

Full Stack Module-III

Manual V8.3

ANUDIP FOUNDATION



ICONS AND THEIR MEANING



HINTS:
Get ready for helpful insites on difficult topics and questions.



STUDENTS:
This icon symbolize important instreutions and guides for the students.



TEACHERS/TRAINERS:
This icon symbolize important instreutions and guides for the trainers.

Lesson No	Lesson Name	Practical Duration (Minutes)	Theory Duration (Minutes)	Page No
1	Introduction to Typescript	60	60	03
2	The Type System	60	60	06
3	OOPS in typescript	60	60	19
4	Namespaces and Modules	60	60	28
5	Generics	60	60	34
6	Decorators	60	60	44
7	Typescript essentials	60	60	51

Total Duration: __Hours

Lesson 1: Introduction to Typescript (120 minutes)

Objective: After completing this lesson you will be able to : <ul style="list-style-type: none">• Learn about the basics of typescript• The connection between JavaScript and TypeScript• Benefits of typescript	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Introduction to Typescript

TypeScript is an open-source, object-oriented programming language, which is developed and maintained by Microsoft under the Apache 2 license. It was introduced by Anders Hejlsberg, a core member of the development team of C# language.

JavaScript and Typescript

TypeScript is a strongly typed superset of JavaScript which compiles to plain JavaScript. It is a language for application-scale JavaScript development, which can be executed on any browser, any Host, and any Operating System. All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.

TypeScript supports other JS libraries. Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

JavaScript is TypeScript. This means that any valid .js file can be renamed to .ts and compiled with other TypeScript files.

TypeScript is not directly run on the browser. It needs a compiler to compile and generate in JavaScript file. TypeScript is the ES6 version of JavaScript with some additional features.

The popular JavaScript framework Angular 2.0 is written in TypeScript. Mastering TypeScript can help programmers to write object-oriented programs and have them compiled to JavaScript, both on server side and client side.

The Portability factor

TypeScript is portable. TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, TypeScript doesn't need a dedicated VM or a specific runtime environment to execute.

The major components of TypeScript

TypeScript has the following three components –

1. Language – It comprises of the syntax, keywords, and type annotations.
2. The TypeScript Compiler – The TypeScript compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.
3. The TypeScript Language Service – The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications. The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.

Benefits of using TypeScript?

TypeScript comes with certain benefits:

- TypeScript supports Static typing, Strongly type, Modules, Optional Parameters, etc.
- TypeScript supports object-oriented programming features such as classes, interfaces, inheritance, generics, etc.
- TypeScript is fast, simple, and most importantly, easy to learn.
- TypeScript provides the error-checking feature at compilation time. It will compile the code, and if any error found, then it highlighted the mistakes before the script is run.
- TypeScript supports all JavaScript libraries because it is the superset of JavaScript.
- TypeScript support reusability because of the inheritance.
- TypeScript make app development quick and easy as possible, and the tooling support of TypeScript gives us autocompletion, type checking, and source documentation.
- TypeScript has a definition file with .d.ts extension to provide a definition for external JavaScript libraries.
- TypeScript supports the latest JavaScript features, including ECMAScript 2015.
- TypeScript gives all the benefits of ES6 plus more productivity.
- Developers can save a lot of time with TypeScript.

Practical Session

Instructions: *Create the code....using an editor....and save it with HTML extension.*

1. Create a program using Typescript that will display Hello World

```
var message:string = "Hello World"
```

```
console.log(message)
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
```

```
var message = "Hello World";
```

```
console.log(message);
```

2. Create a program using Typescript that will display your name

3. Create a program in Typescript that will display the string, "Delhi is the capital of India"

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- TypeScript is an open-source, object-oriented programming language, which is developed and maintained by Microsoft under the Apache 2 license.
- All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.
- JavaScript is TypeScript. This means that any valid .js file can be renamed to .ts and compiled with other TypeScript files.
- TypeScript has the following three components: Language, compiler, language service
- TypeScript supports object-oriented programming features such as classes, interfaces, inheritance, generics, etc.

Testing your skills

1. Typescript was introduced by

a) Lary Ellison, b) Anders Hejlsberg, c) Steve Jobs, d) None of the above

2. Which of the following statements are true:

a) Typescript is a superset of JavaScript, b) JavaScript is a superset of Typescript, c) Both (a) and (b), d) None of the above

3. TypeScript is the _____version of JavaScript with some additional features

a) ES2, b) ES5, c) ES6, d) ES3

4. Which of the following are components of Typescript?

a) Language, b) Compiler, c) Language Service, d) All of the above

5. Which of these are true for Typescript benefits

a) autocompletion, b) type checking, c) source documentation, d) All of the above

Lesson 2: The Type System (120 minutes)

Objective: After completing this lesson you will be able to : <ul style="list-style-type: none">• Learn about boolean, number, string array, tuple, enum, null, undefined, void, never, and objects• Learn about type assertions, classes, and interfaces• Build a basic typescript program	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

In TypeScript, we support much the same types as you would expect in JavaScript, with a convenient enumeration type thrown in to help things along.

Boolean

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

```
let isDone: boolean = false;
```

Number

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;
```

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type string to refer to these textual datatypes. 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 = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;
```

This is equivalent to declaring sentence like so:

```
let sentence: string = "Hello, my name is " + fullName + ".\n\n" +
    "I'll be " + (age + 1) + " years old next month.";
```

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];
```

Tuple

Tuple types allow you to express an array with a fixed number of elements whose types are 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].substring(1)); // OK
console.log(x[1].substring(1)); // Error, 'number' does not have 'substring'
Accessing an element outside the set of known indices fails with an error:
```

```
x[3] = "world"; // Error, Property '3' does not exist on type '[string, number]'.
```

```
console.log(x[5].toString()); // Error, Property '5' does not exist on type '[string, number]'.
```

Enum

A helpful addition to the standard set of datatypes from JavaScript is the enum. As in languages like C#, 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;
```

Or, even manually set all the values in the enum:

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value 2 but weren't sure what that mapped to in the Color enum above, we could look up the corresponding name:

```
enum Color {Red = 1, Green, Blue}
```

```
let colorName: string = Color[2];
```

```
console.log(colorName); // Displays 'Green' as its value is 2 above
```

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. In these cases, we want to opt-out of type checking and let the values pass through compile-time checks. To do so, we label these with the any type:

```
let notSure: any = 4;
```

```
notSure = "maybe a string instead";
```

```
notSure = false; // okay, definitely a boolean
```

The any type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type checking during compilation. You might expect Object to play a similar role, as it does in other languages. However, variables of type Object only allow you to assign any value to them. You can't call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
```

```
notSure.ifItExists(); // okay, ifItExists might exist at runtime
```

```
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)
```

```
let prettySure: Object = 4;
```

```
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

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;
```

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 warnUser(): void {  
    console.log("This is my warning message");  
}
```

Declaring variables of type void is not useful because you can only assign null (only if `--strictNullChecks` is not specified, see next section) or undefined to them:

```
let unusable: void = undefined;  
unusable = null; // OK if `--strictNullChecks` is not given
```

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.

However, when using the `--strictNullChecks` flag, null and undefined are only assignable to any and their respective types (the one exception being that undefined is also assignable to void). This helps avoid many common errors. In cases where you want to pass in either a string or null or undefined, you can use the union type `string | null | undefined`.

Never

The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type never when narrowed by any type guards that can never be true.

The never type is a subtype of, and assignable to, every type; however, no type is a subtype of, or assignable to, never (except never itself). Even any isn't assignable to never.

Some examples of functions returning never:

```
// Function returning never must have unreachable end point
```

```
function error(message: string): never {  
    throw new Error(message);  
}
```

```
// Inferred return type is never
```

```
function fail() {  
    return error("Something failed");  
}
```

```
// Function returning never must have unreachable end point
```

```
function infiniteLoop(): never {  
    while (true) {  
        }  
}
```

Object

Object is a type that represents the non-primitive type, i.e. anything that is not number, string, boolean, symbol, null, or undefined.

With object type, APIs like Object.create can be better represented. For example:

```
declare function create(o: object | null): void;
```

```
create({ prop: 0 }); // OK
```

```
create(null); // OK
```

```
create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

Type assertions

Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.

