# **TypeScript Tutorial**



TypeScript tutorial provides basic and advanced concepts of TypeScript. Our TypeScript Tutorial is designed for beginners and professionals both.

TypeScript is an **open-source**, object-oriented programing 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.

Our TypeScript tutorial includes all the topics which help to learn TypeScript. These are Introduction of TypeScript, Features of TypeScript, Components of TypeScript, TypeScript first program, TypeScript Types, TypeScript Variables, TypeScript Operators, TypeScript Decision-making, TypeScript Loops, Functions, Classes, Arrays, Namespace, Module, Ambient, and many more.

## What is TypeScript?

TypeScript is an open-source pure object-oriented programing 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 Introduction

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 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.

## 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.

# **TypeScript Version**

The complete release history for TypeScript is given below. You can also see the full documentation for recent releases on GitHub.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Version** | **Release Date** | **Significant Changes** |
| **1.** | 0.8 | 01-10-2012 |  |
| **2.** | 0.9 | 18-06-2013 |  |
| **3.** | 1.1 | 06-10-2014 | Performance Improvements |
| **4.** | 1.3 | 12-11-2014 | Protected modifier, tuple types |
| **5.** | 1.4 | 20-01-2015 | union types, let and const declarations, type guards, type aliases, template strings |
| **6.** | 1.5 | 20-07-2015 | ES6 modules, decorators, for..of support, namespace keyword |
| **7.** | 1.6 | 16-09-2015 | JSX support, abstract classes and methods, local type declarations, intersection types, user-defined type guard functions |
| **8.** | 1.7 | 30-11-2015 | async and await support |
| **9.** | 1.8 | 22-02-2016 | constraints generic, control flow analysis errors, string literal types, allowJs |
| **10.** | 2.0 | 22-09-2016 | control flow based type analysis, null-and undefined-aware types, never type, discriminated union types, readonly keyword, type of this for functions |
| **11.** | 2.1 | 08-11-2016 | mapped types, keyof and lookup types, object spread and rest |
| **12.** | 2.2 | 22-02-2017 | object type, mix-in classes |
| **13.** | 2.3 | 27-04-2017 | async iteration, strict option, generic parameter defaults |
| **14.** | 2.4 | 27-06-2017 | dynamic import expressions, improved inference for generics, string enums, strict contravariance, for callback parameters |
| **15.** | 2.5 | 31-08-2017 | optional catch clause variables |
| **16.** | 2.6 | 31-10-2017 | strict function types |
| **17.** | 2.7 | 31-01-2018 | fixed length tuples, constant-named properties |
| **18.** | 2.8 | 27-03-2108 | improved keyof with intersection types, conditional types |
| **19.** | 2.9 | 14-05-2018 | support for symbols and numeric literals in keyof and mapped object types |
| **20.** | 3.0 | 30-07-2018 | project references, extracting and spreading parameter lists with tuples |
| **21.** | 3.1 | 27-09-2018 | mappable tuple and array types |
| **22.** | 3.2 | 30-09-2018 | stricter checking for bind, call, apply |
| **23.** | 3.3 | 31-01-2019 | Improved behavior on methods of union types, incremental builds for composite projects. |
| **24.** | 3.4 | 29-03-2019 | Faster incremental builds with the --incremental flag, type inference from generic functions, readonly modifier for arrays and tuples, const assertions, type-checking for globalThis. |
| **25.** | 3.5 | 29-05-2019 | Speed improvements, Improved excess property checks in union types, faster incremental builds, Omit helper type, smarter union type checking |

# **Difference between JavaScript and TypeScript**

## JavaScript

JavaScript is the most popular programming language of HTML and the Web. JavaScript is an object-based scripting language which is lightweight and cross-platform. It is used to create client-side dynamic pages. The programs in JavaScript language are called scripts. The scripts are written in HTML pages and executed automatically as the page loads. It is provided and executed as plain text and does not need special preparation or compilation to run.

### **History of JavaScript**

Netscape Communications Corporation programmer Brendan Eich developed JavaScript. It was introduced in September 1995, which was initially called Mocha. However, after gaining popularity as the best scripting tool, it was renamed as JavaScript to reflect Netscape's support of Java within its browser. In November 1996, Netscape submitted JavaScript to ECMA (European Computer Manufacturers Association). The current version of JavaScript is ECMAScript 2018, which was released in June 2018./p>

## TypeScript

TypeScript is an open-source pure object-oriented programing language. It is a strongly typed superset of JavaScript which compiles to plain JavaScript. TypeScript is developed and maintained by Microsoft under the Apache 2 license. It is not directly run on the browser. It needs a compiler to compile and generate in JavaScript file. TypeScript source file is in ".ts" extension. We can use any valid ".js" file by renaming it to ".ts" file. TypeScript is the ES6 version of JavaScript with some additional features.

### **History of TypeScript**

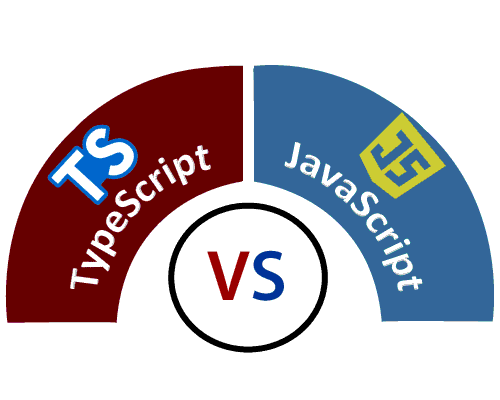
Anders Hejlsberg developed TypeScript. It was first introduced for the public in the month of 1st October 2012. After two years of internal development at Microsoft, the new version of TypeScript 0.9 was released in 2013. The current version of TypeScript is TypeScript 3.4.5 which was released on 24 April 2019.

## Advantage of TypeScript over JavaScript

* TypeScript always highlights errors at compilation time during the time of development, whereas JavaScript points out errors at the runtime.
* TypeScript supports strongly typed or static typing, whereas this is not in JavaScript.
* TypeScript runs on any browser or JavaScript engine.
* Great tooling supports with IntelliSense which provides active hints as the code is added.
* It has a namespace concept by defining a module.

## Disadvantage of TypeScript over JavaScript

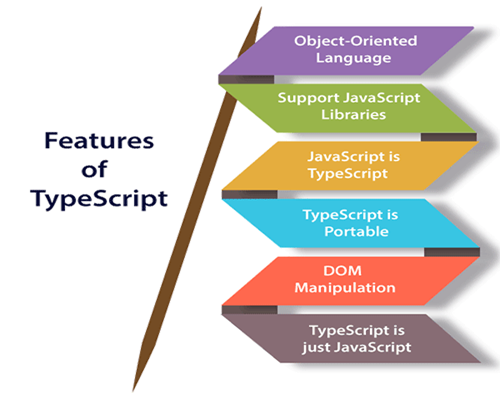
* TypeScript takes a long time to compile the code.
* TypeScript does not support abstract classes.
* If we run the TypeScript application in the browser, a compilation step is required to transform TypeScript into JavaScript.



## TypeScript Vs. JavaScript

|  |  |  |
| --- | --- | --- |
| **SN** | **JavaScript** | **TypeScript** |
| 1. | It doesn't support strongly typed or static typing. | It supports strongly typed or static typing feature. |
| 2. | Netscape developed it in 1995. | Anders Hejlsberg developed it in 2012. |
| 3. | JavaScript source file is in ".js" extension. | TypeScript source file is in ".ts" extension. |
| 4. | It is directly run on the browser. | It is not directly run on the browser. |
| 5. | It is just a scripting language. | It supports object-oriented programming concept like classes, interfaces, inheritance, generics, etc. |
| 6. | It doesn't support optional parameters. | It supports optional parameters. |
| 7. | It is interpreted language that's why it highlighted the errors at runtime. | It compiles the code and highlighted errors during the development time. |
| 8. | JavaScript doesn't support modules. | TypeScript gives support for modules. |
| 9. | In this, number, string are the objects. | In this, number, string are the interface. |
| 10. | JavaScript doesn't support generics. | TypeScript supports generics. |
| 11. | **Example:**  <script>  function addNumbers(a, b) {  return a + b;  }  var sum = addNumbers(15, 25);  document.write('Sum of the numbers is: ' + sum);  </script> | **Example:**  function addNumbers(a, b) {  return a + b;  }  var sum = addNumbers(15, 25);  console.log('Sum of the numbers is: ' + |

# **Features of TypeScript**



**Object-Oriented language:** TypeScript provides a complete feature of an object-oriented programming language such as classes, interfaces, inheritance, modules, etc. In TypeScript, we can write code for both client-side as well as server-side development.

