfaces, inheritance, generics and access modifiers like public and private variables.

Benefits of TypeScript

- **Support for Modules** - TypeScript uses Module system. A module is a container of objects. An object in a module is not available to outside world until you export them. You can import other modules and use their objects in your module. Module helps us to write maintainable code.
- **Integration with popular JavaScript libraries using Declaration Files** - TypeScript allows you to use other popular JavaScript libraries like jQuery to use within TypeScript code. Declaration files has extension ".d.ts" and contains each available function in a library. These declaration files provide intelligence when using library functions and variables.

Setting up the environment

- Typescript is an Open Source technology. It can run on any browser, any host, and any OS. You will need the following tools to write and test a Typescript program

- There are three main ways to get the TypeScript tools:
  - Via npm (the Node.js package manager)
  - By installing TypeScript's Visual Studio plugins
  - Use an online playground

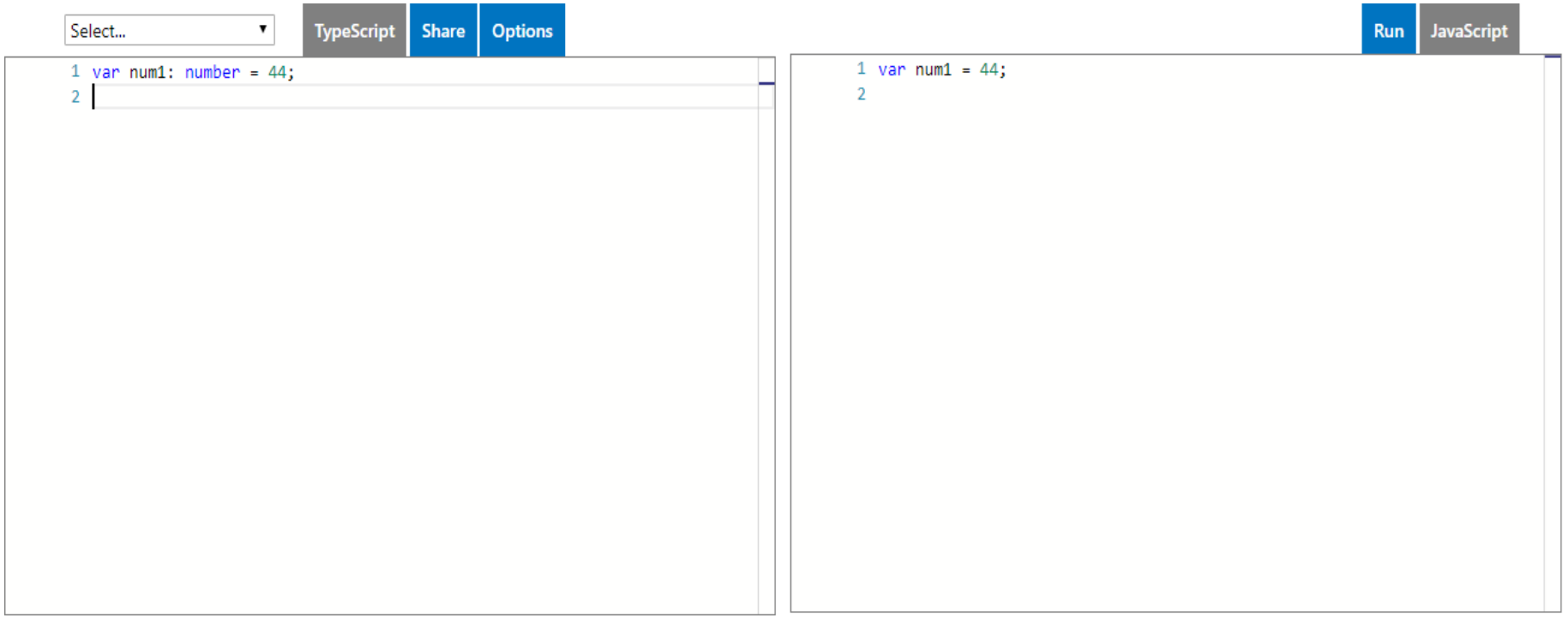
Setting up the environment cont..