Type assertions are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms. One is the "angle-bracket" syntax:

```
let someValue: any = "this is a string";
```

```
let strLength: number = (<string>someValue).length;
```

And the other is the as-syntax:

```
let someValue: any = "this is a string";
```

```
let strLength: number = (someValue as string).length;
```

The two samples are equivalent. Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only as-style assertions are allowed.

A note about let

You may've noticed that so far, we've been using the `let` keyword instead of JavaScript's `var` keyword which you might be more familiar with. The `let` keyword is actually a newer JavaScript construct that TypeScript makes available. We'll discuss the details later, but many common problems in JavaScript are alleviated by using `let`, so you should use it instead of `var` whenever possible.

Installing TypeScript

There are two main ways to get the TypeScript tools:

1. Via npm (the Node.js package manager)
2. By installing TypeScript's Visual Studio plugins

Visual Studio 2017 and Visual Studio 2015 Update 3 include TypeScript by default.

For NPM users:

```
> npm install -g typescript
```

Building your first TypeScript file

In your editor, type the following JavaScript code in greeter.ts:

```
function greeter(person) {  
    return "Hello, " + person;  
}
```

```
let user = "Jane User";
```

```
document.body.textContent = greeter(user);
```

Compiling your code

We used a .ts extension, but this code is just JavaScript. You could have copy/pasted this straight out of an existing JavaScript app.

At the command line, run the TypeScript compiler:

```
tsc greeter.ts
```

The result will be a file greeter.js which contains the same JavaScript that you fed in. We're up and running using TypeScript in our JavaScript app!

Now we can start taking advantage of some of the new tools TypeScript offers. Add a `: string` type annotation to the `'person'` function argument as shown here:

```
function greeter(person: string) {  
    return "Hello, " + person;  
}
```

```
let user = "Jane User";
```

```
document.body.textContent = greeter(user);
```

Type annotations

Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable. In this case, we intend the `greeter` function to be called with a single string parameter. We can try changing the call `greeter` to pass an array instead:

```
function greeter(person: string) {  
    return "Hello, " + person;  
}
```

```
let user = [0, 1, 2];
```

```
document.body.textContent = greeter(user);
```

Re-compiling, you'll now see an error:

```
error TS2345: Argument of type 'number[]' is not assignable to parameter of type 'string'.
```

Similarly, try removing all the arguments to the `greeter` call. TypeScript will let you know that you have called this function with an unexpected number of parameters. In both cases, TypeScript can offer static analysis based on both the structure of your code, and the type annotations you provide.

Notice that although there were errors, the greeter.js file is still created. You can use TypeScript even if there are errors in your code. But in this case, TypeScript is warning that your code will likely not run as expected.

Interfaces

Let's develop our sample further. Here we use an interface that describes objects that have a firstName and lastName field. In TypeScript, two types are compatible if their internal structure is compatible. This allows us to implement an interface just by having the shape the interface requires, without an explicit implements clause.

```
interface Person {  
    firstName: string;  
    lastName: string;  
}  
  
function greeter(person: Person) {  
    return "Hello, " + person.firstName + " " + person.lastName;  
}  
  
let user = { firstName: "Jane", lastName: "User" };  
  
document.body.textContent = greeter(user);
```

Classes

Finally, let's extend the example one last time with classes. TypeScript supports new features in JavaScript, like support for class-based object-oriented programming.

Here we're going to create a Student class with a constructor and a few public fields. Notice that classes and interfaces play well together, letting the programmer decide on the right level of abstraction.

Also of note, the use of public on arguments to the constructor is a shorthand that allows us to automatically create properties with that name.

```
class Student {  
    fullName: string;
```



```
constructor(public firstName: string, public middleInitial: string, public lastName: string) {  
    this.fullName = firstName + " " + middleInitial + " " + lastName;  
}  
}
```

```
interface Person {  
    firstName: string;  
    lastName: string;  
}
```

```
function greeter(person: Person) {  
    return "Hello, " + person.firstName + " " + person.lastName;  
}
```

```
let user = new Student("Jane", "M.", "User");
```

```
document.body.textContent = greeter(user);
```

Re-run `tsc greeter.ts` and you'll see the generated JavaScript is the same as the earlier code. Classes in TypeScript are just a shorthand for the same prototype-based OO that is frequently used in JavaScript.

Running your TypeScript web app

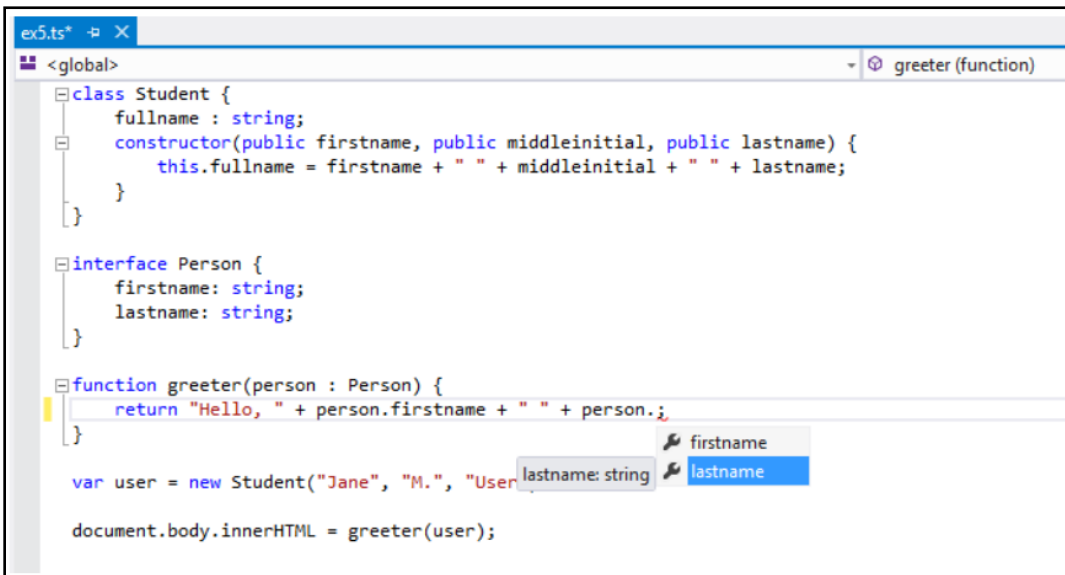
Now type the following in `greeter.html`:

```
<!DOCTYPE html>  
<html>  
  <head><title>TypeScript Greeter</title></head>  
  <body>  
    <script src="greeter.js"></script>  
  </body>  
</html>
```

Open greeter.html in the browser to run your first simple TypeScript web application!

Optional: Open greeter.ts in Visual Studio, or copy the code into the TypeScript playground. You can hover over identifiers to see their types. Notice that in some cases these types are inferred automatically for you. Re-type the last line, and see completion lists and parameter help based on the types of the DOM elements. Put your cursor on the reference to the greeter function, and hit F12 to go to its definition. Notice, too, that you can right-click on a symbol and use refactoring to rename it.