**TypeScript supports JavaScript libraries:** TypeScript supports each JavaScript elements. It allows the developers to use existing JavaScript code with the TypeScript. Here, we can use all of the JavaScript frameworks, tools, and other libraries easily.

**JavaScript is TypeScript:** It means the code written in JavaScript with valid .js extension can be converted to TypeScript by changing the extension from .js to .ts and compiled with other TypeScript files.

**TypeScript is portable:** TypeScript is portable because it can be executed on any browsers, devices, or any operating systems. It can be run in any environment where JavaScript runs on. It is not specific to any virtual-machine for execution.

**DOM Manipulation:** TypeScript can be used to manipulate the DOM for adding or removing elements similar to JavaScript.

**TypeScript is just a JS:** TypeScript code is not executed on any browsers directly. The program written in TypeScript always starts with JavaScript and ends with JavaScript. Hence, we only need to know JavaScript to use it in TypeScript. The code written in TypeScript is compiled and converted into its JavaScript equivalent for the execution. This process is known as **Trans-piled**. With the help of JavaScript code, browsers can read the TypeScript code and display the output.

## Advantage of TypeScript over JavaScript

* TypeScript always highlights errors at compilation time during the time of development, whereas JavaScript points out errors at the runtime.
* TypeScript supports strongly typed or static typing, whereas this is not in JavaScript.
* TypeScript runs on any browser or JavaScript engine.
* Great tooling supports with IntelliSense, which provides active hints as the code is added.
* It has a namespace concept by defining a module.

## Disadvantage of TypeScript over JavaScript

* TypeScript takes a long time to compile the code.
* TypeScript does not support abstract classes.
* If we run the TypeScript application in the browser, a compilation step is required to transform TypeScript into JavaScript.

# **Components of TypeScript**

The TypeScript language is internally divided into three main layers. Each of these layers is divided into sublayers or components. In the following diagram, we can see the three layers and each of their internal components. These layers are:

1. Language
2. The TypeScript Compiler
3. The TypeScript Language Services

Components of TypeScript

## 1. Language

It features the TypeScript language elements. It comprises elements like syntax, keywords, and type annotations.

## 2. The TypeScript Compiler

The TypeScript compiler (TSC) transform the TypeScript program equivalent to its JavaScript code. It also performs the parsing, and type checking of our TypeScript code to JavaScript code.

Components of TypeScript

Browser doesn't support the execution of TypeScript code directly. So the program written in TypeScript must be re-written in JavaScript equivalent code which supports the execution of code in the browser directly. To perform this, TypeScript comes with TypeScript compiler named "tsc." The current version of TypeScript compiler supports ES6, by default. It compiles the source code in any module like ES6, SystemJS, AMD, etc.

We can install the TypeScript compiler by locally, globally, or both with any **npm** package. Once installation completes, we can compile the TypeScript file by running "tsc" command on the command line.

### **Example:**

1. $ tsc helloworld.ts   // It compiles the TS file helloworld into the helloworld.js file.

### **Compiler Configuration**

The TypeScript compiler configuration is given in **tsconfig.json** file and looks like the following:

1. {
2. "compilerOptions": {
3. "declaration": **true**,
4. "emitDecoratorMetadata": **false**,
5. "experimentalDecorators": **false**,
6. "module": "none",
7. "moduleResolution": "node",
8. "noFallthroughCasesInSwitch": **false**,
9. "noImplicitAny": **false**,
10. "noImplicitReturns": **false**,
11. "removeComments": **false**,
12. "sourceMap": **false**,
13. "strictNullChecks": **false**,
14. "target": "es3"
15. },
16. "compileOnSave": **true**
17. }

### **Declaration file**

When we compile the TypeScript source code, it gives an option to generate a **declaration file** with the extension **.d.ts**. This file works as an interface to the components in the compiled JavaScript. If a file has an extension **.d.ts**, then each root level definition must have the **declare** keyword prefixed to it. It makes clear that there will be no code emitted by TypeScript, which ensures that the declared item will exist at runtime. The declaration file provides IntelliSense for JavaScript libraries like jQuery.

## 3. The TypeScript Language Services

The language service provides information which helps editors and other tools to give better assistance features such as automated refactoring and IntelliSense. It exposes an additional layer around the core-compiler pipeline. It supports some standard typical editor operations like code formatting and outlining, colorization, statement completion, signature help, etc.

# **TypeScript Installation**

In this section, we will learn how to install TypeScript, pre-requisites before installation of TypeScript, and in how many ways we can install TypeScript.

### **Pre-requisite to install TypeScript**

1. Text Editor or IDE
2. Node.js Package Manager (npm)
3. The TypeScript compiler

### **Ways to install TypeScript**

There are two ways to install TypeScript:

1. Install TypeScript using Node.js Package Manager (npm).
2. Install the TypeScript plug-in in your IDE (Integrated Development Environment).

### **Install TypeScript using Node.js Package Manager (npm)**

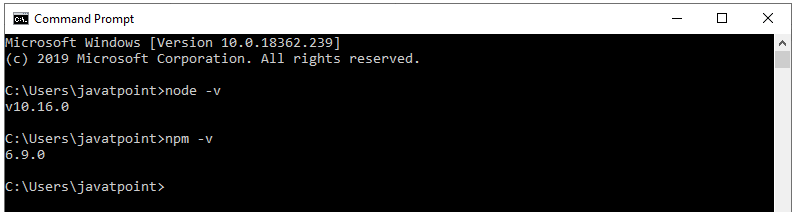
**Step-1** Install Node.js. It is used to setup TypeScript on our local computer.

To install Node.js on Windows, go to the following link: [**https://www.javatpoint.com/install-nodejs**](https://www.javatpoint.com/install-nodejs)

To install Node.js in Linux/Ubuntu/CentOS, go to the following link: [**https://www.javatpoint.com/install-nodejs-on-linux-ubuntu-centos**](https://www.javatpoint.com/install-nodejs-on-linux-ubuntu-centos)

To verify the installation was successful, enter the following command in the Terminal Window.

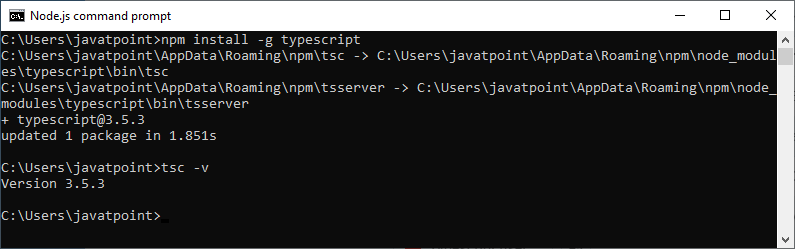
1. $ node -v
2. $ npm -v



**Step-2** Install TypeScript. To install TypeScript, enter the following command in the Terminal Window.

1. $ npm install typescript --save-dev         //As dev dependency
2. $ npm install typescript -g                      //Install as a global module
3. $ npm install typescript@latest -g          //Install latest if you have an older version

**Step-3** To verify the installation was successful, enter the command **$ tsc -v** in the Terminal Window.



### **Install TypeScript plug-in in your IDE**

**Step-1** Install IDE like Eclipse, Visual Studio, WebStorm, Atom, Sublime Text, etc. Here, we install Eclipse. To install Eclipse, go to the following link:

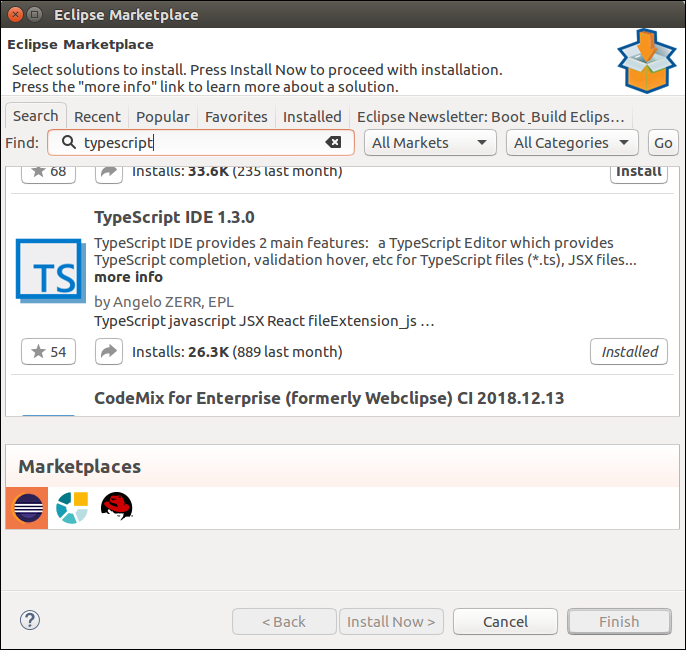
**In Windows:** [**https://www.javatpoint.com/javafx-how-to-install-eclipse**](https://www.javatpoint.com/javafx-how-to-install-eclipse)

**In Ubantu:** [**https://www.javatpoint.com/how-to-install-eclipse-in-ubuntu**](https://www.javatpoint.com/how-to-install-eclipse-in-ubuntu)

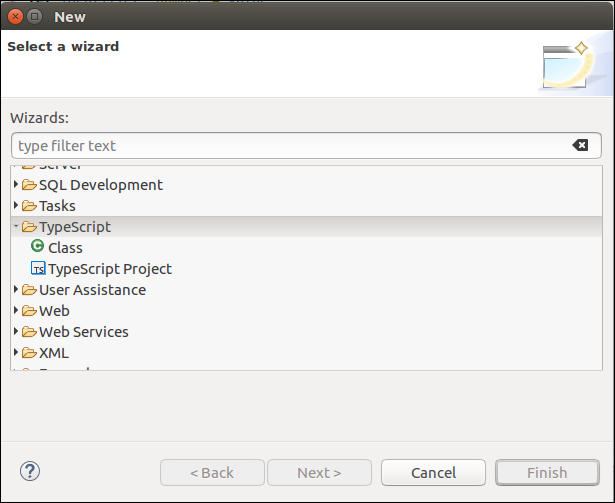
**In CentOS:** [**https://www.javatpoint.com/how-to-install-eclipse-on-centos**](https://www.javatpoint.com/how-to-install-eclipse-on-centos)

**Step-2** Install TypeScript plug-in.

* Open Eclipse and go to **Help->Eclipse Market Place**.
* Search for **TypeScript** and choose **TypeScript IDE**, Click Install.



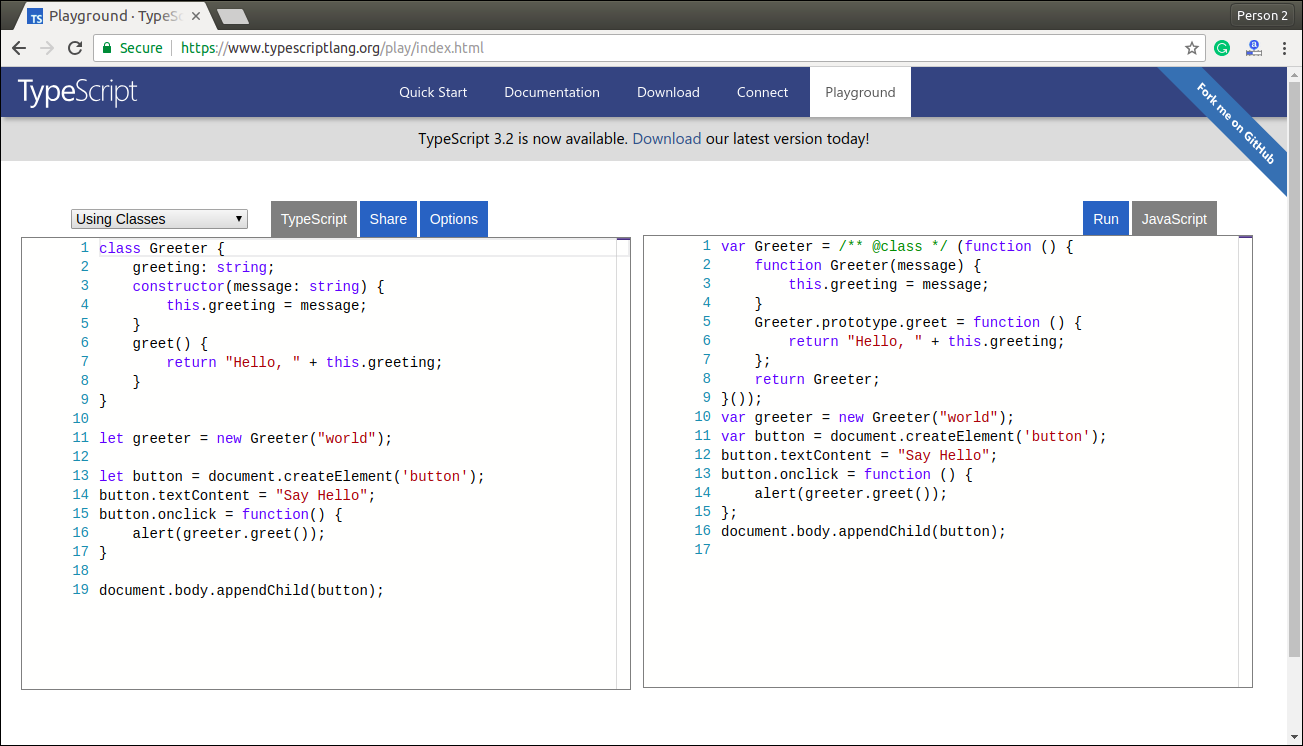
* In the next window, select **Features** which you want to install, and click **Confirm**.
* A new window will open, select **Accept Terms and Condition**, Click **Next**, and follow the on-screen instructions.
* Now **restart** Eclipse. To verify the TypeScript, go to **New->Other->TypeScript**. Once the TypeScript shows in the window, it means that TypeScript is successfully installed on your machine.



## Online Compiler for TypeScript

We can also run our script online with the official compiler. To do this, go to the below link. [**https://www.typescriptlang.org/play/index.html**](https://www.typescriptlang.org/play/index.html)

The following screen appears. Now, you can do any TypeScript program on this.



# **TypeScript First Program**

In this section, we are going to learn how we can write a program in TypeScript, how to compile it, and how to run it. Also, we will see how to compiles the program and shows the error, if any.

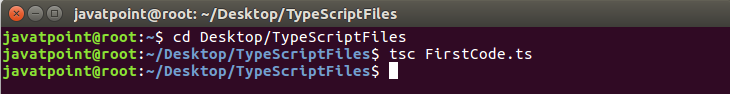
Let us write a program in the text editor, save it, compile it, run it, and display the output to the console. To do this, we need to perform the following steps.

**Step-1** Open the Text Editor and write/copy the following code.

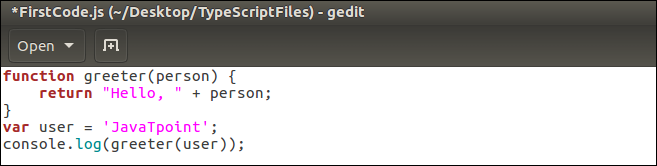
1. function greeter(person) {
2. **return** "Hello, " + person;
3. }
4. let user = 'JavaTpoint';
5. console.log(greeter(user));

**Step-2** Save the above file as "**.ts**" extension.

**Step-3** Compile the TypeScript code. To compile the source code, open the **command prompt**, and then goes to the file directory location where we saved the above file. For example, if we save the file on the desktop, go to the terminal window and type: - **cd Desktop/folder\_name**. Now, type the following command tsc **filename.ts** for compilation and press **Enter**.