- For NPM users:
  - Node is available here – https://nodejs.org/en/download
  - **npm install -g typescript**

- Visual Studio 2017 and Visual Studio 2015 Update 3 include TypeScript by default. If you're using an older version of Visual Studio or a different environment, you can get the TypeScript source code
- Visual Studio offers a wonderful side-by-side view for corresponding TypeScript and JavaScript files. Whenever you save your TypeScript, you can immediately see the changes in your JavaScript

Setting up the environment cont..

- **Typescriptlang.org** provides a web page where we can immediately write and see JavaScript output in a web page. You can click on **Playground** link.



Typescript Compiler

- Once installed, you can start making TypeScript files and adding them to existing applications. TypeScript files can be identified by the ***.ts** extension.
- Since .ts files cannot be directly used in browsers, they must be compiled to regular JavaScript, which can be accomplished in a few ways.
- As with any npm package, you can install it locally or globally, or both, and compile the TS files by running **tsc filename** on the command line.
- Whenever you save a TypeScript file, the Visual Studio plugin **automatically generates** a corresponding JavaScript file with the same name that's ready for use.

Typescript Compiler cont..

- Aside from using an IDE or an automated task runner like Gulp, the simplest way is to use the command line tool tsc as follows:

```
tsc index.ts
```

- The above command will give you a file named index.js. If a .js file with that name already exists, it will be overwritten.
- It's also possible to compile more than one file at once by simply listing them:

```
tsc index.ts main.ts
```

- You can compile all of the .ts files in the current folder with the following command, but keep in mind that it doesn't work recursively:

```
tsc *.ts
```

Typescript Compiler cont..

- Instead of running the tsc command all the time you can use the option --watch.

```
tsc index.ts --watch
```

- Every time there's an update to a TypeScript file it'll recompile the source files to JavaScript.
- You can also compile all your TypeScript files down to a single JavaScript file. This can reduce the number of HTTP requests a browser has to make and improve performance on HTTP 1.x sites. To do this use the --out option like so:

```
tsc *.ts --out app.js
```

Hello World Example

- Create **test.ts** file and write below code inside it

```
var message: string = "Hello World !!";
function printMe() {
    console.log(message);
}
printMe();
```

- At the command line, run the TypeScript compiler: **tsc test.ts**
- The result will be a file test.js which contains the same JavaScript that you fed in

```
var message= "Hello World !!";
function printMe() {
    console.log(message);
}
printMe();
```

Variable Declaration in TypeScript

- When you declare a variable, you have four options –
  - Declare its type and value in one statement.

var name : string = ”mary”;

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

var name : string;

Variable Declaration Cont..

- Declare its value but no type. The variable type will be set to any.

var name = ”mary”;

- Declare neither value not type. In this case, the data type of the variable will be any and will be initialized to undefined.

var name;

Type Annotations

- Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable.
- TypeScript support much the same types as you would expect in JavaScript: numbers, strings, boolean values.

- **Boolean:**
  - The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

```
let isDone: boolean = false;
var isDisabled:boolean=true;
```

Type Annotations cont..

- **Number**
  - Typescript does not have separate integer and float/double type.
  - As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

Type Annotations cont..

- **String**
  - Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

```
let color: string = "blue";
color = 'red';
```

  - You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

```
let fullName: string = `John Smith`;
let sentance:string=`Hello my name is ${fullName}`;
```

Type Annotations cont..

- **Array**
  - TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

  - The second way uses a generic array type, Array<elemType>:

```
let list: Array<number> = [1, 2, 3];
```

Type Annotations cont..

- **Tuple**
  - Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a string and a number:

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

  - When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

Type Annotations cont..

