

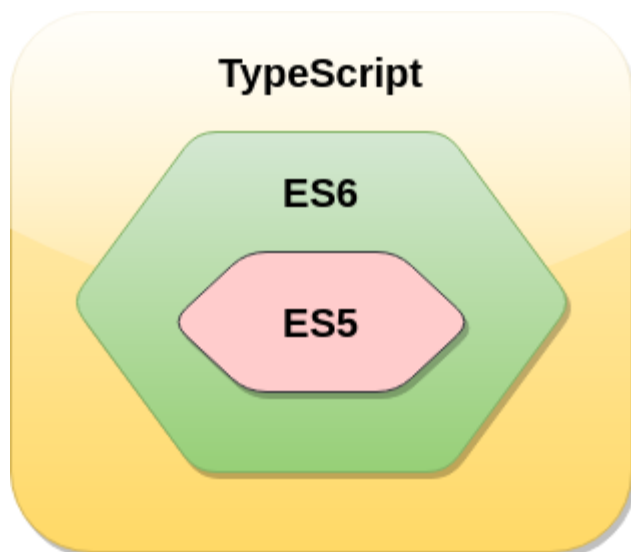
Unit-4

Type Script Introduction:

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. 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**. 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.

What is TypeScript?

TypeScript is an open-source pure object-oriented programming language. It is a strongly typed superset of JavaScript which compiles to plain JavaScript. It contains all elements of the JavaScript. It is a language designed for large-scale JavaScript application development, which can be executed on any browser, any Host, and any Operating System. The TypeScript is a language as well as a set of tools. TypeScript is the ES6 version of JavaScript with some additional features.



TypeScript cannot run directly on the browser. It needs a compiler to compile the file and generate it in JavaScript file, which can run directly on the browser. The TypeScript source file is in ".ts" extension. We can use any valid ".js" file by renaming it to ".ts" file. TypeScript uses TSC (TypeScript Compiler) compiler, which convert Typescript code (.ts file) to JavaScript (.js file).



History of TypeScript

In 2010, Anders Hejlsberg, a core member of the development team of C# language, started working on TypeScript at Microsoft. The first version of TypeScript was released to the public in the month of 1st October 2012 and was labeled as version 0.8. Now, it is maintained by Microsoft under the Apache 2 license. The latest version of Typescript is TypeScript 3.5, which was released to the public on May 2019.

Why use TypeScript?

We use TypeScript because of the following 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 has a definition file with .d.ts extension to provide a definition for external JavaScript libraries.
- TypeScript gives all the benefits of ES6 plus more productivity.
- Developers can save a lot of time with TypeScript.

Text Editors with TypeScript Support

The TypeScript was initially supported only in Microsoft's Visual Studio platform. But today, there are a lot of text editors and IDEs available which either natively or through plugins have support for the TypeScript programming. Some of them are given below.

1. Visual Studio Code
2. Official Free Plugin for Sublime Text.
3. The latest version of WebStorm
4. It also supports in Vim, Atom, Emacs, and others.

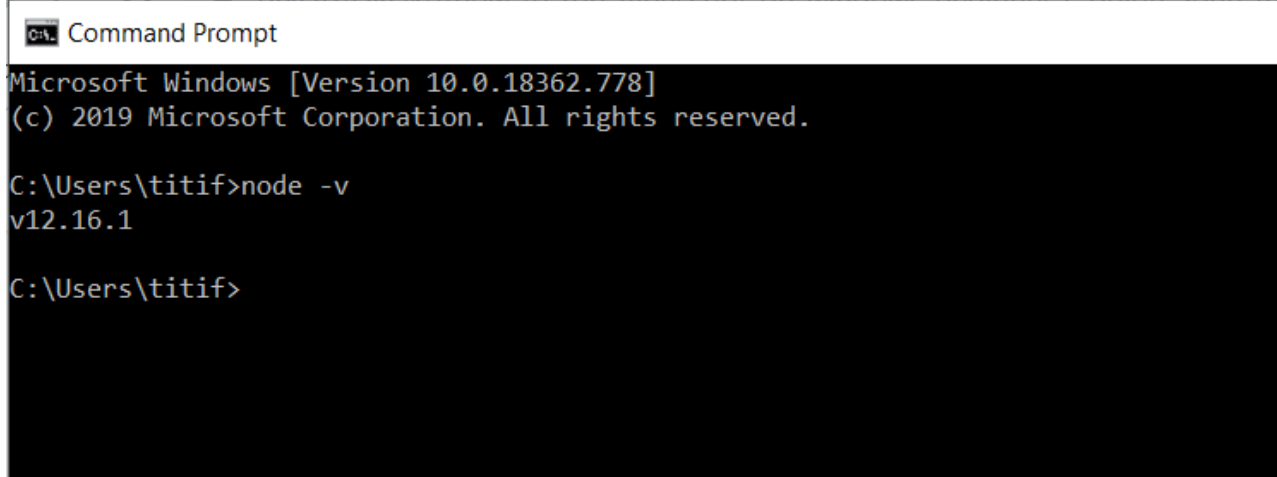
How to install and run Typescript on Windows - beginner's guide

But how exactly can we use it? Typescript doesn't run in the browser. It needs to be compiled into regular Javascript and it lives inside files with a .ts extension. If you want to first get comfortable with the syntax and avoid installation, you can use it online, inside a playground that can be found here: <https://www.typescriptlang.org/play/>. The way it works is that you write Typescript code on the left side and see the code it compiles to on the right side (this works on any OS, it's not Windows specific).