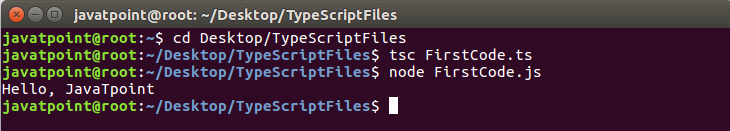


It will generate JavaScript file with ".js" extension at the same location where the TypeScript source file exists. The below ".js" file is the output of TypeScript (.ts) file.



#### NOTE:**If we directly run "**.ts**" file on the web browser, it will throw an error message. But after the compilation of "**.ts**" file, we will get a "**.js**" file, which can be executed on any browser.**

**Step-4** Now, to run the above JavaScript file, type the following command in the terminal window: node filename.js and press Enter. It gives us the final output as:



## Compile-Time error

TypeScript always gives an error at compilation time. For this, we need to write the program in TypeScript, compile it, and see the error, if found.

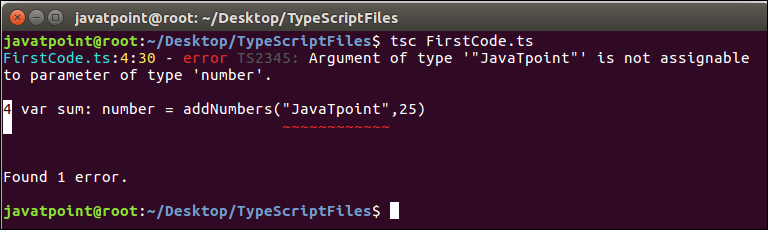
**Step 1** Open the Text Editor and write/copy the following code.

1. function addNumbers(a, b) {
2. **return** a + b;
3. }
4. var sum = addNumbers("JavaTpoint", 25);
5. console.log('Sum of the numbers is: ' + sum);

**Step-2** Save the above file as "**.ts**" extension.

**Step-3** Compile the TypeScript code. To compile the source code, open the **command prompt**, and then goes to the file directory location where we saved the above file. For example, if we save the file on the desktop, go to the terminal window and type: - **cd Desktop/folder\_name**. Now, type the following command **tsc filename.ts** for compilation and press **Enter**.

This TypeScript source file will generate an error which can be shown in the following image.

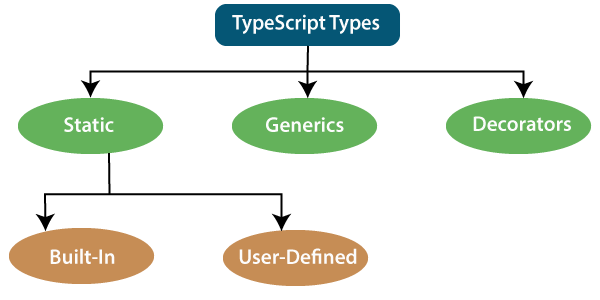


#### NOTE:**This program gives an error because we were taking the variable "a" and "b" as of number type. But, we were passing the variable "a" as the string and variable "b" as the number.**

# **TypeScript Type**

The TypeScript language supports different types of values. It provides data types for the JavaScript to transform it into a strongly typed programing language. JavaScript doesn't support data types, but with the help of TypeScript, we can use the data types feature in JavaScript. TypeScript plays an important role when the object-oriented programmer wants to use the type feature in any scripting language or object-oriented programming language. The Type System checks the validity of the given values before the program uses them. It ensures that the code behaves as expected.

TypeScript provides data types as an optional Type System. We can classify the TypeScript data type as following.



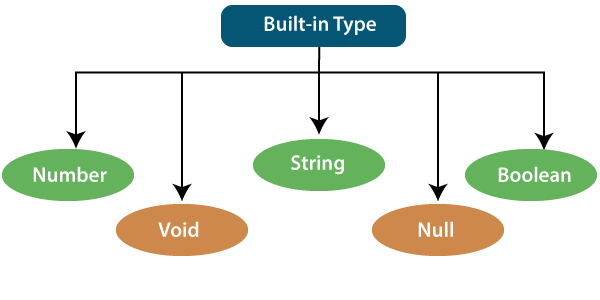
## 1. Static Types

In the context of type systems, static types mean "at compile time" or "without running a program." In a statically typed language, variables, parameters, and objects have types that the compiler knows at compile time. The compiler used this information to perform the type checking.

Static types can be further divided into two sub-categories:

### **Built-in or Primitive Type**

The TypeScript has five built-in data types, which are given below.



### **Number**

Like JavaScript, all the numbers in TypeScript are stored as floating-point values. These numeric values are treated like a number data type. The numeric data type can be used to represents both integers and fractions. TypeScript also supports Binary(Base 2), Octal(Base 8), Decimal(Base 10), and Hexadecimal(Base 16) literals.

**Syntax:**

1. let identifier: number = value;

**Examples:-**

1. let first: number = 12.0;             // number
2. let second: number = 0x37CF;          // hexadecimal
3. let third: number = 0o377 ;           // octal
4. let fourth: number = 0b111001;        // binary
6. console.log(first);           // 123
7. console.log(second);          // 14287
8. console.log(third);           // 255
9. console.log(fourth);          // 57

### **String**

We will use the string data type to represents the text in TypeScript. String type work with textual data. We include string literals in our scripts by enclosing them in single or double quotation marks. It also represents a sequence of Unicode characters. It embedded the expressions in the form of **$ {expr}**.

**Syntax**

1. let identifier: string = " ";
2. Or
3. let identifier: string = ' ';

**Examples**

1. let empName: string = "Rohan";
2. let empDept: string = "IT";
4. // Before-ES6
5. let output1: string = employeeName + " works in the " + employeeDept + " department.";
7. // After-ES6
8. let output2: string = `${empName} works in the ${empDept} department.`;
10. console.log(output1);//Rohan works in the IT department.
11. console.log(output2);//Rohan works in the IT department.

### **Boolean**

The string and numeric data types can have an unlimited number of different values, whereas the Boolean data type can have only two values. They are "true" and "false." A Boolean value is a truth value which specifies whether the condition is true or not.

**Syntax**

1. let identifier: BooleanBoolean = Boolean value;

**Examples**

1. let isDone: boolean = false;

### **Void**

A void is a return type of the functions which do not return any type of value. It is used where no data type is available. A variable of type void is not useful because we can only assign undefined or null to them. An undefined data type denotes uninitialized variable, whereas null represents a variable whose value is undefined.

**Syntax**

1. let unusable: void = undefined;

**Examples**

1. 1. function helloUser(): void {
2. alert("This is a welcome message");
3. }
4. 2. let tempNum: void = undefined;
5. tempNum = null;
6. tempNum = 123;      //Error

### **Null**

Null represents a variable whose value is undefined. Much like the void, it is not extremely useful on its own. The Null accepts the only one value, which is null. The Null keyword is used to define the Null type in TypeScript, but it is not useful because we can only assign a null value to it.

**Examples**

1. let num: number = null;
2. let bool: boolean = null;
3. let str: string = null;

## Undefined

The Undefined primitive type denotes all uninitialized variables in TypeScript and JavaScript. It has only one value, which is undefined. The undefined keyword defines the undefined type in TypeScript, but it is not useful because we can only assign an undefined value to it.

**Example**

1. let num: number = undefined;
2. let bool: boolean = undefined;
3. let str: string = undefined;

### **Any Type**

It is the "super type" of all data type in TypeScript. It is used to represents any JavaScript value. It allows us to opt-in and opt-out of type-checking during compilation. If a variable cannot be represented in any of the basic data types, then it can be declared using "**Any**" data type. Any type is useful when we do not know about the type of value (which might come from an API or 3rd party library), and we want to skip the type-checking on compile time.