The type information provided works together with the tools to work with JavaScript at application scale.



Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a simple class-based example using Typescript

```
class Greeter {  
  
  greeting: string;  
  
  constructor(message: string) {  
  
    this.greeting = message;}  
  
  greet() {  
  
    return "Hello, " + this.greeting;  
  }  
}
```

```
}}let greeter = new Greeter("world");
```

2. Create a new class named Day using Typescript

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.
- All numbers in TypeScript are floating point values
- TypeScript, like JavaScript, allows you to work with arrays of values.
- Tuple types allow you to express an array with a fixed number of elements whose types are known, but need not be the same
- The never type represents the type of values that never occur. For instance, never is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns
- Object is a type that represents the non-primitive type, i.e. anything that is not number, string, boolean, symbol, null, or undefined.
- A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data.
- Type annotations in TypeScript are lightweight ways to record the intended contract of the function or variable.

Testing your skills

1. All numbers in TypeScript are ____point values

a) Floating, b) Regular, c) Algebraic, d) None of the above

2. Typescript supports which of the following literals

a) Binary, b) Octal, c) Both (a) and (b), d) None of the above

3. TypeScript also uses _____ to surround string data

a) double quotes, b) Back slash, c) Hash, d) None of the above

4. Array types in Typescript can be written in which of the following ways:

a) [], b) <>, c) Both (a) and (b), d) Only (a)

5. _____ allow you to express an array with a fixed number of elements whose types are known

a) Tuple, b Enum, c) Any, d) Never

Lesson 3: OOPS In TypeScript (120 minutes)

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">• Classes, Class properties, Static Properties• Constructors, getters & setters• Inheritance, Abstract classes, Interfaces• Access modifiers	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Object Oriented Programming In Typescript

Traditional JavaScript uses functions and prototype-based inheritance to build up reusable components, but this may feel a bit awkward to programmers more comfortable with an object-oriented approach, where classes inherit functionality and objects are built from these classes. Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers will be able to build their applications using this object-oriented class-based approach. In TypeScript, we allow developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

Classes

A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class. JavaScript ES5 or earlier didn't support classes. Typescript gets this feature from ES6.

Syntax to create classes

```
class class_name {  
    //class scope  
}
```

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.

Let's take a look at a simple class-based example:

```
class Greeter {  
  greeting: string;  
  constructor(message: string) {  
    this.greeting = message;  
  }  
  greet() {  
    return "Hello, " + this.greeting;  
  }  
}  
  
let greeter = new Greeter("world");
```

Breaking up this entire example code, we have the following explanation:

The syntax should look familiar if you've used C# or Java before. We declare a new class Greeter. This class has three members: a property called greeting, a constructor, and a method greet.

You'll notice that in the class when we refer to one of the members of the class we prepend this.. This denotes that it's a member access.

In the last line we construct an instance of the Greeter class using new. This calls into the constructor we defined earlier, creating a new object with the Greeter shape, and running the constructor to initialize it.

Constructors

Constructors are identified with keyword "constructor". A Constructor is a special type of method of a class and it will be automatically invoked when an instance of the class is created. A class may contain at least one constructor declaration.

If a class has no constructor, a constructor is provided automatically. A class can have any number of constructors. When you are using the attribute public or private with constructor parameters, a field is automatically created, which is assigned the value.

A basic example:

```
class MyExample {  
  public FName: string;  
  public Lname: string;  
  constructor (Fname: string, Lname: string) {  
    this.Fname = FName;  
    this.Lname = Lname;  
    alert("Company name="+this.Fname+" "+this.Lname);  
  }  
}  
  
window.onload = () =>  
{  
  var data = new MyExample ("Mcn","Solution");  
}
```

Getters in TypeScript

Sometimes it is desirable to allow access to a property that returns a dynamically computed value, or you may want to reflect the status of an internal variable without requiring the use of explicit method calls.

In JavaScript, this can be accomplished with the use of a getter. It is not possible to simultaneously have a getter bound to a property and have that property actually hold a value, although it is possible to use a getter and a setter in conjunction to create a type of pseudo-property.

Note the following when working with the getter (GET) syntax:

- It can have an identifier which is either a number or a string;
- It must have exactly zero parameters (see Incompatible ES5 change: literal getter and setter functions must now have exactly zero or one arguments for more information);
- It must not appear in an object literal with another get or with a data entry for the same property (`{ get x() { }, get x() { } }` and `{ x: ..., get x() { } }` are forbidden).

A sample example that defines GETTER on new objects in object initializers:

```
var obj = {  
  log: ['example','test'],  
  get latest() {  
    if (this.log.length == 0) return undefined;  
    return this.log[this.log.length - 1];  
  }  
}  
  
console.log(obj.latest); // "test".
```

Do keep in mind that Getters give you a way to define a property of an object, but they do not calculate the property's value until it is accessed. A getter defers the cost of calculating the value until the value is needed, and if it is never needed, you never pay the cost.

Setters in TypeScript

In JavaScript, a setter can be used to execute a function whenever a specified property is attempted to be changed. Setters are most often used in conjunction with getters to create a type of pseudo-property. It is not possible to simultaneously have a setter on a property that holds an actual value.

Note the following when working with the setter (SET) syntax:

- It can have an identifier which is either a number or a string;
- It must have exactly one parameter (see Incompatible ES5 change: literal getter and setter functions must now have exactly zero or one arguments for more information);
- It must not appear in an object literal with another set or with a data entry for the same property.
- ({ set x(v) { } }, set x(v) { } } and { x: ..., set x(v) { } } are forbidden)

A sample example that defines SETTER on new objects in object initializers:

```
var language = {  
  set current(name) {  
    this.log.push(name);  
  },  
  log: []  
}  
  
language.current = 'EN';  
console.log(language.log); // ['EN']  
  
language.current = 'FA';  
console.log(language.log); // ['EN', 'FA']
```

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take an example:

```
class Animal {
  move(distanceInMeters: number = 0) {
    console.log(`Animal moved ${distanceInMeters}m.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log('Woof! Woof!');
  }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

This example shows the most basic inheritance feature: classes inherit properties and methods from base classes. Here, Dog is a derived class that derives from the Animal base class using the extends keyword. Derived classes are often called subclasses, and base classes are often called superclasses.

Because Dog extends the functionality from Animal, we were able to create an instance of Dog that could both bark() and move().

Abstract Classes

A TypeScript Abstract class is a class which may have some unimplemented methods. These methods are called abstract methods. We can't create an instance of an abstract class. But other classes can derived from abstract class and reuse the functionality of base class.

A basic example to create abstract class:

```
abstract class BaseEmployee {

  abstract doWork(): void;

  workStarted(): void {
    console.log('work started.');
```

```
  }
}
```

In above example, we have created an abstract class. First method doWork is abstract and we put abstract keyword before the method name. Abstract method does not have any implementation. Second method workStarted has implementation and it is not an abstract method.

Interfaces

TypeScript has built-in support for interfaces. An interface defines the specifications of an entity. It lays out the contract that states what needs to be done but doesn't specify how it will be done. An interface contains the name of all the properties along with their types. It also includes the signature for functions along with the type of arguments and return type.

A class or function can implement an interface to define the implementation of the properties as defined in that interface.

Let's observe an example:

```
interface IPerson {  
    firstName:string,  
    lastName:string,  
    sayHi: ()=>string  
}  
  
var customer:IPerson = {  
    firstName:"Tom",  
    lastName:"Hanks",  
    sayHi: ():string =>{return "Hi there"}  
}  
  
console.log("Customer Object ")  
console.log(customer.firstName)  
console.log(customer.lastName)  
console.log(customer.sayHi())  
  
var employee:IPerson = {  
    firstName:"Jim",  
    lastName:"Blakes",  
    sayHi: ():string =>{return "Hello!!!"}  
}  
  
console.log("Employee Object ")  
console.log(employee.firstName);  
console.log(employee.lastName);
```

The output will be:

```
Customer object  
Tom  
Hanks  
Hi there  
Employee object  
Jim  
Blakes  
Hello!!!
```

Access Modifiers

Like others programming languages, TypeScript supports access modifiers at the class level. TypeScript supports three access modifiers - public, private, and protected.

Public - By default, members (properties and methods) of TypeScript class are public - so you don't need to prefix members with the public keyword. Public members are accessible everywhere without restrictions

Private - A private member cannot be accessed outside of its containing class. Private members can be accessed only within the class.

Protected - A protected member cannot be accessed outside of its containing class. Protected members can be accessed only within the class and by the instance of its sub/child class.

In compiled JavaScript code, there will be no such types of restriction on the members.

Let's have a look at a typescript code using the access modifiers:

```
class Foo {
  private x: number;
  protected y: number;
  public z: number;
  saveData(foo: Foo): void {
    this.x = 1; // ok
    this.y = 1; // ok
    this.z = 1; // ok

    foo.x = 1; // ok
    foo.y = 1; // ok
    foo.z = 1; // ok
  }
}

class Bar extends Foo {
  getData(foo: Foo, bar: Bar) {
    this.y = 1; // ok
    this.z = 1; // ok

    bar.y = 1; // ok
    bar.z = 1; // ok
    foo.z = 1; // ok

    foo.x = 1; // Error, x only accessible within A
    bar.x = 1; // Error, x only accessible within A
    bar.y = 1; // Error, y only accessible through instance of B
  }
}
```

Now, focus on the compiled javascript code for the above typescript example:

```
var Foo = (function () {  
  function Foo() {  
  }  
  Foo.prototype.saveData = function (foo) {  
    this.x = 1;  
    this.y = 1;  
    this.z = 1;  
    foo.x = 1;  
    foo.y = 1;  
    foo.z = 1;  
  };  
  return Foo;  
})();  
var Bar = (function (_super) {  
  __extends(Bar, _super);  
  function Bar() {  
    return _super.apply(this, arguments) || this;  
  }  
  Bar.prototype.getData = function (foo, bar) {  
    this.y = 1;  
    this.z = 1;  
    bar.y = 1;  
    bar.z = 1;  
    foo.z = 1;  
    foo.x = 1;  
    bar.x = 1;  
    bar.y = 1;  
  };  
  return Bar;  
})(Foo);
```

Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a program that will declare a class in Typescript

```
class Car {  
  //field  
  engine:string;  
  //constructor  
  constructor(engine:string) {  
    this.engine = engine  
  }  
  //function  
  disp():void {  
    console.log("Engine is : "+this.engine)  
  }  
}
```

2. Create a program that will declare a class named Root.

3. Create a program that will declare a class named Animals.

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class.
- A class definition will include fields, definition, and functions
- A Constructor is a special type of method of a class and it will be automatically invoked when an instance of the class is created.
- A TypeScript Abstract class is a class which may have some unimplemented methods. These methods are called abstract methods.

Testing your skills

1. A class definition can include which of the following
a) Fields, b) Constructors, c) Functions, d) All of the above
2. _____ is a special type of method of a class that is invoked when an instance of the class is created.
a) Constructor, b) Getters, c) Setters, d) None of the above
3. There are _ types of Access Modifiers in typeScript.
a) 2, b) 4, c) 3, d) 5
4. _____ is a class which may have some unimplemented methods
a) Constructor, b) Abstract Classes, c) Interfaces, d) Getters
- 5) _____ defines the specifications of an entity
a) Constructor, b) Abstract Classes, c) Interfaces, d) Access Modifiers

Lesson 4: Namespace and modules (120 minutes)

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">NamespacesModulesMultiple files	Materials Required: <ul style="list-style-type: none">Computer With Windows XP and aboveStable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Namespaces

The namespace is a way which is used for logical grouping of functionalities. It encapsulates the features and objects that share common relationships. It allows us to organize our code in a much cleaner way.

A namespace is also known as internal modules. A namespace can also include interfaces, classes, functions, and variables to support a group of related functionalities.

Unlike JavaScript, namespaces are inbuilt into TypeScript. In JavaScript, the variables declarations go into the global scope. If the multiple JavaScript files are used in the same project, then there will be a possibility of confusing new users by overwriting them with a similar name. Hence, the use of TypeScript namespace removes the naming collisions.

Namespace Declaration

We can declare the namespace as below:

```
namespace <namespace_name> {  
    export interface I1 { }  
    export class c1{ }  
}
```

Just focus on the above example to witness that the namespace is created using the namespace keyword followed by the namespace_name. All the interfaces, classes, functions, and variables can be defined in the curly braces{ } by using the export keyword. The export keyword makes each component accessible to outside the namespaces.

To access the interfaces, classes, functions, and variables in another namespace, we can use the following syntax:

```
namespaceName.className;  
namespaceName.functionName;
```

If the namespace is in separate TypeScript file, then it must be referenced by using triple-slash (///) reference syntax.

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

Let's focus on a sample example

Creating the namespace file - studentCalc

```
namespace studentCalc{  
  export function AnualFeeCalc(feeAmount: number, term: number){  
    return feeAmount * term;  
  }  
}
```

The main file - app.ts

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

```
let TotalFee = studentCalc.AnualFeeCalc(1500, 4);
```

```
console.log("Output: " +TotalFee);
```

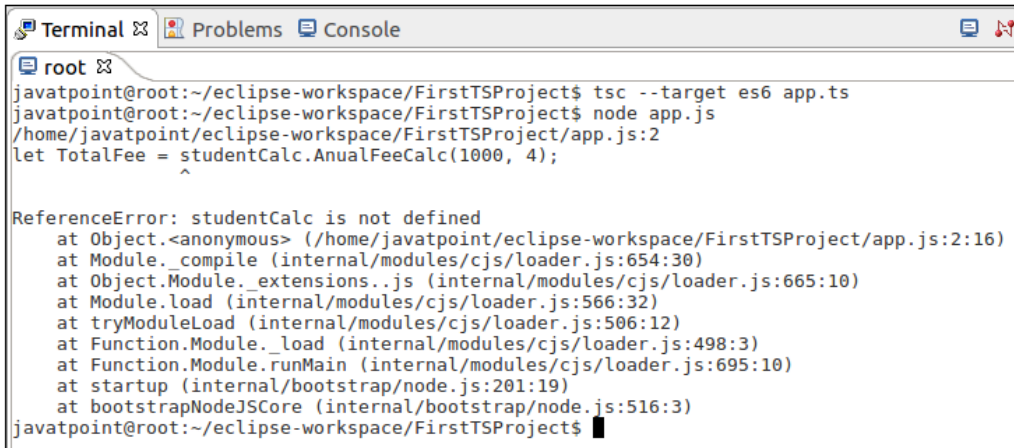
Compiling and Executing Namespaces

Open the terminal and go to the location where you stored your project. Then, type the following command.