If you want to have it on your machine and use it anytime you wish, you'll need to install it. The way I personally use it is with Visual Studio Code, as an extension, but when I first started learning the syntax, I had it installed via npm (Node's package manager).

Steps to install Typescript on a Windows machine using npm:

1. Install Node.js. Unless you need to install a different version (like for example if you're doing a tutorial and the tutor tells you to install a specific version), I would advise to install the latest. You can find it here: <https://nodejs.org/en/>. Use the LTS (long time support) variant. If you want to check if node has been installed successfully, go open the cmd and type `node -v`. If everything is ok, you should see something like `v12.16.1` (this is the current version I'm using).

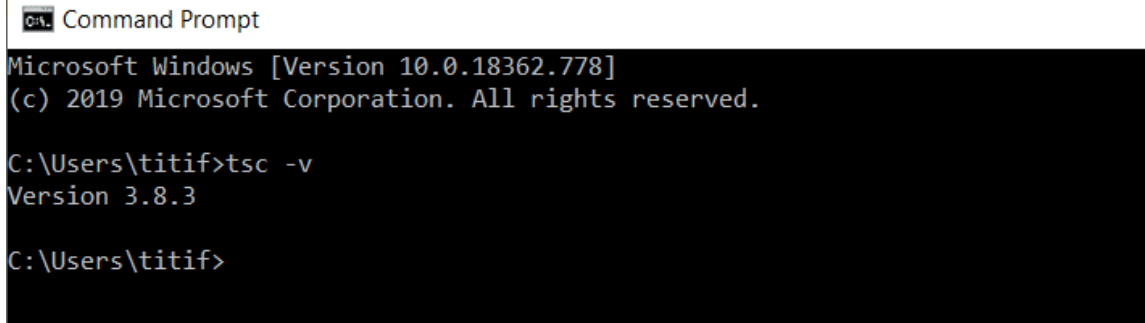
A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The window content shows the following text: "Microsoft Windows [Version 10.0.18362.778]", "(c) 2019 Microsoft Corporation. All rights reserved.", "C:\Users\titif>node -v", "v12.16.1", and "C:\Users\titif>".

```
Command Prompt
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\titif>node -v
v12.16.1

C:\Users\titif>
```

2. Install Typescript. Run the following command in the cmd.
`npm install -g typescript`. This will install Typescript globally. In the same manner, if you want to check for Typescript being installed, type `tsc -v` in the cmd and you should get back something like Version 3.8.3 (it's possible that you'll be using a different version so you'll get different numbers).

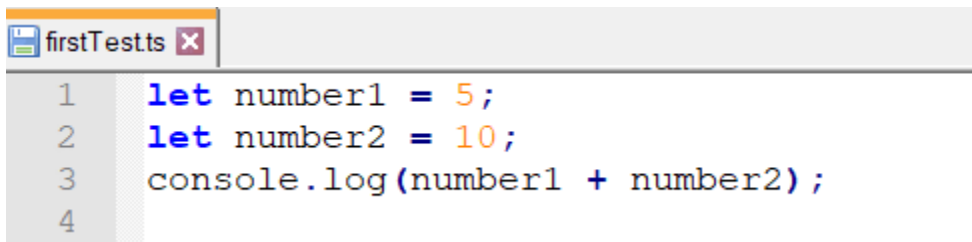


```
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\titit>tsc -v
Version 3.8.3

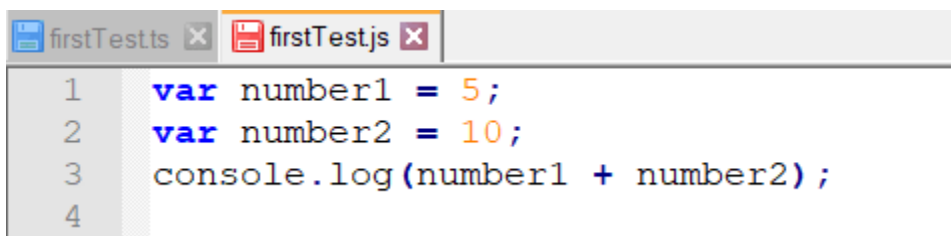
C:\Users\titit>
```

3. Create a file with a .ts extension. On your machine, create a file with a .ts extension. I called mine firstTest.ts. Open it in your editor (for simplicity, I'm using Notepad++) and write any Javascript you want inside it. I wrote:



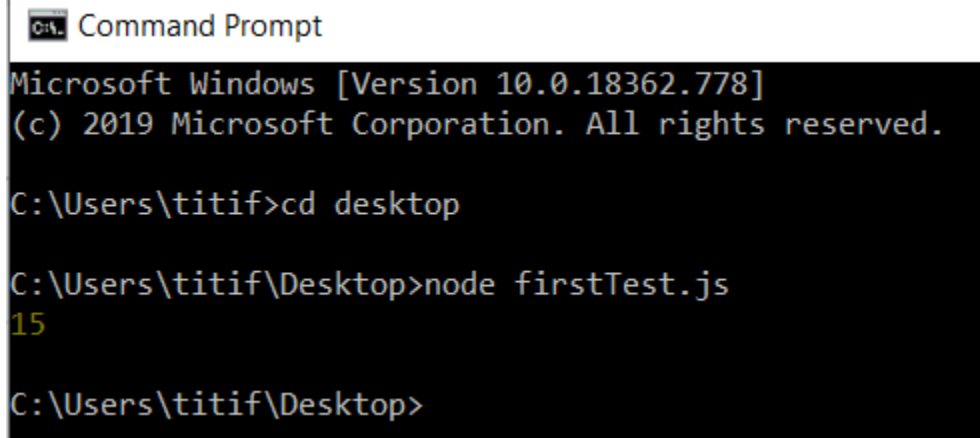
```
1 let number1 = 5;
2 let number2 = 10;
3 console.log(number1 + number2);
4
```

4. Create a .js file out of your .ts one. Remember, Typescript can't be used like Javascript. It first needs to be compiled (translated) to regular Js. In the cmd, navigate inside the folder that contains your .ts file and run the following command `tsc filename.ts`. This will create, in the same folder, a new file with the same name but a .js extension. This is the step in which the "translation" takes place. What the newly created js file contains is basically all the Typescript we wrote, only compiled to Javascript. In my case, the Javascript code it's almost identical, because I didn't really use any of the Typescript features (notice though, the let ES6 feature was converted to a var).



```
1 var number1 = 5;
2 var number2 = 10;
3 console.log(number1 + number2);
4
```

5. Run your Javascript code using Node. To see the result of your code, we can run it using Node. So, inside the cmd write `node filename.ts`. In my case it will be `node firstTest.js`. The result I got was 5.



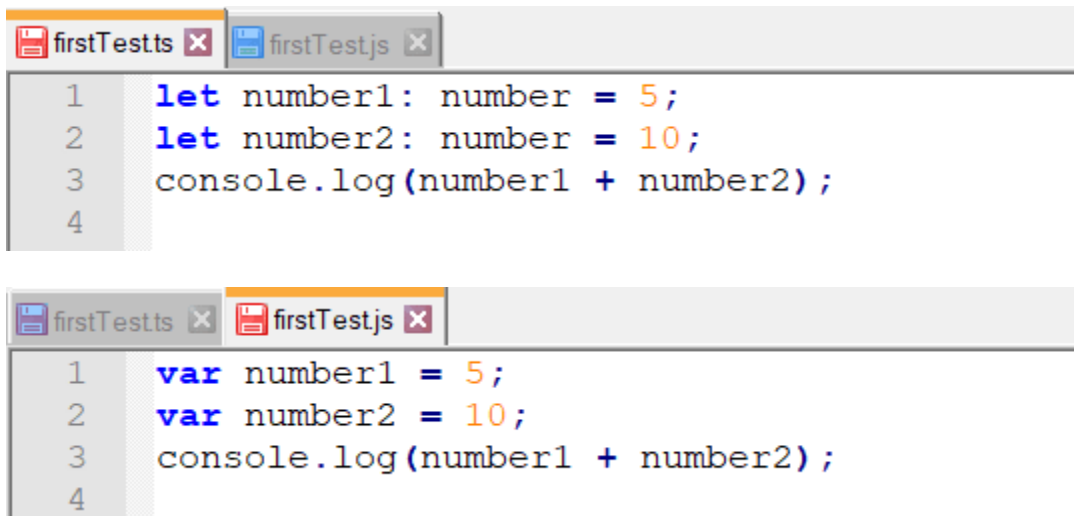
```
Command Prompt
Microsoft Windows [Version 10.0.18362.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\titif>cd desktop

C:\Users\titif\Desktop>node firstTest.js
15

C:\Users\titif\Desktop>
```

Now let's use some very simple Typescript code. We're going to explicitly set our two variables to be of type number. Ignore the syntax, the example is strictly to show you how the Javascript code looks like in the end.



The first screenshot shows a code editor with two tabs: 'firstTest.ts' and 'firstTest.js'. The 'firstTest.ts' tab is active, displaying the following TypeScript code:

```
1 let number1: number = 5;
2 let number2: number = 10;
3 console.log(number1 + number2);
4
```

The second screenshot shows the same code editor with the 'firstTest.js' tab active. The code has been converted to JavaScript, and the TypeScript type annotations have been removed:

```
1 var number1 = 5;
2 var number2 = 10;
3 console.log(number1 + number2);
4
```

Notice in the .js file the types don't show up and the let has been converted to a var.

One thing to keep in mind is that every time you make a change in your .ts file, you must run Typescript so the changes also appear in the .js file. For less typing, you can combine the two commands like so `tsc filename.ts && node filename.js` (the extensions' names can be omitted, and our code will look like `tsc filename && node filename`).

What is the Function type in TypeScript ?

TypeScript is a JavaScript-based programming language with a typed syntax. It provides improved tools of any size. It adds extra syntax to JavaScript. This helps in facilitating a stronger interaction between you and your editor. It also helps in catching the mistakes well in advance.

It uses type inference to provide powerful tools without the need for extra code. TypeScript may be executed everywhere JavaScript is supported as it can be converted to JavaScript code.