**Syntax**

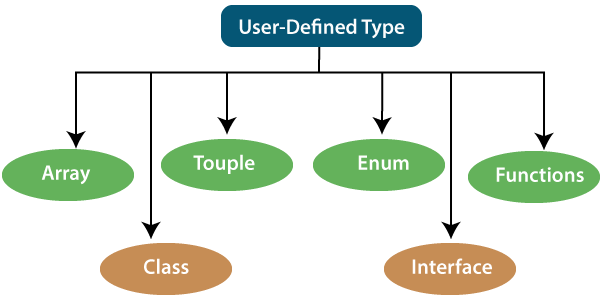
1. let identifier: any = value;

**Examples**

1. 1. let val: any = 'Hi';
2. val = 555;   // OK
3. val = true;   // OK
4. 2. function ProcessData(x: any, y: any) {
5. return x + y;
6. }
7. let result: any;
8. result = ProcessData("Hello ", "Any!"); //Hello Any!
9. result = ProcessData(2, 3); //5

### **User-Defined DataType**

TypeScript supports the following user-defined data types:



### **Array**

An array is a collection of elements of the same data type. Like JavaScript, TypeScript also allows us to work with arrays of values. An array can be written in two ways:

1. Use the type of the elements followed by [] to denote an array of that element type:

1. var list : number[] = [1, 3, 5];

2. The second way uses a generic array type:

1. var list : Array**<number>** = [1, 3, 5];

### **Touple**

The Tuple is a data type which includes two sets of values of different data types. It allows us to express an array where the type of a fixed number of elements is known, but they are not the same. For example, if we want to represent a value as a pair of a number and a string, then it can be written as:

1. // Declare a tuple
2. let a: [string, number];
4. // Initialize it
5. a = ["hi", 8, "how", 5]; // OK

### **Interface**

An Interface is a structure which acts as a contract in our application. It defines the syntax for classes to follow, means a class which implements an interface is bound to implement all its members. It cannot be instantiated but can be referenced by the class which implements it. The TypeScript compiler uses interface for type-checking that is also known as "duck typing" or "structural subtyping."

**Example**

1. interface Calc {
2. subtract (first: number, second: number): any;
3. }
5. let Calculator: Calc = {
6. subtract(first: number, second: number) {
7. return first - second;
8. }
9. }

### **Class**

Classes are used to create reusable components and acts as a template for creating objects. It is a logical entity which store variables and functions to perform operations. TypeScript gets support for classes from ES6. It is different from the interface which has an implementation inside it, whereas an interface does not have any implementation inside it.

**Example**

1. class Student
2. {
3. RollNo: number;
4. Name: string;
5. constructor(\_RollNo: number, Name: string)
6. {
7. this.RollNo = \_rollNo;
8. this.Name = \_name;
9. }
10. showDetails()
11. {
12. console.log(this.rollNo + " : " + this.name);
13. }
14. }

### **Enums**

Enums define a set of named constant. TypeScript provides both string-based and numeric-based enums. By default, enums begin numbering their elements starting from 0, but we can also change this by manually setting the value to one of its elements. TypeScript gets support for enums from ES6.

**Example**

1. enum Color {
2. Red, Green, Blue
3. };
4. let c: Color;
5. ColorColor = Color.Green;

### **Functions**

A function is the logical blocks of code to organize the program. Like JavaScript, TypeScript can also be used to create functions either as a **named function** or as an **anonymous function**. Functions ensure that our program is readable, maintainable, and reusable. A function declaration has a function's name, return type, and parameters.

**Example**

1. //named function with number as parameters type and return type
2. function add(a: number, b: number): number {
3. return a + b;
4. }
6. //anonymous function with number as parameters type and return type
7. let sum = function (a: number, y: number): number {
8. return a + b;
9. };

## 2. Generic

Generic is used to create a component which can work with a variety of data type rather than a single one. It allows a way to create reusable components. It ensures that the program is flexible as well as scalable in the long term. TypeScript uses generics with the type variable <T> that denotes types. The type of generic functions is just like non-generic functions, with the type parameters listed first, similarly to function declarations.

**Example**

1. function identity**<T>**(arg: T): T {
2. return arg;
3. }
4. let output1 = identity**<string>**("myString");
5. let output2 = identity**<number>**( 100 );

## 3. Decorators

A decorator is a special of data type which can be attached to a class declaration, method, property, accessor, and parameter. It provides a way to add both annotations and a meta-programing syntax for classes and functions. It is used with "@" symbol.

A decorator is an experimental feature which may change in future releases. To enable support for the decorator, we must enable the **experimentalDecorators** compiler option either on the **command line** or in our tsconfig.json.

**Example**

1. function f() {
2. console.log("f(): evaluated");
3. return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
4. console.log("f(): called");
5. }
6. }
8. class C {
9. @f()
10. method() {}
11. }

# **Difference between Null and Undefined**

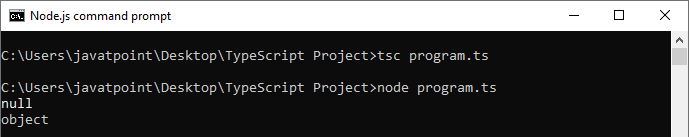
## Null

Null is used to represent an intentional absence of value. It represents a variable whose value is undefined. It accepts only one value, which is null. The Null keyword is used to define the Null type in TypeScript, but it is not useful because we can only assign a null value to it.

### **Example**

1. //Variable declared and assigned to null
2. var a = null;
3. console.log( a );   //output: null
4. console.log( typeof(a) );   //output: object

**Output:**



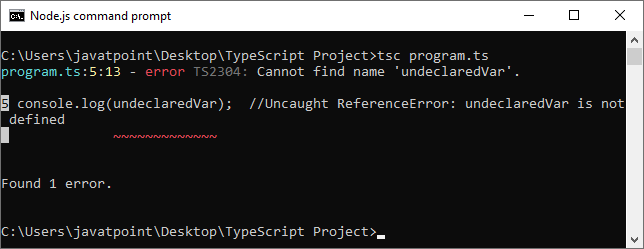
## Undefined

It represents uninitialized variables in TypeScript and JavaScript. It has only one value, which is undefined. The undefined keyword defines the undefined type in TypeScript, but it is not useful because we can only assign an undefined value to it.

### **Example**

1. //Variable declaration without assigning any value to it
2. var a;
3. console.log(a);  //undefined
4. console.log(typeof(a));  //undefined
5. console.log(undeclaredVar);  //Uncaught ReferenceError: undeclaredVar is not defined

**Output:**



## Null vs. Undefined

The important difference between Null and Undefined are:

|  |  |  |
| --- | --- | --- |
| **SN** | **Null** | **Undefined** |
| 1. | It is an assignment value. It can be assigned to a variable which indicates that a variable does not point any object. | It is not an assignment value. It means a variable has been declared but has not yet been assigned a value. |
| 2. | It is an object. | It is a type itself. |
| 3. | The null value is a primitive value which represents the null, empty, or non-existent reference. | The undefined value is a primitive value, which is used when a variable has not been assigned a value. |
| 4. | Null indicates the absence of a value for a variable. | Undefined indicates the absence of the variable itself. |
| 5. | Null is converted to zero (0) while performing primitive operations. | Undefined is converted to NaN while performing primitive operations. |

# **TypeScript Variables**

A variable is the storage location, which is used to store value/information to be referenced and used by programs. It acts as a container for value in code and must be declared before the use. We can declare a variable by using the var keyword. In TypeScript, the variable follows the same naming rule as of JavaScript variable declaration. These rules are-

* The variable name must be an **alphabet** or **numeric digits**.
* The variable name cannot start with digits.
* The variable name cannot contain **spaces** and **special character**, except the u**nderscore(\_)** and the **dollar($)** sign.

In **ES6**, we can define variables using **let** and **const** keyword. These variables have similar syntax for variable declaration and initialization but differ in scope and usage. In TypeScript, there is always recommended to define a variable using **let** keyword because it provides the **type safety**.

The **let** keyword is similar to **var** keyword in some respects, and **const** is an let which prevents prevents re-assignment to a variable.

## Variable Declaration

We can declare a variable in one of the four ways:

**1. Declare type and value in a single statement**

1. var [identifier] : [type-annotation] = value;

**2. Declare type without value. Then the variable will be set to undefined.**

1. var [identifier] : [type-annotation];

**3. Declare its value without type. Then the variable will be set to any.**

1. var [identifier] = value;

**4. Declare without value and type. Then the variable will be set to any and initialized with undefined.**

1. var [identifier];

Let's understand all the three variable keywords one by one.