```
$ tsc --target es6 app.ts
```

```
$ node app.js
```

We will see the output below:



```
Terminal Problems Console
root
javatpoint@root:~/eclipse-workspace/FirstTSProject$ tsc --target es6 app.ts
javatpoint@root:~/eclipse-workspace/FirstTSProject$ node app.js
/home/javatpoint/eclipse-workspace/FirstTSProject/app.js:2
let TotalFee = studentCalc.AnuualFeeCalc(1000, 4);
                  ^
ReferenceError: studentCalc is not defined
    at Object.<anonymous> (/home/javatpoint/eclipse-workspace/FirstTSProject/app.js:2:16)
    at Module._compile (internal/modules/cjs/loader.js:654:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:665:10)
    at Module.load (internal/modules/cjs/loader.js:566:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:506:12)
    at Function.Module._load (internal/modules/cjs/loader.js:498:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:695:10)
    at startup (internal/bootstrap/node.js:201:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:516:3)
javatpoint@root:~/eclipse-workspace/FirstTSProject$
```

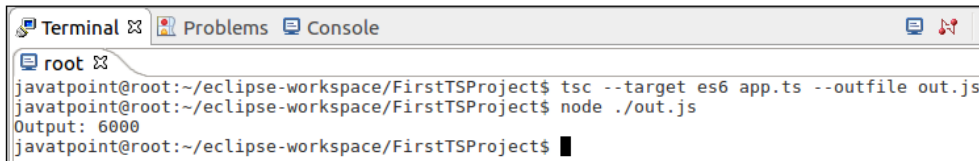
The error “studentCalc is not defined” is being displayed.

So, the correct way to compile and execute the above code, we need to use the following command in the terminal window.

```
$ tsc --target es6 app.ts --outfile out.js
```

```
$ node ./out.js
```

Now, we can see the following output:



```
Terminal Problems Console
root
javatpoint@root:~/eclipse-workspace/FirstTSProject$ tsc --target es6 app.ts --outfile out.js
javatpoint@root:~/eclipse-workspace/FirstTSProject$ node ./out.js
Output: 6000
javatpoint@root:~/eclipse-workspace/FirstTSProject$
```

Multiple Files or Nested Namespace

A namespace also allows us to define one namespace into another namespace. We can access the members of the nested namespace by using the dot(.) operator. The following example helps us to understand the nested namespace more clearly.

Let's observe an example:

Nested NameSpace file - StoreCalc

```
namespace invoiceCalc {
  export namespace invoiceAccount {
    export class Invoice {
      public calculateDiscount(price: number) {
```

```
        return price * .60;
    }
}
}
```

The main file - app.ts

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

```
let invoice = new invoiceCalc.invoiceAccount.Invoice();
```

```
console.log("Output: " +invoice.calculateDiscount(400));
```

Now compile and execute the above code with the below command.

```
$ tsc --target es6 app.ts --outfile out.js
```

```
$ node ./out.js
```

It produces the following output.

Output: 240

Modules

A module is a way to create a group of related variables, functions, classes, and interfaces, etc. It executes in the local scope, not in the global scope. In other words, the variables, functions, classes, and interfaces declared in a module cannot be accessible outside the module directly. We can create a module by using the export keyword and can use in other modules by using the import keyword.

Modules import another module by using a module loader. At runtime, the module loader is responsible for locating and executing all dependencies of a module before executing it. The most common modules loaders which are used in JavaScript are the CommonJS module loader for Node.js and require.js for Web applications.

Modules can be divided into two categories:

1. Internal Module
2. External Module

Internal Modules

Internal modules were in the earlier version of Typescript. It was used for logical grouping of the classes, interfaces, functions, variables into a single unit and can be exported in another module. The modules are named as a namespace in the latest version of TypeScript. So today, internal modules are obsolete. But they are still supported by using namespace over internal modules.

Syntax:

```
namespace Sum {  
    export function add(a, b) {  
        console.log("Sum: " +(a+b));  
    }  
}
```

External Modules

External modules are also known as a module. When the applications consisting of hundreds of files, then it is almost impossible to handle these files without a modular approach. External Module is used to specify the load dependencies between the multiple external js files. If the application has only one js file, the external module is not relevant.

Syntax:

```
//FileName : EmployeeInterface.ts  
export interface Employee {  
    //code declarations  
}
```

Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a program using nested namespace file storecalc

Nested NameSpace file - StoreCalc

```
namespace invoiceCalc {  
    export namespace invoiceAccount {
```

```
export class Invoice {  
  public calculateDiscount(price: number) {  
    return price * .60;  
  }  
}  
}
```

2. Create a program using a nested namespace file storedigits

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- Namespace encapsulates the features and objects that share common relationships
- The export keyword makes each component accessible to outside the namespaces
- A namespace also allows us to define one namespace into another namespace
- A module is a way to create a group of related variables, functions, classes, and interfaces, etc

Testing your skills

1. ____ encapsulates the features and objects that share common relationships
a) Abstract classes, b) Namespaces, c) Access Modifiers, d) Modules
2. Which of the following are defined within the curly braces {}
a) variables, b) functions, c) classes d) All of the above
3. _____ keyword makes each component accessible to outside the namespaces.
a) export, b) import, c) access, d) None of the above
4. If the namespace is in separate TypeScript file, then it must be referenced by using ____
a) //#, b) ///, c) </>, d) #
5. Accessing the members of the nested namespace can be done through ____ operator
a) dot, b) colon, c) hash, d) none of the above

Lesson 5: Generics

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">• Generic functions• Classes, types, and variables• Constraints	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Introduction

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, that is, being able to create 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.

Generic functions

To start off, let's do the "hello world" of generics: the identity function. The identity function is a function that will return back whatever is passed in. You can think of this in a similar way to the echo command.

Without generics, we would either have to give the identity function a specific type:

```
function identity(arg: number): number {  
    return arg;  
}
```

Or, we could describe the identity function using the any type:

```
function identity(arg: any): any {  
    return arg;  
}
```

While using `any` is certainly generic in that it will cause the function to accept any and all types for the type of `arg`, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a type variable, a special kind of variable that works on types rather than values.

```
function identity<T>(arg: T): T {  
  return arg;  
}
```

We've now added a type variable `T` to the identity 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. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the identity function is generic, as it works over a range of types. Unlike using `any`, it's also just as precise (ie, it doesn't lose any information) as the first identity function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
let output = identity<string>("myString"); // type of output will be 'string'
```

Here we explicitly set `T` to be `string` as one of the arguments to the function call, denoted using the `<>` around the arguments rather than `()`.

The second way is also perhaps the most common. Here we use type argument inference – that is, we want the compiler to set the value of `T` for us automatically based on the type of the argument we pass in:

```
let output = identity("myString"); // type of output will be 'string'
```

Notice that we didn't have to explicitly pass the type in the angle brackets `<>`; the compiler just looked at the value `"myString"`, and set `T` to its type. While type argument inference can be a helpful tool to keep

code shorter and more readable, you may need to explicitly pass in the type arguments as we did in the previous example when the compiler fails to infer the type, as may happen in more complex examples.

Working with Generic Type Variables

When you begin to use generics, you'll notice that when you create generic functions like `identity`, the compiler will enforce that you use any generically typed parameters in the body of the function correctly. That is, that you actually treat these parameters as if they could be any and all types.

Let's take our `identity` function from earlier:

```
function identity<T>(arg: T): T {  
  return arg;  
}
```

What if we want to also log the length of the argument `arg` to the console with each call? We might be tempted to write this:

```
function loggingIdentity<T>(arg: T): T {  
  console.log(arg.length); // Error: T doesn't have .length  
  return arg;  
}
```

When we do, the compiler will give us an error that we're using the `.length` member of `arg`, but nowhere have we said that `arg` has this member. Remember, we said earlier that these type variables stand in for any and all types, so someone using this function could have passed in a number instead, which does not have a `.length` member.