TypeScript Functions: Functions are the most crucial aspect of JavaScript as it is a functional programming language. Functions are pieces of code that execute specified tasks. They are used to implement object-oriented programming principles like classes, objects, polymorphism, and abstraction. It is used to ensure the reusability and maintainability of the program. Although TypeScript has the idea of classes and modules, functions are still an important aspect of the language.

Function declaration: The name, parameters, and return type of a function are all specified in a function declaration. The function declaration has the following:

Syntax:

```
function functionName(arg1, arg2, ... , argN);
```

Function definition: It includes the actual statements that will be executed. It outlines what should be done and how it should be done. The function definition has the following

Syntax:

```
function functionName(arg1, arg2, ... , argN){  
  // Actual code for execution  
}
```

Function call: A function can be called from anywhere in the application. In both function calling and function definition, the parameter/argument must be the same. We must pass the same number of parameters that the function definition specifies. The function call has the following

Syntax:

```
functionName(arg1, arg2, ... , argM);
```

Function Types:

Types of Functions in TypeScript: There are two types of functions in TypeScript:

- Named Function
- Anonymous Function

1. Named function: A named function is defined as one that is declared and called by its given name. They may include parameters and have return types.

Syntax:

```
functionName( [args] ) { }
```

Example:

- Javascript

```
// Named Function Definition
```

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

```
// Function Call
```

```
myFunction(7, 5);
```

Output:

12

2. Anonymous Function: An anonymous function is a function without a name. At runtime, these kinds of functions are dynamically defined as an expression. We may save it in a variable and eliminate the requirement for function names.

Syntax:

```
let result = function( [args] ) { }
```

Example:

- Javascript

```
// Anonymous Function
```

```
let myFunction = function (a: number, b: number) : number {
```

```
    return a + b;  
  
};  
  
// Anonymous Function Call  
  
console.log(myFuction(7, 5));
```

Output:

12

Advantage of function: Benefits of functions may include but are not limited to the following:

- **Code Reusability:** We can call a function several times without having to rewrite the same code block. The reusability of code saves time and decreases the size of the program.
- **Less coding:** Our software is more concise because of the functions. As a result, we don't need to write a large number of lines of code each time we execute a routine activity.
- **Easy to debug:** It makes it simple for the programmer to discover and isolate incorrect data.

Return Type

The type of the value returned by the function can be explicitly defined.

/ the `: number` here specifies that this function returns a number

```
function getTime(): number {  
    return new Date().getTime();  
}
```

```
console.log(getTime());
```

O/P:

1648464745471

Void Return Type

The type **void** can be used to indicate a function doesn't return any value.


```
function printHello(): void {  
    console.log('Hello!');  
}
```

```
printHello();
```

O/P:

```
Hello!
```

Parameters

Function parameters are typed with a similar syntax as variable declarations.

```
function multiply(a: number, b: number) {  
    return a * b;  
}  
  
console.log(multiply(2,5))
```

```
O/P: 10
```

Optional Parameters

By default TypeScript will assume all parameters are required, but they can be explicitly marked as optional.

// the `?` operator here marks parameter `c` as optional

```
function add(a: number, b: number, c?: number) {  
    return a + b + (c || 0);  
}
```

```
console.log(add(2,5))
```

O/P::7

Default Parameters

For parameters with default values, the default value goes after the type annotation:

Example

```
function pow(value: number, exponent: number = 10) {  
    return value ** exponent;  
}  
  
console.log(pow(10));
```

O/P: 10000000000

Named Parameters

Typing named parameters follows the same pattern as typing normal parameters.

```
function divide({ dividend, divisor }: { dividend: number, divisor: number }) {  
    return dividend / divisor;  
}  
  
console.log(divide({dividend: 10, divisor: 2}));
```

O/P: 5

Rest Parameters

Rest parameters can be typed like normal parameters, but the type must be an array as rest parameters are always arrays.

```
function add(a: number, b: number, ...rest: number[]) {  
    return a + b + rest.reduce((p, c) => p + c, 0);  
}
```

```
console.log(add(10,10,10,10,10));
```

TYPE SCRIPT INTERFACE:

The TypeScript compiler uses interface for **type-checking** (also known as "duck typing" or "structural subtyping") whether the object has a specific structure or not.

The interface contains only the **declaration** of the **methods** and **fields**, but not the **implementation**. We cannot use it to build anything. It is inherited by a class, and the class which implements interface defines all members of the interface.

Interface Declaration

We can declare an interface as below.

```
1. Interface interface_name {  
2.     // variables' declaration  
3.     // methods' declaration  
4. }
```

- An **interface** is a keyword which is used to declare a TypeScript Interface.
- An **interface_name** is the name of the interface.
- An interface body contains variables and methods declarations.