- **Enum**
  - A helpful addition to the standard set of data types from JavaScript is the enum. An enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

  - By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

Type Annotations cont..

- **Any**
  - We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library.

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

  - The any type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
let list: any[] = [1, true, "free"];
list[1] = 100;
```

Type Annotations cont..

- **Void**
  - void is a little like the opposite of any: the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function printMe(): void {
    alert("hello world!!");
}
```

  - Declaring variables of type void is not useful because you can only assign undefined or null to them:

```
let unusable: void = undefined;
```

Type Annotations cont..

- **Null and Undefined**
  - In TypeScript, both undefined and null actually have their own types named undefined and null respectively. Much like void, they're not extremely useful on their own.

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

  - By default null and undefined are subtypes of all other types. That means you can assign null and undefined to something like number.

Type Annotations cont..

- **Object**
  - Object is a type that represents the non-primitive type, i.e. any thing that is not number, string, Boolean, symbol, null, or undefined.
  - An object is an instance which contains set of key value pairs. The values can be scalar values or functions or even array of other objects.

```
var person = {
   firstName:"John",
   lastName:"Smith"
};
//access the object values
console.log(person.firstName)
console.log(person.lastName)
```

TypeScript Union Type

- TypeScript Union Type is a hint of possible data types of a variable. We used union types when we are not sure about the possible variable data type but we know about possible data types are limited.
- **Union Type Syntax**
  - A union type possible data types are separated by pipe symbol (|). Below is the example:

```
var itemAvailable: boolean | number;
```

  - We have a colon (:) after the variable name and then possible data types of **itemAvailable** variable separated by pipe symbol.

TypeScript Union Type Cont..

- For example we have a variable discount. We know a discount variable is always be a number data type. But we are using external JavaScript libraries and that libraries can send discount value in a string data type. So we created a single union type variable which takes both a number and a string value. Below is the example:

```
function calculateDiscount(discount: number | string) {
}

calculateDiscount(11);

calculateDiscount("15");.
```

Functions

- Functions are the fundamental building block of any applications in JavaScript. They're how you build up layers of abstraction, information hiding, and modules.
- Just as in JavaScript, TypeScript functions can be created both as a **named function** or as an **anonymous function**.
- This allows you to choose the most appropriate approach for your application, whether you're building a list of functions in an API or a one-off function to hand off to another function.

Functions cont..

- Named Function

```
function add(x: number, y: number): number {
    return x + y;
}
```

- Anonymous Function

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

Default Parameters

- In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place. These are called default-initialized parameters.

```
function getFullName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

let result1 = getFullName("John");                  // works correctly now, returns "John Smith"
let result2 = getFullName("John", undefined);       // still works, also returns "John Smith"
let result3 = getFullName("John", "Adams", "Sr."); // error, too many parameters
let result4 = getFullName("John", "Adams");
```

Optional Parameters

- In JavaScript, every parameter is optional, and users may leave them off as they see it. When they do, their value is undefined. We can get this functionality in TypeScript by adding a ? to the end of parameters we want to be optional.
- When you write code for functions with optional parameters, you need to provide application logic that handles the cases when the optional parameters aren’t provided.
- Optional parameters always come last in the parameter list of function. We can not put optional parameter first and then required parameters in the function.

Optional Parameters

- By using Optional parameters features, we can declare some parameters in the function optional, so that we need not required to pass value to optional parameters.

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
let result1 = buildName("John");                  // works correctly now
let result2 = buildName("John", "Smith", "Sr."); // error, too many parameters
let result3 = buildName("John", "Smith");
```

Rest Parameters

- Required, optional, and default parameters all have one thing in common: they talk about one parameter at a time. Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take. In JavaScript, you can work with the arguments directly using the arguments variable that is visible inside every function body.
- Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none.

Rest Parameters

- In TypeScript, you can gather these arguments together into a variable
- The compiler will build an array of the arguments passed in with the name given after the ellipsis (...), allowing you to use it in your function.