Let's say that we've actually intended this function to work on arrays of `T` rather than `T` directly. Since we're working with arrays, the `.length` member should be available. We can describe this just like we would create arrays of other types:

```
function loggingIdentity<T>(arg: T[]): T[] {  
  console.log(arg.length); // Array has a .length, so no more error  
  return arg;  
}
```

You can read the type of `loggingIdentity` as “the generic function `loggingIdentity` takes a type parameter `T`, and an argument `arg` which is an array of `T`s, and returns an array of `T`s.” If we passed in an array of numbers, we’d get an array of numbers back out, as `T` would bind to `number`. This allows us to use our generic type variable `T` as part of the types we’re working with, rather than the whole type, giving us greater flexibility.

We can alternatively write the sample example this way:

```
function loggingIdentity<T>(arg: Array<T>): Array<T> {  
  console.log(arg.length); // Array has a .length, so no more error  
  return arg;  
}
```

You may already be familiar with this style of type from other languages. In the next section, we’ll cover how you can create your own generic types like `Array<T>`.

Generic Types

In previous sections, we created generic identity functions that worked over a range of types. In this section, we’ll explore the type of the functions themselves and how to create generic interfaces.

The type of generic functions is just like those of non-generic functions, with the type parameters listed first, similarly to function declarations:

```
function identity<T>(arg: T): T {  
  return arg;  
}  
  
let myIdentity: <T>(arg: T) => T = identity;
```

We could also have used a different name for the generic type parameter in the type, so long as the number of type variables and how the type variables are used line up.

```
function identity<T>(arg: T): T {  
  return arg;  
}  
  
let myIdentity: <U>(arg: U) => U = identity;
```

We can also write the generic type as a call signature of an object literal type:

```
function identity<T>(arg: T): T {  
  return arg;  
}
```

```
let myIdentity: {<T>(arg: T): T} = identity;
```

Which leads us to writing our first generic interface. Let's take the object literal from the previous example and move it to an interface:

```
interface GenericIdentityFn {  
  <T>(arg: T): T;  
}
```

```
function identity<T>(arg: T): T {  
  return arg;  
}
```

```
let myIdentity: GenericIdentityFn = identity;
```

In a similar example, we may want to move the generic parameter to be a parameter of the whole interface. This lets us see what type(s) we're generic over (e.g. Dictionary<string> rather than just Dictionary). This makes the type parameter visible to all the other members of the interface.

```
interface GenericIdentityFn<T> {  
  (arg: T): T;  
}
```

```
function identity<T>(arg: T): T {  
  return arg;  
}
```

```
let myIdentity: GenericIdentityFn<number> = identity;
```

Notice that our example has changed to be something slightly different. Instead of describing a generic function, we now have a non-generic function signature that is a part of a generic type. When we use `GenericIdentityFn`, we now will also need to specify the corresponding type argument (here: `number`), effectively locking in what the underlying call signature will use.

Understanding when to put the type parameter directly on the call signature and when to put it on the interface itself will be helpful in describing what aspects of a type are generic.

In addition to generic interfaces, we can also create generic classes. Note that it is not possible to create generic enums and namespaces.

Generic Classes

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (`<>`) following the name of the class.

```
class GenericNumber<T> {  
  zeroValue: T;  
  add: (x: T, y: T) => T;  
}
```

```
let myGenericNumber = new GenericNumber<number>();  
myGenericNumber.zeroValue = 0;  
myGenericNumber.add = function(x, y) { return x + y; };
```

This is a pretty literal use of the `GenericNumber` class, but you may have noticed that nothing is restricting it to only use the `number` type. We could have instead used `string` or even more complex objects.

```
let stringNumeric = new GenericNumber<string>();  
stringNumeric.zeroValue = "";  
stringNumeric.add = function(x, y) { return x + y; };  
  
console.log(stringNumeric.add(stringNumeric.zeroValue, "test"));
```


Just as with interface, putting the type parameter on the class itself lets us make sure all of the properties of the class are working with the same type.

As we covered in our section on classes, a class has two sides to its type: the static side and the instance side. Generic classes are only generic over their instance side rather than their static side, so when working with classes, static members can not use the class's type parameter.

Generic Constraints

If you remember from an earlier example, you may sometimes want to write a generic function that works on a set of types where you have some knowledge about what capabilities that set of types will have. In our `loggingIdentity` example, we wanted to be able to access the `.length` property of `arg`, but the compiler could not prove that every type had a `.length` property, so it warns us that we can't make this assumption.

```
function loggingIdentity<T>(arg: T): T {  
  console.log(arg.length); // Error: T doesn't have .length  
  return arg;  
}
```

Instead of working with any and all types, we'd like to constrain this function to work with any and all types that also have the `.length` property. As long as the type has this member, we'll allow it, but it's required to have at least this member. To do so, we must list our requirement as a constraint on what `T` can be.

To do so, we'll create an interface that describes our constraint. Here, we'll create an interface that has a single `.length` property and then we'll use this interface and the `extends` keyword to denote our constraint:

```
interface Lengthwise {  
  length: number;  
}  
  
function loggingIdentity<T extends Lengthwise>(arg: T): T {  
  console.log(arg.length); // Now we know it has a .length property, so no more error  
  return arg;  
}
```

Because the generic function is now constrained, it will no longer work over any and all types:

```
loggingIdentity(3); // Error, number doesn't have a .length property
```

Instead, we need to pass in values whose type has all the required properties:

```
loggingIdentity({length: 10, value: 3});
```

Using Type Parameters in Generic Constraints

You can declare a type parameter that is constrained by another type parameter. For example, here we'd like to get a property from an object given its name. We'd like to ensure that we're not accidentally grabbing a property that does not exist on the obj, so we'll place a constraint between the two types:

```
function getProperty<T, K extends keyof T>(obj: T, key: K) {  
  return obj[key];  
}
```

```
let x = { a: 1, b: 2, c: 3, d: 4 };
```

```
getProperty(x, "a"); // okay
```

```
getProperty(x, "m"); // error: Argument of type 'm' isn't assignable to 'a' | 'b' | 'c' | 'd'.
```

Using Class Types in Generics

When creating factories in TypeScript using generics, it is necessary to refer to class types by their constructor functions. For example,

```
function create<T>(c: {new(): T; }): T {  
  return new c();  
}
```

Let's have a look at a more advanced example that uses the prototype property to infer and constrain relationships between the constructor function and the instance side of class types:

```

class BeeKeeper {
    hasMask: boolean;
}

class ZooKeeper {
    nametag: string;
}

class Animal {
    numLegs: number;
}

class Bee extends Animal {
    keeper: BeeKeeper;
}

class Lion extends Animal {
    keeper: ZooKeeper;
}

function createInstance<A extends Animal>(c: new () => A): A {
    return new c();
}

createInstance(Lion).keeper.nametag; // typechecks!
createInstance(Bee).keeper.hasMask;  // typechecks!

```

Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a generic display function with constraints.

```

class Person {
    firstName: string;
    lastName: string;
    constructor(fname:string, lname:string) {
        this.firstName = fname;
        this.lastName = lname;
    }
}

function display<T extends Person>(per: T): void {
    console.log(` ${ per.firstName} ${per.lastName}` );
}

var per = new Person("Bill", "Gates");
display(per); //Output: Bill Gates
display("Bill Gates");//Compiler Error

```

2. Create a generic display function that will display the birth year of a person.

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, that is, being able to create a component that can work over a variety of types rather than a single one

When you begin to use generics, you'll notice that when you create generic functions like identity, the compiler will enforce that you use any generically typed parameters in the body of the function correctly

A generic class has a similar shape to a generic interface. Generic classes have a generic type parameter list in angle brackets (<>) following the name of the class.