Example

```
1. interface OS {  
2.     name: String;  
3.     language: String;  
4. }  
5. let OperatingSystem = (type: OS): void => {  
6.     console.log('Android ' + type.name + ' has ' + type.language + ' language.');7. };  
8. let Oreo = {name: 'O', language: 'Java'}  
9. OperatingSystem(Oreo);
```

Example

```
1. interface OS {  
2.     name: String;  
3.     language: String;
```

```
4. }  
5. let OperatingSystem = (type: OS): void => {  
6.   console.log('Android ' + type.name + ' has ' + type.language + ' language.');
```

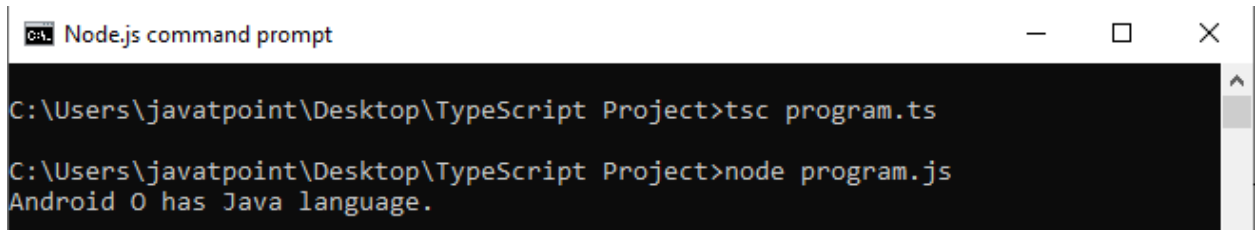
```
7. };  
8. let Oreo = { name: 'O', language: 'Java'}  
9. OperatingSystem(Oreo);
```

In the above example, we have created an interface OS with properties name and language of string type. Next, we have defined a function, with one argument, which is the type of interface OS.

Now, compile the TS file into the JS which looks like the below output.

```
1. let OperatingSystem = (type) => {  
2.   console.log('Android ' + type.name + ' has ' + type.language + ' language.');
```

```
3. };  
4. let Oreo = { name: 'O', language: 'Java' };  
5. OperatingSystem(Oreo);  
6.
```



```
Node.js command prompt  
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts  
C:\Users\javatpoint\Desktop\TypeScript Project>node program.js  
Android O has Java language.
```

TypeScript Classes

In object-oriented programming languages like Java, classes are the fundamental entities which are used to create **reusable** components. It is a group of objects which have common properties. In terms of OOPs, a class is a **template** or **blueprint** for creating objects. It is a logical entity.

A class definition can contain the following properties:

- **Fields:** It is a variable declared in a class.
- **Methods:** It represents an action for the object.
- **Constructors:** It is responsible for initializing the object in memory.

- **Nested class and interface:** It means a class can contain another class.

TypeScript is an Object-Oriented JavaScript language, so it supports object-oriented programming features like classes, interfaces, polymorphism, data-binding, etc. JavaScript **ES5** or **earlier version** did not support classes. TypeScript support this feature from **ES6** and **later version**. TypeScript has **built-in** support for using classes because it is based on ES6 version of JavaScript. Today, many developers use class-based object-oriented programming languages and compile them into JavaScript, which works across all major browsers and platforms.

Syntax to declare a class

A class keyword is used to declare a class in TypeScript. We can create a class with the following syntax:

1. class **<class_name>**{
2. field;
3. method;
4. }

Example

1. class Student {
2. studCode: number;
3. studName: string;
- 4.
5. constructor(code: number, name: string) {
6. **this.studName** = **name**;
7. **this.studCode** = **code**;
8. }
- 9.
10. getGrade() : string {
11. return "A+" ;
12. }
13. }

The TypeScript compiler converts the above class in the following JavaScript code.

1. var **Student** = /** @class */ (function () {
2. function Student(code, name) {
3. **this.studName** = **name**;

```
4.     this.studCode = code;
5.   }
6.   Student.prototype.getGrade = function () {
7.     return "A+";
8.   };
9.   return Student;
10. }());
```

Creating an object of class

A class creates an object by using the **new** keyword followed by the **class name**. The **new** keyword allocates memory for object creation at runtime. All objects get memory in heap memory area. We can create an object as below.

Syntax

1. let **object_name** = **new** class_name(parameter)
 1. **new keyword:** it is used for instantiating the object in memory.
 2. The right side of the expression invokes the constructor, which can pass values.

Example

1. //Creating an object or instance
 2. let **obj** = **new** Student();

)
-

Object Initialization

Object initialization means storing of data into the object. There are three ways to initialize an object. These are:

1. By reference variable

Example

1. //Creating an object or instance
2. let **obj** = **new** Student();
- 3.
4. //Initializing an object by reference variable
5. **obj.id** = 101;
6. **obj.name** = "Virat Kohli";

2. By method

A method is similar to a function used to expose the behavior of an object.