## var keyword

Generally, **var** keyword is used to declare a variable in JavaScript.

1. var x = 50;

We can also declare a variable inside the function:

1. function a() {
2. var msg = " Welcome to JavaTpoint !! ";
3. return msg;
4. }
5. a();

We can also access a variable of one function with the other function:

1. function a() {
2. var x = 50;
3. return function b() {
4. var y = x+5;
5. return y;
6. }
7. }
8. var  b = a();
9. b();       //returns '55'

### **Scoping rules**

For other language programmers, they are getting some odd scoping rules for var declaration in JavaScript. Variables declared in TypeScript with the var keyword have function scope. This variable has global scope in the function where they are declared. It can also be accessed by any function which shares the same scope.

**Example**

1. function f()
2. {
3. var X = 5; //Available globally inside f()
4. if(true)
5. {
6. var Y = 10; //Available globally inside f()
7. console.log(X); //Output 5
8. console.log(Y); //Output 10
9. }
10. console.log(X); //Output 5
11. console.log(Y); //Output 10
12. }
13. f();
14. console.log(X); //Returns undefined because value cannot accesses from outside function
15. console.log(Y); //Returns undefined because value cannot accesses from outside function

#### **NOTE: As var declarations are accessible anywhere within their containing module, function, global scope, or namespace, some people call this var-scoping or function-scoping. Parameters are also called function scoped.**

## let declarations

The let keyword is similar to the var keyword. The var declaration has some problems in solving programs, so ES6 introduced let keyword to declare a variable in TypeSript and JavaScript. The let keyword has some restriction in scoping in comparison of the var keyword.

The let keyword can enhance our code readability and decreases the chance of programming error.

The let statement are written as same syntax as the var statement:

1. var declaration: var b = 50;
2. let declaration: let b = 50;

The key difference between var and let is not in the syntax, but it differs in the semantics. The Variable declared with the let keyword are scoped to the nearest enclosing block which can be smaller than a function block.

### **Block Scoping**

When the variable declared using the let keyword, it uses block scoping or lexical scoping. Unlike variable declared using var keyword whose scopes leak out to their containing function, a block-scoped variable cannot visible outside of its containing block.

1. function f(input: boolean) {
2. let x = 100;
3. if (input) {
4. // "x" exists here
5. let y = x + 1;
6. return y;
7. }
8. // Error: "y" doesn't exist here
9. return y;
10. }

Here, we have two local variables x and y. Scope of x is limited to the body of the function f() while the scope of y is limited to the containing if statement's block.

#### **NOTE- The variables declared in a try-catch clause also have similar scoping rules. For example:**

1. try {
2. throw "Hi!!";
3. }catch (e) {
4. console.log("Hello");
5. }
6. // 'e' doesn't exist here, so error will found
7. console.log(e);

### **Re-declaration and Shadowing**

With the var declaration, it did not matter how many time's we declared variables. We just got only one. In the below example, all declarations of x refer to the same x, which is perfectly valid. But there is some bug, which can be found by the let declaration.

**Example without let keyword:**

1. function f(a) {
2. var a;
3. var a;
4. if (true) {
5. var a;
6. }
7. }

**Example with let keyword:**

1. let a = 10;
2. let a = 20; // it gives error: can't re-declare 'a' in the same scope

**Shadowing** is the act of introducing a new name in a more nested scope. It declares an identifier which has already been declared in an outer scope. This is not incorrect, but there might be confusion. It will make the outer identifier unavailable inside the loop where the loop variable is shadowing it. It can introduce bugs on its own in the event of accidental Shadowing, as well as it also helps in preventing the application from certain bugs.

**Example**

1. var currencySymbol = "$";
2. function showMoney(amount) {
3. var currencySymbol = "€";
4. document.write(currencySymbol + amount);
5. }
6. showMoney("100");

The above example has a global variable name that shares the same name as the inner method. The inner variable used only in that function, and all other functions will use the global variable declaration.

The Shadowing is usually avoided in writing of clearer code. While in some scenarios, if there may be fitting to take advantage of it, we should use it with the best judgment.

## Hoisting

### **Hoisting of var**

Hoisting is a mechanism of JavaScript. In hoisting, variables and function declarations are moved to the top of their enclosing scope before code execution. We can understand it with the following example.

#### **Note: Hoisting does not happen if we initialize the variable.**

**Example**

1. function get(x){
2. console.log(a);  //printing x variable. Value is undefined
3. //declared variable after console hoisted to the top at run time
4. var a = x;
5. //again printing x variable. Value is 3.
6. console.log(a);
7. }
8. get(4);

**Output:**

undefined

4

### **Hoisting of let**

A variable declared with **let** keyword is not hoisted. If we try to use a let variable before it is declared, then it will result in a **ReferenceError**.

**Example**

1. {
2. //program doesn't know about variable b so it will give me an error.
3. console.log(b); // ReferenceError: b is not defined
4. let b = 3;
5. }

## const declarations

The const declaration is used to declare permanent value, which cannot be changed later. It has a fixed value. The const declaration follows the same scoping rules as let declaration, but we cannot re-assign any new value to it.

#### **Note: According to the naming standards, the const variable must be declared in capital letters. Naming standards should be followed to maintain the code for the long run.**

**Example**

1. function constTest(){
2. const VAR = 10;
3. console.log("Value is: " +VAR);
4. }
5. constTest();

**Output:**

Value is: 10

### **What will happen when we try to re-assign the const variable?**

If we try to re-assign the existing const variable in a code, the code will throw an error. So, we cannot re-assign any new value to an existing const variable.

**Example**

1. function constTest(){
2. const VAR = 10;
3. console.log("Output: " +VAR);  // Output: 10
4. const VAR = 10;
5. console.log("Output: " +VAR);  //Uncaught TypeError: Assignment to constant variable
6. }
7. constTest();

**Output:**

SyntaxError: Identifier 'VAR' has already been declared.

# **Difference between let and var keyword**

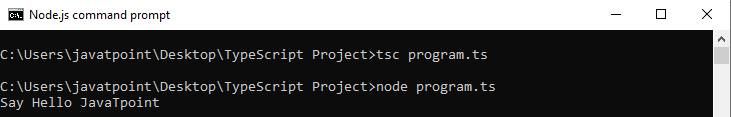
## var keyword

The **var** statement is used to declare a variable in JavaScript. A variable declared with the **var keyword** is defined throughout the program.

### **Example**

1. var greeter = "hey hi";
2. var times = 5;
3. if (times **>** 3) {
4. var greeter = "Say Hello JavaTpoint";
5. }
6. console.log(greeter) //Output: Say Hello JavaTpoint

**Output:**



## let keyword

The **let** statement is used to declare a local variable in TypeScript. It is similar to the var keyword, but it has some restriction in scoping in comparison of the var keyword. The let keyword can enhance our code readability and decreases the chance of programming error. A variable declared with the let keyword is limited to the block-scoped only.

#### **Note: The key difference between var and let is not in the syntax, but it differs in the semantics.**

### **Example**

1. let greeter = "hey hi";
2. let times = 5;
3. if (times **>** 3) {
4. let hello = "Say Hello JavaTpoint";
5. console.log(hello) // Output: Say Hello JavaTpoint
6. }
7. console.log(hello) // Compile error: greeter is not defined

**Output:**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

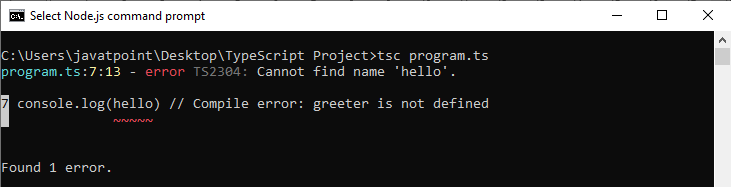
Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

The above code snippet throws an error because the variable "**hello**" is not defined globally.