Testing your skills

1. What type of function is Generics

a) Image, b) Identity, c) links, d) none of the above

2. Generic type is similar to _____ declaration

a) function, b) array, c) module, d) none of the above

3. Generic classes are only generic over their _____ side rather than their _____ side

a) instance, static, b) static, instance, c) boolean, static, d) static, generic

4. You can declare a type parameter that is constrained by another ____ parameter

a) function, b) alt, c) type, d) none of the above

5) Factories can be created in TypeScript through the use of ____ functions

a) Assembly, b) constructor, c) <href>, <none of the above>

Lesson 6: Decorators

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">• The basics of decorators• Method decorator, class decorator, decorator factories	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

Decorators are one the most powerful features Typescript has to offer, allowing us to extend the functionality of classes and methods in a clean and declarative fashion. Decorators are currently a stage 2 proposal for JavaScript but have already gained popularity in the TypeScript eco-system, being used by leading open-source projects such as Angular and Inversify.

Decorators are just a clean syntax for wrapping a piece of code with a function. To allow the use of decorators add "experimentalDecorators": true to your compiler options in tsconfig.json file, and make sure your transpilation target is ES5 or later.

Understanding Class Decorators

Say you have a business that rents old castles to powerful families, and you're working on setting up an HTTP server. You decided to build each endpoint of your API as a class, and the public methods of the class would correspond to the HTTP methods. This might look something like that:

```
class Families {
  private houses = ["Lannister", "Targaryen"];

  get() {
    return this.houses;
  }
  post(request) {
    this.houses.push(request.body);
  }
}

class Castles {
  private castles = ["Winterfell", "Casterly Rock"];

  get() {
    return this.castles;
  }
  post(request) {
    this.castles.push(request.body);
  }
}
```

Now we need a simple way to “register” each of these classes as an endpoint in our HTTP server. Let’s create a simple function to take care of that. Our function will get a class as an argument, and add an instance of that class as an endpoint to our server. Like so:

```
const httpEndpoints = {};  
  
function registerEndpoint(constructor) {  
  const className = constructor.name;  
  const endpointPath = "/" + className.toLowerCase();  
  httpEndpoints[endpointPath] = new constructor();  
}  
  
registerEndpoint(Families)  
registerEndpoint(Castles)  
  
console.log(httpEndpoints) // {"/families": Families, "/castles": Castles}  
httpEndpoints["/families"].get() // ["Lannister", "Targaryen"]
```

Well congratulations! You have just written your first decorator!

It’s just this much simple. A decorator is basically a function that takes a class as an argument. And that’s what you created.

Now instead of calling registerEndpoint in the “regular” way, we can just decorate our classes with @registerEndpoint. Just have a look:

```
const httpEndpoints = {};  
  
function registerEndpoint(constructor) {  
  const className = constructor.name;  
  const endpointPath = "/" + className.toLowerCase();  
  httpEndpoints[endpointPath] = new constructor();  
}  
  
@registerEndpoint  
class Families {  
  // implementation...  
}  
  
@registerEndpoint  
class Castles {  
  // implementation...  
}  
  
console.log(httpEndpoints) // {"/families": Families, "/castles": Castles}  
httpEndpoints["/families"].get() // ["Lannister", "Targaryen"]
```

It just took a couple of minutes to create the decorator and that too, in its running state. Now, let’s look at method decorator.

Understanding Method Decorator

Let's say that we want to protect some of our endpoints so that only authenticated users will be able to access them. To do that we can create a new decorator called protect.

For now, all our decorator will do is to add the protected method to an array called protectedMethods.

```
const protectedMethods = [];  
  
function protect(target, propertyKey, descriptor) {  
  const className = target.constructor.name;  
  protectedMethods.push(className + "." + propertyKey);  
}  
  
@registerEndpoint  
class Families {  
  private houses = ["Lannister", "Targaryen"];  
  
  @protect  
  get() {  
    return this.houses;  
  }  
  
  @protect  
  post(request) {  
    this.houses.push(request.body);  
  }  
}  
  
console.log(protectedMethods) // ["Families.get", "Families.post"]
```

As you can see, the method decorator takes 3 arguments:

- target — The prototype of our class (or the constructor of the class if the decorated method is static).
- propertyKey — The name of the decorated method.
- descriptor — An object that holds the decorated function and some meta-data regarding it.

So far we only read information regarding the classes and methods we decorated, but the real fun begins when we start changing their behavior. We can do that by simply returning a value from the decorator. When a method decorator returns a value, this value will be used instead of the original descriptor (which holds the original method).

Let's try it out by creating a decorator called nope, that replaces our original method with a method that prints "nope" whenever it is called. To do that I'll override descriptor.value, which is where the original function is stored, with my function:

```
function nope(target, propertyKey, descriptor) {
  descriptor.value = function() {
    console.log("nope");
  };
  return descriptor;
}

@registerEndpoint
class Families {
  private houses = ["Lannister", "Targaryen"];

  @nope
  get() {
    return this.houses;
  }
}

httpEndpoints[("/families").get()] // nope
```

By this point, we played around with the basics of method decorators, but haven't created anything useful yet. What I plan to do next is rewrite the protect decorator, but this time instead of only logging the names of the decorated methods, it will actually block unauthorized requests.

This next step will be a bit more complicated from the last ones so bear with me. Here are the steps we should take:

- Extract the original function from the descriptor, and store it somewhere else for later use.
- Override descriptor.value with a new function that takes the same arguments as the original.
- Within our new function, check if the request is authenticated, and if not throw an error.
- Finally, also within our new function, we can now call the original function with the arguments it needs, capture its return value, and return it.

Amazing! We have a fully operational protect decorator. Now adding or removing protection from our methods is a piece of cake.

Note how in line 8 we 'bind' the original function to 'this', so it will have access to the instance of its class. For example, without this line, our get() method wouldn't be able to read this.houses.

Learning about Decorator Factories

Let's say we now wish to add a new endpoint to return the Stark family members, at the path /families/stark/members. Well, obviously we can't create a class with that name, so what are we going to do?

What we need here is a way to pass parameters that will dictate the behavior of our decorator function. Let's take another look at our good old registerEndpoint decorator:

```
const httpEndpoints = {};  
  
function registerEndpoint(constructor) {  
  const className = constructor.name;  
  const endpointPath = "/" + className.toLowerCase();  
  httpEndpoints[endpointPath] = new constructor();  
}  
  
@registerEndpoint  
class Families {  
  // implementation...  
}  
  
@registerEndpoint  
class Castles {  
  // implementation...  
}  
  
console.log(httpEndpoints) // {"/families": Families, "/castles": Castles}  
httpEndpoints["/families"].get() // ["Lannister", "Targaryen"]
```

Now this is important - to send parameters to our decorator, we need to transform it from a regular decorator to a decorator factory. A decorator factory is a function that returns a decorator. To make one, all we need is to wrap our original decorator, like this:

```
const httpEndpoints = {};  
  
function registerEndpointFactory(endpointPath) {  
  return function registerEndpoint(constructor) {  
    httpEndpoints[endpointPath] = new constructor();  
  }  
}  
  
@registerEndpointFactory("/families/stark/members")  
class StarkMembers {  
  private members = ["Robb", "Sansa", "Arya"];  
  
  get() {  
    return this.members;  
  }  
  
  @protect  
  post(request) {  
    this.members.push(request.body);  
  }  
}  
  
console.log(httpEndpoints) // {"/families/stark/members": StarkMembers}  
httpEndpoints["/families/stark/members"].get() // ["Robb", "Sansa", "Arya"]
```

Now we can also wrap our protect decorator, so it will get the expected token as a parameter:

```
function protect(token) {
  return function(target, propertyKey, descriptor) {
    const originalFunction = descriptor.value;

    descriptor.value = function(request) {
      if (request.token !== token) {
        throw new Error("Forbidden!");
      }
      const bindedOriginalFunction = originalFunction.bind(this);
      const result = bindedOriginalFunction(request);
      return result;
    };

    return descriptor;
  };
}

class StarkMembers {
  private members = ["Robb", "Sansa", "Arya"];

  @protect("abc")
  post(request) {
    this.members.push(request.body);
  }
}
```

Decorators in Typescript can be applied not only to classes and methods, but also to properties of classes, and method arguments.

Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a class decorator Frozen

@Frozen

class IceCream {}

function Frozen(constructor: Function) {

Object.freeze(constructor);

Object.freeze(constructor.prototype);

}

console.log(Object.isFrozen(IceCream)); // true

class FroYo extends IceCream {} // error, cannot be extended

2. Create a class decorator Wealth

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- Decorators are just a clean syntax for wrapping a piece of code with a function.
- The method decorator takes 3 arguments: target, propertykey, and descriptor
- Decorators in Typescript can be applied not only to classes and methods, but also to properties of classes, and method arguments.

Testing your skills

1. Decorators are a Stage__ proposal for JavaScript
a) Stage1, b) Stage2 c) Stage3, d) Stage4

2. _____ specifies the name of the decorated method
a) Descriptor, b) target, c) propertykey, d) none of the above

3. An object that holds the decorated function and some meta-data regarding it is known as
a) Descriptor, b) target, c) propertykey, d) none of the above

4. Which of the following arguments are parts of the method decorator
a) Descriptor, b) target, c) propertykey, d) all of the above

5. A _____ is basically a function that takes a class as an argument
a) Descriptor, b) Decorator, c) propertykey, d) constructor

Lesson 7: Typescript Essentials

Objective: After completing this lesson you will be able to learn about : <ul style="list-style-type: none">• tsc and tsconfig file• debugging typescript	Materials Required: <ul style="list-style-type: none">• Computer With Windows XP and above• Stable Internet connection
Self- Learning Duration: 60 minutes	Practical Duration: 60 minutes
Total Duration: 120 minutes	

The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project. A project is compiled in one of the following ways:

Using tsconfig.json

By invoking tsc with no input files, in which case the compiler searches for the tsconfig.json file starting in the current directory and continuing up the parent directory chain.

By invoking tsc with no input files and a --project (or just -p) command line option that specifies the path of a directory containing a tsconfig.json file, or a path to a valid .json file containing the configurations.

When input files are specified on the command line, tsconfig.json files are ignored.

Let's look at an example using the "files" property:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "noImplicitAny": true,
    "removeComments": true,
    "preserveConstEnums": true,
    "sourceMap": true
  },
  "files": [
    "core.ts",
    "sys.ts",
    "types.ts",
    "scanner.ts",
    "parser.ts",
    "utilities.ts",
    "binder.ts",
    "checker.ts",
    "emitter.ts",
    "program.ts",
    "commandLineParser.ts",
    "tsc.ts",
    "diagnosticInformationMap.generated.ts"
  ]
}
```

Debugging Typescript

TypeScript, being the superset of JavaScript, makes it very convenient to write applications that have support for things like data types and generics to mention a few. However in order to be able to execute TypeScript code we need to transpile it to JavaScript. Having this JavaScript code also means that, when looking at our application code in the browser, we won't be able to see the TypeScript source code.

There are of course ways that we can do to overcome this and in this article we are going to be discussing how to debug TypeScript in web browsers by using source maps.

Source maps

Source maps are special files that get created explicitly for JavaScript or CSS files. The idea is simple - imagine a bunch of JavaScript files or even SASS generated CSS files that get minified and concatenated in one single file. There's no way that we can debug using these files in the browser.

Having a source map will allow us to map our minified (or otherwise modified files) to their sources and be able to view the source in the browser. On top of being able to see the original source we can also add breakpoints easily and further debug our application.

Generate source maps

Now, in order to generate a source map, we'll be using the tsc transpiler with the `--sourceMaps true` option. Let's take a look at a very simple example and write some TypeScript code first:

```
interface IWarrior {  
  name: string;  
  health: number;  
  fight: Function;  
}  
  
let myWarrior: IWarrior = {  
  name: 'John the Nomad',  
  health: 100,  
  fight: function() {  
    return `${this.name} attack!`;  
  }  
};  
  
console.log(  
  myWarrior.fight()  
);
```

Now let's go ahead and transpile our code by executing `tsc --sourceMaps true`.

The result is going to be a JavaScript file with the following line appended at the end:

```
//# sourceMappingURL=app.js.map
```

And we also have a map file generated with the following content:

```
{  
  "version": 3,  
  "file": "app.js",  
  "sourceRoot": "",  
  "sources": ["app.ts"],  
  "names": [],  
  "mappings": "AAMA,IAAI,SAAS,GAaA;IACxB,IAAI,EAAE,gBAAGB;IACtB,MAAM,EAAE,GAAG  
}
```

Understanding the source map

The source map is really just a JSON document that has a few properties:

- version - indicating the source map standard's version
- file - the transpiled filename
- sourceRoot - An optional source root, useful for relocating source files on a server or removing repeated values in the "sources" entry
- sources: the original TypeScript source file

- names: a list of symbol names used by the following, mappings entry
- mappings: encoded mapping data

So we have the generated filename, the source and the mapping between the two that uses Base 64 VLQ (Variable-Length Quality) encoding. This encoding stores letters against numbers: for example 0 - A, 1 - C so on and so forth.

Practical Session

Instructions: Create the code....using an editor....and save it with HTML extension.

1. Create a tsconfig.json file using the FILES property

```
{  
  "compilerOptions": {  
    "module": "commonjs",  
    "noImplicitAny": true,  
    "removeComments": true,  
    "preserveConstEnums": true,  
    "sourceMap": true  
  },  
  "files": [  
    "core.ts",  
    "sys.ts",  
    "types.ts",  
    "scanner.ts",  
    "parser.ts",  
    "utilities.ts",  
    "binder.ts",  
    "checker.ts",  
    "emitter.ts",  
    "program.ts",  
    "commandLineParser.ts",  
    "tsc.ts",  
    "diagnosticInformationMap.generated.ts"  
  ]  
}
```

2. Create a tsconfig.json file using the OPTIONS property

Note: After saving the code, click the created HTML file to view the output.

Reviewing the chapter

- The presence of a tsconfig.json file in a directory indicates that the directory is the root of a TypeScript project.
- The tsconfig.json file specifies the root files and the compiler options required to compile the project
- Source maps are special files that get created explicitly for JavaScript or CSS files

Testing your skills

1. Which of the following are properties of source map

a) names, b) mapping, c) file, d) all of the above

2. _____ indicating the source map standard's version

a) version, b) sourceroot, c) sources, d) file

3. _____ are special files that get created explicitly for JavaScript or CSS files

a) Source route, b) Source Map, c) file selector dialog boxes, d) all of the above

4. To generate source maps, we have to use _____ transpiler

a) tsc, b) cgi, c) tcs, d) none of the above

5. a list of symbol names used by the following, mappings entry

a) mapping b) JSON, c) names, d) sourceroot