```
function getFullName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let employeeName = getFUllName("Joseph", "Samuel", "John", "Smith");
```

Arrow-functions

- TypeScript supports simplified syntax for using anonymous functions in expressions. There's no need to use the function keyword, and the fat-arrow symbol (=>) is used to separate function parameters from the body.

```
var getName = () => 'John Smith';
console.log(getName());
```

- Above example is similar to below:

```
var getName = function () {
  return 'John Smith';
};
console.log(getName());
```

Classes

- Classes in TypeScript work mostly the same as classes in other object-oriented languages.
- In TypeScript, the class keyword is syntactic sugar to simplify coding. In the end, your classes will be transpiled into JavaScript objects with prototypal inheritance.
- You can create a class and even add fields, properties, access modifiers(public, private), constructors, and functions.
- Properties and methods declared are often referred as **class members.**

Classes

- The class keyword is followed by the class name. The rules for identifiers must be considered while naming a class.
- A class definition can include the following −
  - **Fields** − A field is any variable declared in a class. Fields represent data pertaining to objects.
  - **Constructors** − Responsible for allocating memory for the objects of the class.
  - **Functions** − Functions represent actions an object can take. They are also at times referred to as methods.
- These components put together are termed as the data members of the class.

Class Example

```
1  class Person {
2      firstName: string;
3      lastName: string;
4      age: number;
5      ssn: string;
6  }
7
8  var p = new Person();
9
10 p.firstName = "John";
11 p.lastName = "Smith";
12 p.age = 29;
13 p.ssn = "123-90-4567";
```

```
1  var Person = (function () {
2      function Person() {
3      }
4      return Person;
5  })();
6  var p = new Person();
7  p.firstName = "John";
8  p.lastName = "Smith";
9  p.age = 29;
10 p.ssn = "123-90-4567";
11
```

Class Constructor

- In JavaScript, you can declare a constructor function and instantiate it with the new keyword. In TypeScript, you can also declare a class and instantiate it with the new operator.
- TypeScript also supports class constructors that allow you to initialize object variables while instantiating the object.
- A class constructor is invoked only once during object creation. In the next example of the Person class, which uses the constructor keyword that initializes the fields of the class with the values given to the constructor.
- The generated ES5 version is shown on the right.

Class Constructor Example

```
1  class Person {
2      firstName: string;
3      lastName: string;
4      age: number;
5      ssn: string;
6
7      constructor(firstName:string, lastName: string,
8          age: number, ssn: string) {
9
10         this.firstName = firstName;
11         this.lastName;
12         this.age = age;
13         this.ssn = ssn;
14     }
15 }
16
17 var p = new Person("John", "Smith", 29, "123-90-4567");
```

```
1  var Person = (function () {
2      function Person(firstName, lastName, age, ssn) {
3          this.firstName = firstName;
4          this.lastName;
5          this.age = age;
6          this.ssn = ssn;
7      }
8      return Person;
9  })();
10 var p = new Person("John", "Smith", 29, "123-90-4567");
11
```

Methods

- When a function is declared in a class, it's called a method. In JavaScript, you need to declare methods on the prototype of an object; but with a class you declare a method by specifying a name followed by parentheses and curly braces, as you would in other object-oriented languages.