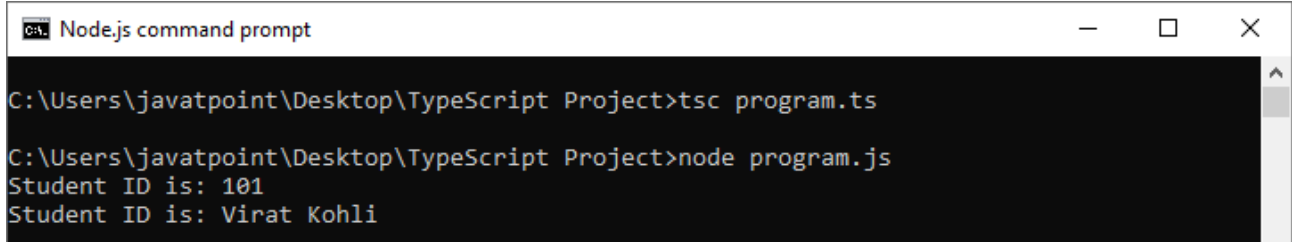
Advantage of Method

- Code Reusability
- Code Optimization

Example

1. //Defining a Student class.
2. class Student {
3. //defining fields
4. id: number;
5. name:string;
- 6.
7. //creating method or function
8. display():void {
9. console.log("Student ID is: "+this.id)
10. console.log("Student ID is: "+this.name)
11. }
12. }
- 13.
14. //Creating an object or instance
15. let **obj** = **new** Student();
16. **obj.id** = 101;
17. **obj.name** = "Virat Kohli";
18. obj.display();

Output:



```
Node.js command prompt
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts
C:\Users\javatpoint\Desktop\TypeScript Project>node program.js
Student ID is: 101
Student ID is: Virat Kohli
```

3. By Constructor

A constructor is used to **initialize** an object. In TypeScript, the constructor method is always defined with the name "**constructor**." In the constructor, we can access the member of a class by using **this** keyword.

Note: It is not necessary to always have a constructor in the class.

Example

1. //defining constructor
2. constructor(id: number, name:string) {
3. **this.id** = id;
4. **this.name** = name;
5. }

Extending Interfaces

Interfaces can extend each other's definition.

Extending an interface means you are creating a new interface with the same properties as the original, plus something new.

Example

```
interface Rectangle {
  height: number,
  width: number
}

interface ColoredRectangle extends Rectangle {
  color: string
```



```
}
```

```
const coloredRectangle: ColoredRectangle = {  
  height: 20,  
  width: 10,  
  color: "red"  
};
```

OUTPUT:

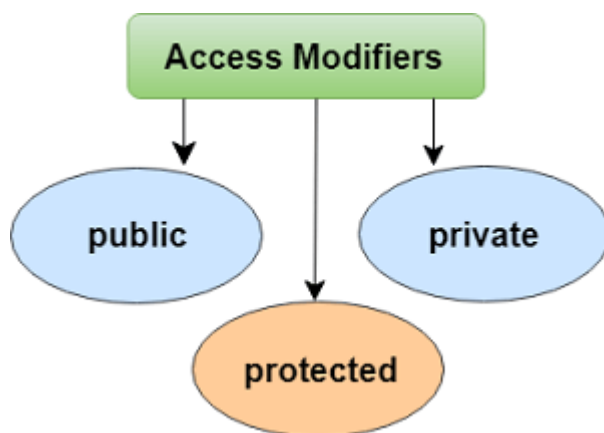
```
{ height: 20, width: 10, color: 'red' }
```

Access Modifiers:

TypeScript Access Modifiers

Like other programming languages, Typescript allows us to use access modifiers at the class level. It gives direct access control to the class member. These class members are functions and properties. We can use class members inside its own class, anywhere outside the class, or within its child or derived class.

The access modifier increases the security of the class members and prevents them from invalid use. We can also use it to control the visibility of data members of a class. If the class does not have to be set any access modifier, TypeScript automatically sets public access modifier to all class members.



The TypeScript access modifiers are of three types. These are:

1. Public
2. Private
3. Protected.

Understanding all TypeScript access modifiers

Let us understand the access modifiers with a given table.

Access Modifier	Accessible within class	Accessible in subclass	Accessible externally via class instance
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Public

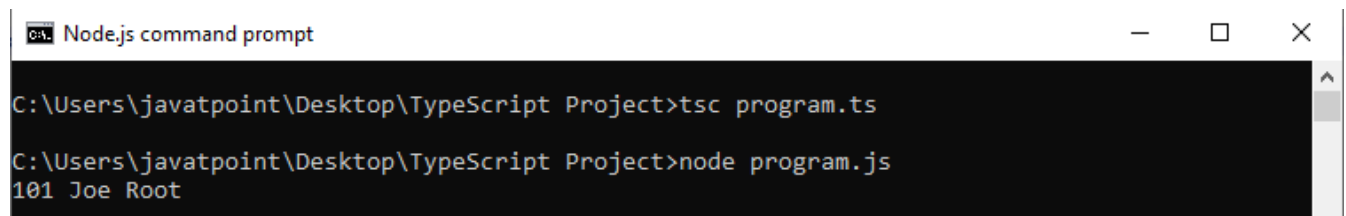
In TypeScript by default, all the members (properties and methods) of a class are public. So, there is no need to prefix members with this keyword. We can access this data member anywhere without any restriction.

Example

```
1. class Student {  
2.     public studCode: number;  
3.     studName: string;  
4. }  
5.  
6. let stud = new Student();  
7. stud.studCode = 101;  
8. stud.studName = "Joe Root";  
9.  
10. console.log(stud.studCode+ " "+stud.studName);
```

In the above example, **studCode** is public, and **studName** is declared without a modifier, so TypeScript treats them as **public** by default. Since data members are public, they can be accessed outside of the class using an object of the class.