## Var vs. Let Keyword

|  |  |  |
| --- | --- | --- |
| **SN** | **var** | **let** |
| 1. | The var keyword was introduced with JavaScript. | The let keyword was added in ES6 (ES 2015) version of JavaScript. |
| 2. | It has global scope. | It is limited to block scope. |
| 3. | It can be declared globally and can be accessed globally. | It can be declared globally but cannot be accessed globally. |
| 4. | Variable declared with var keyword can be re-declared and updated in the same scope. **Example:**  function varGreeter(){  var a = 10;  var a = 20; //a is replaced  console.log(a);  }  varGreeter(); | Variable declared with let keyword can be updated but not re-declared. **Example:**  function varGreeter(){  let a = 10;  let a = 20; //SyntaxError:  //Identifier 'a' has already been declared  console.log(a);  }  varGreeter(); |
| 5. | It is hoisted. **Example:**  {  console.log(c); // undefined.  //Due to hoisting  var c = 2;  } | It is not hoisted. **Example:**  {  console.log(b); // ReferenceError:  //b is not defined  let b = 3;  } |

# **TypeScript Operators**

An Operator is a symbol which operates on a value or data. It represents a specific action on working with data. The data on which operators operates is called operand. It can be used with one or more than one values to produce a single value. All of the standard JavaScript operators are available with the TypeScript program.

### **Example**

1. 10 + 10 = 20;

In the above example, the values '10' and '20' are known as an operand, whereas '+' and '=' are known as operators.

## Operators in TypeScript

In TypeScript, an operator can be classified into the following ways.

* [Arithmetic operators](https://www.javatpoint.com/typescript-operators#arithmetic-operator)
* [Comparison (Relational) operators](https://www.javatpoint.com/typescript-operators#comparison-operator)
* [Logical operators](https://www.javatpoint.com/typescript-operators#logical-operator)
* [Bitwise operators](https://www.javatpoint.com/typescript-operators#bitwise-operator)
* [Assignment operators](https://www.javatpoint.com/typescript-operators#assignment-operator)
* [Ternary/conditional operator](https://www.javatpoint.com/typescript-operators#ternary-operator)
* [Concatenation operator](https://www.javatpoint.com/typescript-operators#concatenation-operator)
* [Type Operator](https://www.javatpoint.com/typescript-operators#type-operator)

## Arithmetic Operators

Arithmetic operators take numeric values as their operands, performs an action, and then returns a single numeric value. The most common arithmetic operators are addition(+), subtraction(-), multiplication(\*), and division(/).

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Operator\_Name** | **Description** | **Example** |
| + | Addition | It returns an addition of the values. | let a = 20;  let b = 30;  let c = a + b;  console.log( c ); //  **Output**  30 |
| - | Subtraction | It returns the difference of the values. | let a = 30;  let b = 20;  let c = a - b;  console.log( c ); //  **Output**  10 |
| \* | Multiplication | It returns the product of the values. | let a = 30;  let b = 20;  let c = a \* b;  console.log( c ); //  **Output**  600 |
| / | Division | It performs the division operation, and returns the quotient. | let a = 100;  let b = 20;  let c = a / b;  console.log( c ); //  **Output**  5 |
| % | Modulus | It performs the division operation and returns the remainder. | let a = 95;  let b = 20;  let c = a % b;  console.log( c ); //  **Output**  15 |
| ++ | Increment | It is used to increments the value of the variable by one. | let a = 55;  a++;  console.log( a ); //  **Output**  56 |
| -- | Decrement | It is used to decrements the value of the variable by one. | let a = 55;  a--;  console.log( a ); //  **Output**  54 |

## Comparison (Relational) Operators

The comparison operators are used to compares the two operands. These operators return a Boolean value true or false. The important comparison operators are given below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Operator\_Name** | **Description** | **Example** |
| == | Is equal to | It checks whether the values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a==b); //false  console.log(a==10); //true  console.log(10=='10'); //true |
| === | Identical(equal and of the same type) | It checks whether the type and values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a===b); //false  console.log(a===10); //true  console.log(10==='10'); //false |
| != | Not equal to | It checks whether the values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a!=b); //true  console.log(a!=10); //false  console.log(10!='10'); //false |
| !== | Not identical | It checks whether the type and values of the two operands are equal or not. | let a = 10;  let b = 20;  console.log(a!==b); //true  console.log(a!==10); /false  console.log(10!=='10'); //true |
| > | Greater than | It checks whether the value of the left operands is greater than the value of the right operand or not. | let a = 30;  let b = 20;  console.log(a>b); //true  console.log(a>30); //false  console.log(20> 20'); //false |
| >= | Greater than or equal to | It checks whether the value of the left operands is greater than or equal to the value of the right operand or not. | let a = 20;  let b = 20;  console.log(a>=b); //true  console.log(a>=30); //false  console.log(20>='20'); //true |
| < | Less than | It checks whether the value of the left operands is less than the value of the right operand or not. | let a = 10;  let b = 20;  console.log(a<b); //true  console.log(a<10); //false  console.log(10<'10'); //false |
| <= | Less than or equal to | It checks whether the value of the left operands is less than or equal to the value of the right operand or not. | let a = 10;  let b = 20;  console.log(a<=b); //true  console.log(a<=10); //true  console.log(10<='10'); //true |

## Logical Operators

Logical operators are used for combining two or more condition into a single expression and return the Boolean result true or false. The Logical operators are given below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Operator\_Name** | **Description** | **Example** |
| && | Logical AND | It returns true if both the operands(expression) are true, otherwise returns false. | let a = false;  let b = true;  console.log(a&&b); /false  console.log(b&&true); //true  console.log(b&&10); //10 which is also 'true'  console.log(a&&'10'); //false |
| || | Logical OR | It returns true if any of the operands(expression) are true, otherwise returns false. | let a = false;  let b = true;  console.log(a||b); //true  console.log(b||true); //true  console.log(b||10); //true  console.log(a||'10'); //'10' which is also 'true' |
| ! | Logical NOT | It returns the inverse result of an operand(expression). | let a = 20;  let b = 30;  console.log(!true); //false  console.log(!false); //true  console.log(!a); //false  console.log(!b); /false  console.log(!null); //true |

## Bitwise Operators

The bitwise operators perform the bitwise operations on operands. The bitwise operators are as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Operator\_Name** | **Description** | **Example** |
| & | Bitwise AND | It returns the result of a Boolean AND operation on each bit of its integer arguments. | let a = 2;  let b = 3;  let c = a & b;  console.log(c); // **Output**  **2** |
| | | Bitwise OR | It returns the result of a Boolean OR operation on each bit of its integer arguments. | let a = 2;  let b = 3;  let c = a | b;  console.log(c); // **Output**  3 |
| ^ | Bitwise XOR | It returns the result of a Boolean Exclusive OR operation on each bit of its integer arguments. | let a = 2;  let b = 3;  let c = a ^ b;  console.log(c); //  **Output**  1 |
| ~ | Bitwise NOT | It inverts each bit in the operands. | let a = 2;  let c = ~ a;  console.log(c); //  **Output**  -3 |
| >> | Bitwise Right Shift | The left operand's value is moved to the right by the number of bits specified in the right operand. | let a = 2;  let b = 3;  let c = a >> b;  console.log(c); //  **Output**  0 |
| << | Bitwise Left Shift | The left operand's value is moved to the left by the number of bits specified in the right operand. New bits are filled with zeroes on the right side. | let a = 2;  let b = 3;  let c = a << b;  console.log(c); //  **Output**  16 |
| >>> | Bitwise Right Shift with Zero | The left operand's value is moved to the right by the number of bits specified in the right operand and zeroes are added on the left side. | let a = 3;  let b = 4;  let c = a >>> b;  console.log(c); //  **Output**  0 |

## Assignment Operators

Assignment operators are used to assign a value to the variable. The left side of the assignment operator is called a variable, and the right side of the assignment operator is called a value. The data-type of the variable and value must be the same otherwise the compiler will throw an error. The assignment operators are as follows.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Operator\_Name** | **Description** | **Example** |
| = | Assign | It assigns values from right side to left side operand. | let a = 10;  let b = 5;  console.log("a=b:" +a); //  **Output**  10 |
| += | Add and assign | It adds the left operand with the right operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a += b;  console.log(c); //  **Output**  15 |
| -= | Subtract and assign | It subtracts the right operand from the left operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a -= b;  console.log(c); //  **Output**  5 |
| \*= | Multiply and assign | It multiplies the left operand with the right operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a \*= b;  console.log(c); //  **Output**  50 |
| /= | Divide and assign | It divides the left operand with the right operand and assigns the result to the left side operand. | let a = 10;  let b = 5;  let c = a /= b;  console.log(c); //  **Output**  2 |
| %= | Modulus and assign | It divides the left operand with the right operand and assigns the result to the left side operand. | let a = 16;  let b = 5;  let c = a %= b;  console.log(c); //  **Output**  1 |

## Ternary/Conditional Operator

The conditional operator takes three operands and returns a Boolean value based on the condition, whether it is true or false. Its working is similar to an if-else statement. The conditional operator has right-to-left associativity. The syntax of a conditional operator is given below.