```
class myClass{
   doSomething(): void{
      console.log("hello world!!");
   }
}

var obj = new myClass();
obj.doSomething();
```

Access Modifiers

- JavaScript doesn’t have a way to declare a variable or a method as private (hidden from external code). To hide a property (or a method) in an object, you need to create a closure that neither attaches this property to the this variable nor returns it in the closure’s return statement.
- TypeScript supports the common access modifiers that control class members access. Typescript access modifiers are :
  - **public** : available on instances everywhere
  - **private** : not available for access outside the class.
  - **protected** : available on child classes but not on instances directly.

Public

- The public access modifier helps to share the class field data member and member methods visible outside the class area. If we haven't specified our class field as public, private or protected the by default specifier of our class members is public.
- Public keyword makes our class field less restrictive and accessible outside the class. Our data member and member methods can be extended and can be used in other class as well.
- In the next example, We have Person class with some properties. Let's use the private keyword to hide the value of the ssn property. So ssn can't be directly accessed from outside of the person object.

Example

```
class Person {
    public firstName: string;
    public lastName: string;
    private ssn: string;

    constructor(firstName:string, lastName: string, ssn: string) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.ssn = ssn;
    }
}
var p = new Person("John", "Smith","123-45-6789");
console.log("Last name: " + p.lastName + " SSN: " + p.ssn);  // Error: 'ssn' is private;
```

Private

- TypeScript allows you to provide access modifiers with the constructor's arguments, as shown in the following short version of the Person class.

```
class Person {
    constructor(public firstName:string, public lastName: string,
                private ssn: string) {}
}
var p = new Person("John", "Smith","123-45-6789");
console.log("Last name: " + p.lastName + " SSN: " + p.ssn);  // Error: 'ssn' is private;
```

Protected

- The protected modifier acts much like the private modifier with the exception that members declared protected can also be accessed within deriving classes. For example,

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

class Employee extends Person {
    private department: string;
    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }
    public getDetail() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let e = new Employee("John", "Sales");
console.log(e.getDetail());
console.log(e.name); // error
```

Readonly Modifier

- You can make properties readonly by using the **readonly** keyword. Readonly properties must be initialized at their declaration or in the constructor.

```
class Person {
    readonly firstName: string;
    public lastName: string;

    constructor(firstName:string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

let p = new Person("John", "Smith");
p.firstName = "Tom";// error! firstName is readonly.
p.lastName = "Peterson";
```

Static

- We can also create static members and methods of a class, those that are visible on the class itself rather than on the instances. In this example, we use static on the doSomething() method. Each instance accesses this value through prepending the name of the class.

```
class MyClass{

    static doSomething(): void{
        console.log("hello world!!");
    }
}

MyClass.doSometing();
```

Accessors

- TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.
- In this getter and setter methods you use **this** keyword to access the property of the object. It’s mandatory in Typescript.
- Accessors with a get and no set are automatically inferred to be readonly.
- This is helpful when generating a .d.ts(typescript definition) file from your code, because users of your property can see that they can’t change it.

Accessors

- Let’s add public setter and getter methods to the Person class to set and get the value of _ssn.

```
class Person {
    constructor(public firstName: string,
                public lastName: string,  private _ssn?: string) {
    }
    get ssn(): string{
        return this._ssn;
    }
    set ssn(value: string){
        this._ssn = value;
    }
}
var p = new Person("John", "Smith");
p.ssn= "456-70-1234";
console.log("Last name: " + p.lastName + " SSN: " + p.ssn);
```

Inheritance

- Inheritance is the ability of a program to create new classes from an existing class. The class that is extended to create newer classes is called the parent class/super class. The newly created classes are called the child/sub classes.
- JavaScript supports prototypal object-based inheritance, where one object can use another object as a prototype.
- TypeScript has the keyword extends for inheritance of classes, like ES6 and other object-oriented languages. But during transpiling to JavaScript, the generated code uses the syntax of the prototypal inheritance.
- A class inherits from another class using the ‘extends’ keyword. Child classes inherit all properties and methods except private members and constructors from the parent class.

Inheritance cont..

- Inheritance can be classified as −
  - **Single** − Every class can at the most extend from one parent class
  - **Multiple** − A class can inherit from multiple classes. TypeScript doesn't support multiple inheritance.
  - **Multi-level** −A class can inherit from a derived class, thereby making this derived class as the base class for the new class.
- In next example, we have created an **Employee** class that extends a **Person** class. On right side you can see the transpiled JavaScript version of the code.