Output:



```
Node.js command prompt  
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts  
C:\Users\javatpoint\Desktop\TypeScript Project>node program.js  
101 Joe Root
```

Private

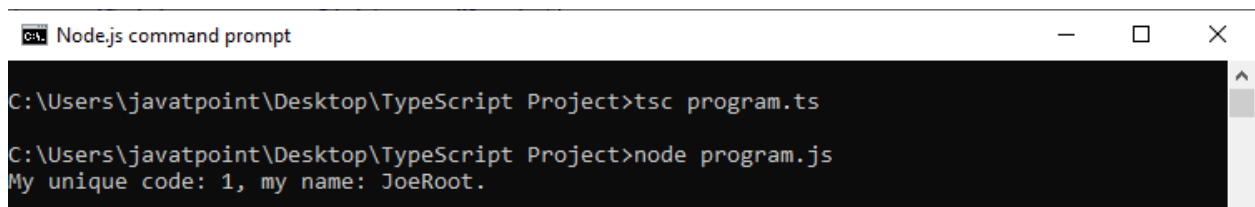
The private access modifier cannot be accessible outside of its containing class. It ensures that the class members are visible only to that class in which it is containing.

Example

```
1. class Student {
2.   public studCode: number;
3.   private studName: string;
4.   constructor(code: number, name: string){
5.     this.studCode = code;
6.     this.studName = name;
7.   }
8.   public display() {
9.     return `My unique code: ${this.studCode}, my name: ${this.studName}.`;
10.  }
11. }
12.
13. let student: Student = new Student(1, "JoeRoot");
14. console.log(student.display());
```

In the above example, **studCode** is private, and **studName** is declared without a modifier, so TypeScript treats it as public by default. If we access the private member outside of the class, it will give a compile error.

Output:



```
Node.js command prompt
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts
C:\Users\javatpoint\Desktop\TypeScript Project>node program.js
My unique code: 1, my name: JoeRoot.
```

Protected

A Protected access modifier can be accessed only within the class and its subclass. We cannot access it from the outside of a class in which it is containing.

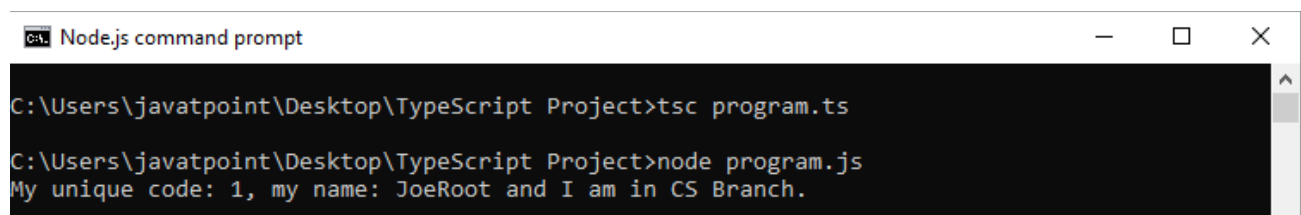
Example

```
1. class Student {
2.   public studCode: number;
3.   protected studName: string;
4.   constructor(code: number, name: string){
5.     this.studCode = code;
```

```
6.     this.studName = name;
7.     }
8. }
9. class Person extends Student {
10.   private department: string;
11.
12.   constructor(code: number, name: string, department: string) {
13.     super(code, name);
14.     this.department = department;
15.   }
16.   public getElevatorPitch() {
17.     return (`My unique code: ${this.studCode}, my name: ${this.studName} and I am in ${this.
       department} Branch.`);
18.   }
19. }
20. let joeRoot: Person = new Person(1, "JoeRoot", "CS");
21. console.log(joeRoot.getElevatorPitch());
```

In the above example, we can't use the name from outside of **Student** class. We can still use it from within an instance method of **Person** class because **Person** class derives from **Student** class.

Output:



```
Node.js command prompt
C:\Users\javatpoint\Desktop\TypeScript Project>tsc program.ts
C:\Users\javatpoint\Desktop\TypeScript Project>node program.js
My unique code: 1, my name: JoeRoot and I am in CS Branch.
```

TypeScript 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**.

```
1. namespace <namespace_name> {
2.     export interface I1 { }
```

3. `export class c1{ }`
4. `}`

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

`namespaceName.className;`

1. `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" />
```

Example

The following program helps us to understand the use of namespaces.

Create Project and Declare files

NameSpace file: **studentCalc**