Inheritance Example

TypeScript | Select... | Share

```typescript
class Person {

    constructor(public firstName: string,
        public lastName: string, public age: number,
        private _ssn: string) {
    }
}

class Employee extends Person{

}
```

Run | JavaScript

```javascript
var __extends = this.__extends || function (d, b) {
    for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
    function __() { this.constructor = d; }
    __.prototype = b.prototype;
    d.prototype = new __();
};
var Person = (function () {
    function Person(firstName, lastName, age, _ssn) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this._ssn = _ssn;
    }
    return Person;
})();
var Employee = (function (_super) {
    __extends(Employee, _super);
    function Employee() {
        _super.apply(this, arguments);
    }
    return Employee;
})(Person);
```

Inheritance Cont..

- If you invoke a method declared in a superclass on the object of the subclass type, you can use the name of this method as if it were declared in the subclass. But sometimes you want to specifically call the method of the superclass, and this is when you should use the **super** keyword.
- The super keyword can be used two ways. In the constructor of a derived class, you invoke it as a method. You can also use the super keyword to specifically call a method of the superclass. It's typically used with method overriding.

Inheritance Example

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

class Employee extends Person {
    private department: string;
    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }
    public getDetail() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let e = new Employee("John", "Sales");
console.log(e.getDetail());
console.log(e.name); // error
```

Method Overriding

- Method Overriding is a mechanism by which the child class redefines the superclass's method. The following example illustrates the same
- For example, if both a superclass and its descendant have a printMe() method, the descendant can reuse the functionality programmed in the superclass and add other functionality as well.

```
class Parent {
    printMe():void {
     alert("calling from Parent class");
   }
}

class Child extends Parent {
   printMe():void {
      alert("calling from child class");
   }
}

let p = new Parent();
let c = new Child();
p.printMe();
c.printMe();
```

Abstract Class

- Abstract classes are base classes from which other classes may be derived. They may not be instantiated directly.
- An abstract class may contain implementation details for its members.
- Abstract class is a class which may have some unimplemented methods. These methods are called abstract methods.
- The abstract keyword is used to define abstract classes as well as abstract methods within an abstract class.
- In the next example, **BaseEmployee** is the abstract class which contains the abstract method **doWork** without any implementation. **Employee** class extends **BaseEmployee** class. In **Employee** class constructor we call **BaseEmployee** constructor using super method.

Abstract Class Example

```
abstract class BaseEmployee {
    firstName: string;
    lastName: string;

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    abstract doWork(): void;
}
```

```
class Employee extends BaseEmployee {
    constructor(firstName: string, lastName: string) {
        super(firstName, lastName);
    }

    doWork(): void {
        console.log(`${this.firstName}, ${this.lastName} doing work...`);
    }
}

let emp = new Employee('John', 'Smith');
emp.doWork(); //Print John, Smith doing work..
```

Interface

- In TypeScript interface, we can define only properties and functions signatures. We do not provide any kind of implementation.
- An object which implements this interface has the responsibility for providing implementation.
- It often helps in providing a standard structure that the deriving classes would follow.
- An interface is a virtual structure that only exists within the context of TypeScript. The TypeScript compiler uses interfaces solely for type-checking purposes.
- Once your code is transpiled to its target language, it will be stripped from its interfaces - JavaScript isn’t typed, there’s no use for them there.

Interface Example

- A TypeScript interface is start with 'interface' keyword. In the above example, we have defined two properties and one functions in an interface.

```
interface IPerson {
    firstName: string;
    lastName: string;
    getFullName(): void;
}

class Person implements IPerson {
    firstName: string;
    lastName: string;

    getFullName = () => console.log(this.firstName + ", " + this.lastName);
}

var p = new Person();
p.firstName = "John";
p.lastName = "Smith";
p.getFullName(); //print 'John, Smith'
```

Interface Example

Select... TypeScript Share Options

```typescript
interface IPerson {
    firstName: string;
    lastName: string;
    getFullName(): void;
}

class Person implements IPerson {
    firstName: string;
    lastName: string;

  getFullName = () =>
  console.log(this.lastName + ", " + this.firstName);
}

var p = new Person();
p.firstName = "John";
p.lastName = "Smith";
p.getFullName(); //print 'John, Smith'
```

Run JavaScript

```javascript
var Person = /** @class */ (function () {
    function Person() {
        var _this = this;
        this.getFullName = function () {
            return console.log(_this.lastName + ", " + _this.firstName);
        };
    }
    return Person;
}());
var p = new Person();
p.firstName = "John";
p.lastName = "Smith";
p.getFullName(); //print 'John, Smith'
```

TypeScript Interface vs Abstract Class

Interface	Abstract Class
All members are abstract.	Some members are abstract, and some are fully implemented.
Interfaces support multiple inheritances	Abstract class does not support multiple inheritances.
TypeScript does not produce any code in compiled JavaScript file.	Abstract class compile to JavaScript functions

Generics

- TypeScript supports parameterized types, also known as generics, which can be used in a variety of scenarios. For example, you can create a function that can take values of any type, but during its invocation in a particular context you can explicitly specify a concrete type.
- It creates a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.
- Generics enable common code to be reused and make encapsulation more obvious.
- An example of this is found during development of an application that performs common operations on different models. Often developers find existing code that acts similar to what they need. This leads to copying and pasting code, changing variables as needed - this is wrong

Generics Cont..

- **any** type is generic, that cause the function to accept any and all types for the type of argument.

```
function logResult(arg: any): any {
    console.log(arg);
    return arg;
}
var result = logResult('hello world!!');
var result1 = logResult(1);
```

- In the above example, the return type of the function is also any. The type passed as argument and the type returned, need not be the same.

Generics Example

- We've now added a type variable T to the function. This T allows us to capture the type the user provides (e.g. number), so that we can use that information later. Here, we use T again as the return type.
- On inspection, we can now see the same type is used for the argument and the return type.

```
function logResult<T>(arg: T): T {
    console.log(arg);
    return arg;
}
var result = logResult<string>('hello generics');
var result1 = logResult<number>(12);
```

Modules

- In any programming language, splitting code into modules helps organize the application into logical and possibly reusable units. Modularized applications allow programming tasks to be split between software developers more efficiently. Developers get to decide which logic should be exposed by the module for external use and which should be used internally.
- Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the **export** forms.
- Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the **import** forms.

Modules Cont..

- Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.
- Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known modules loaders used in JavaScript are the CommonJS and SystemJS module loader for Node.js and require.js for Web applications.
- In TypeScript, just as in ECMAScript 2015, any file containing a top-level **import** or **export** is considered a module. Conversely, a file without any top-level import or export declarations is treated as a script whose contents are available in the global scope (and therefore to modules as well).

Export

- A module can export any number of functions, classes or variables. By default, the objects are exported with their original names. We can change this, if required. A module can have a default exported member as well. Following snippet shows examples of different export statements.

```
//file myModule.ts
//Inline export with same name
export class myClass{
  //the code goes here
}
export function myFunc(){
  //the code goes here
}
```

```
//Exporting a group of members
export {
  myClass,
  myFunc
};

//Rename while exporting
export {
  myClass as AnotherClass,
  myFunc as anotherFunc
};
```

Import

- When a module imports another module, it may import all exported members, some of them or none at all. The importing module also has the option to rename the importing object. The following snippet shows examples of different import statements.

```
//app.ts
//Importing all exported objects
import * as module1 from "myModule";

//Importing selected objects
import {myClass, myFunc} from "myModule";

//Importing selected objects and rename them
import {myClass as ModuleMyClass, myFunc as ModuleFunc} from "myModule";
```