```
namespace studentCalc{
```

1. `export function AnualFeeCalc(feeAmount: number, term: number){`
2. `return feeAmount * term;`
3. `}`
4. `}`

Main File: **app.ts**

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

- 1.
2. `let TotalFee = studentCalc.AnualFeeCalc(1500, 4);`
- 3.

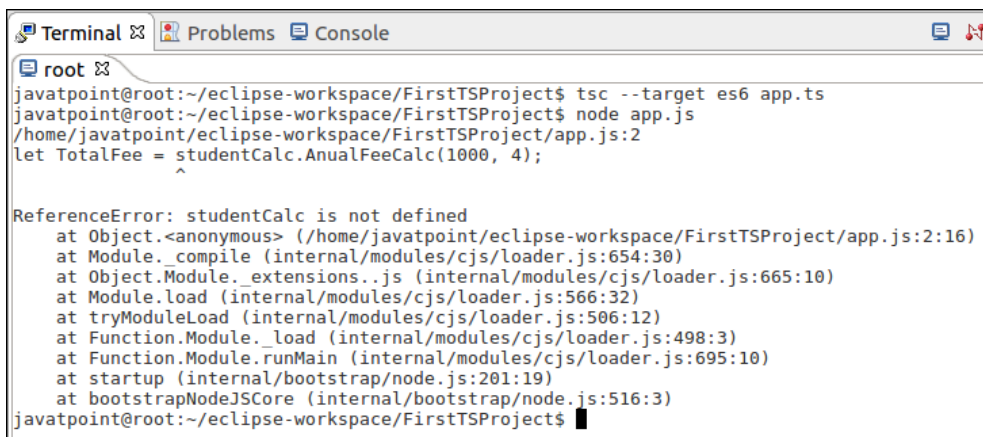
4. `console.log("Output: " +TotalFee);`

Compiling and Executing Namespaces

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

1. `$ tsc --target es6 app.ts`
2. `$ node app.js`

We will see the output below: **studentCalc** is not defined.



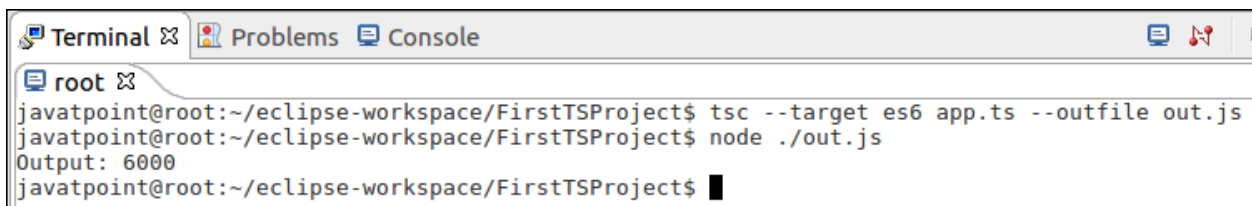
```
Terminal Problems Console
root
javatpoint@root:~/eclipse-workspace/FirstTSProject$ tsc --target es6 app.ts
javatpoint@root:~/eclipse-workspace/FirstTSProject$ node app.js
/home/javatpoint/eclipse-workspace/FirstTSProject/app.js:2
let TotalFee = studentCalc.AnuualFeeCalc(1000, 4);
                  ^
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$
```

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`

1. `$ 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$
```

TypeScript Module

JavaScript has a concept of modules from **ECMAScript 2015**. TypeScript shares this concept of a module.

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.

We can divide the module into **two** categories:

1. Internal Module
2. External Module

Internal Module

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.

Internal Module Syntax in Earlier Version:

```
module Sum {
```

1. export function add(a, b) {
2. console.log("Sum: " +(a+b));
3. }
4. }

Internal Module Syntax from ECMAScript 2015:

```
namespace Sum {
```

1. export function add(a, b) {
2. console.log("Sum: " +(a+b));
3. }

4. }

External Module

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. ECMAScript 2015(ES6) module system treats every file as a module.

Module declaration

We can declare a module by using the **export** keyword. The syntax for the module declaration is given below.

```
//FileName : EmployeeInterface.ts
```

1. export interface Employee {
2. //code declarations
3. }

We can use the declare module in other files by using an **import** keyword, which looks like below. The **file/module** name is specified without an **extension**.

```
import { class/interface name } from 'module_name';
```

Example

Let us understand the module with the following example.

Module Creation: **addition.ts**

```
export class Addition{
```

1. constructor(private x?: number, private y?: number){
2. }
3. Sum(){
4. console.log("SUM: " +(this.x + this.y));
5. }

6. }

Accessing the module in another file by using the import keyword: **app.ts**

```
import { Addition } from './addition';
```

- 1.
2. let **addObject** = **new** Addition(10, 20);
- 3.
4. addObject.Sum();

Module vs. Namespace

Module	Namespace
A module is a way which is used to organize the code in separate files and can execute in their local scope, not in the global scope.	A namespace is a way which is used for logical grouping of functionalities with local scoping.
A Module uses the export keyword to expose module functionalities.	We can create a namespace by using the namespace keyword and all the interfaces, classes, functions, and variables can be defined in the curly braces{ } by using the export keyword
All the exports functions and classes in a module are accessible outside the module.	We must use the export keyword for functions and classes to be able to access it outside the namespace.
It is also known as an external module.	It is also known as an internal module
We can compile the module by using the "--module" command.	We can compile the namespace by using the "--outFile" command.
A module can declare their dependencies.	Namespaces cannot declare their dependencies.

Module Declaration:FileName: **addition.ts**

```
export class Addition{
  constructor(private x?: number, private y?:
number){
  }
  Sum(){
    console.log("SUM: " +(this.x + this.y));
  }
}
```

Accessing Modules:

```
import { Addition } from './addition';
let addObject = new Addition(10, 20);
addObject.Sum();
```

Namespace Declaration:FileName: **StoreCalc.ts**

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

Accessing Namespace:

```
///<reference path="./StoreCalc.ts"/>
Letinvoicenew
invoiceCalc.invoiceAccount.Invoice();
console.log("Output:
+invoice.calculateDiscount(400));
```