1. expression ? expression-1 : expression-2;

* **expression:** It refers to the conditional expression.
* **expression-1:** If the condition is true, expression-1 will be returned.
* **expression-2:** If the condition is false, expression-2 will be returned.

### **Example**

1. let num = 16;
2. let result = (num **>** 0) ? "True":"False"
3. console.log(result);

**Output:**

True

## Concatenation Operator

The concatenation (+) operator is an operator which is used to append the two string. In concatenation operation, we cannot add a space between the strings. We can concatenate multiple strings in a single statement. The following example helps us to understand the concatenation operator in TypeScript.

### **Example**

1. let message = "Welcome to " + "JavaTpoint";
2. console.log("Result of String Operator: " +message);

**Output:**

Result of String Operator: Welcome to JavaTpoint

## Type Operators

There are a collection of operators available which can assist you when working with objects in TypeScript. Operators such as typeof, instanceof, in, and delete are the examples of Type operator. The detail explanation of these operators is given below.

|  |  |  |
| --- | --- | --- |
| **Operator\_Name** | **Description** | **Example** |
| in | It is used to check for the existence of a property on an object. | let Bike = {make: 'Honda', model: 'CLIQ', year: 2018};  console.log('make' in Bike); //  **Output:**  true |
| delete | It is used to delete the properties from the objects. | let Bike = { Company1: 'Honda',  Company2: 'Hero',  Company3: 'Royal Enfield'  };  delete Bike.Company1;  console.log(Bike); //  **Output:**  { Company2: 'Hero', Company3: 'Royal Enfield' } |
| typeof | It returns the data type of the operand. | let message = "Welcome to " + "JavaTpoint";  console.log(typeof message); //  **Output:**  String |
| instanceof | It is used to check if the object is of a specified type or not. | let arr = [1, 2, 3];  console.log( arr instanceof Array ); // true  console.log( arr instanceof String ); // false |

# **TypeScript Type Annotation**

We know that JavaScript is not a typed language so we cannot specify the type of a variable such as a number, string, Boolean in JavaScript. However, in TypeScript, we can specify the type of variables, function parameters, and object properties because TypeScript is a typed language.

Type Annotations are annotations which can be placed anywhere when we use a type. The use of Type annotation is not mandatory in TypeScript. It helps the compiler in checking the types of variable and avoid errors when dealing with the data types.

We can specify the type by using a **colon(: Type)** after a variable name, parameter, or property. There can be a space between the colon and variable name, parameter, or property. TypeScript includes all the primitive data types of JavaScript such as number, string, Boolean, etc.

### **Syntax**

1. var variableName: TypeAnnotation = value;

The following example demonstrates type annotations for variables with different data types.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

1. var age: number = 44;          // number variable
2. var name: string = "Rahul";     // string variable
3. var isUpdated: boolean = true; // Boolean variable

In the above example, the variables are declared with their data type. These examples demonstrate type annotations. Here, we cannot change the value by using a different data type with the available data type. If we try to do this, TypeScript compiler will throw an error. For example, if we assign a **string** to a variable **age or number** to the **name**, then it will give a compilation error.

**Use of Type Annotation as a parameter**

The below example demonstrates the type annotation with parameters.

### **Example**

1. function display(id:number, name:string)
2. {
3. console.log("Id = " + id + ", Name = " + name);
4. }
5. display(101, "Rohit Sharma");

**Output:**

Id = 101, Name = Rohit Sharma

## Inline Type Annotation

In TypeScript, inline type annotations allow us to declare an object for each of the properties of the object.

### **Syntax**

1. :{ /\*Structure\*/ }

### **Example**

1. var student : {
2. id: number;
3. name: string;
4. };
6. student = {
7. id: 100,
8. name : "John"
9. }

Here, we declare an object student with two properties "id" and "name" with the data type number and string, respectively. If we try to assign a string value to id, the TypeScript compiler will throw an error: Type of property are incompatible.

# **TypeScript Type Inference**

In TypeScript, it is not necessary to annotate type always. The TypeScript compiler infers the type information when there is no explicit information available in the form of type annotations.

In TypeScript, TypeScript compiler infers the type information when:

* Variables and members are initialized
* Setting default values for parameters
* Determined function return types

**For example**

1. let x = 3;

In the above, the type of the variable "x" infers in a number. The type inference takes place when initializing variables and members, setting parameter default values, and determining function return types.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

**Let us take another example.**

1. var x = "JavaTpoint";
2. var y = 501;
3. x = y; // Compile-time Error: Type 'number' is not assignable to type 'string'

In the above example, we get an error because while inferring types, TypeScript inferred the type of variable "x" as a string and variable "y" as a number. When we try to assign y to x, the compiler generates an error that a number type is not assignable to a string type.

## Best Common Type: Type Inference

Type inference is helpful in type-checking when there are no explicit type annotation is available. In type inference, there can be a situation where an object may be initialized with multiple types.

**For example**

1. let arr = [ 10, 20, null, 40 ];

In the above example, we have an array with values 10, 20, null, and, 30. Here, we have given two choices for the type of an **array: number and null**. The best common type algorithm picks the one which is compatible with all types, i.e., number and null.

**Let us take another example.**

1. let arr2 = [ 10, 20, "JavaTpoint" ];

In the above example, the array contains values of type number and string both. Now, the TypeScript compiler uses the most common type algorithm and picks the one which is compatible with all types. In such cases, the compiler treats the type as a union of all types in the array. Here, the type would be (string or number), which means that the array can hold either string values or numeric values.

The return type of a function is also inferred by the returning value. For example:

1. function sum(x: number, y: number )
2. {
3. return x + y;
4. }
5. let Addition: number = sum(10,20); // Correct
6. let str: string = sum(10,20); // Compiler Error

In the above example, the return type of the function **sum** is **number**. So, its result will be stored in a number type variable, not a string type variable.

# **TypeScript Type Assertion**

In TypeScript, type assertion is a mechanism which tells the compiler about the type of a variable. When TypeScript determines that the assignment is invalid, then we have an option to override the type using a type assertion. If we use a type assertion, the assignment is always valid, so we need to be sure that we are right. Otherwise, our program may not work correctly.

Type assertion is explicitly telling the compiler that we want to treat the entity as a different type. It allows us to treat **any** as a number, or number as a string. Type assertion is commonly used when we are migrating over code from JavaScript to TypeScript.

Type assertion works like typecasting, but it does not perform type checking or restructuring of data just like other languages can do like C# and Java. The typecasting comes with runtime support, whereas type assertion has no impact on runtime. However, type assertions are purely a compile-time construct and provide hints to the compiler on how we want our code to be analyzed.

### **Example**

1. let empCode: any = 111;
2. let employeeCode = **<number>** code;
3. console.log(typeof(employeeCode)); //Output: number

In the above example, we have declared a variable **empCode** as of type any. In the next line, we assign a value of this variable to another variable named **employeeCode**. Here, we know that empCode is of type number, even though we declared it as **'any.'** when we are assigning **empCode** to **employeeCode**, we have asserted that **empCode** is of type number. Now the type of **employeeCode** is **number**.

Backward Skip 10sPlay VideoForward Skip 10s

TypeScript provides two ways to do Type Assertion. They are

1. Using Angular Bracket <>
2. Using as keyword

### **Using Angular Bracket <>**

In TypeScript, we can use angular "**bracket <>**" to show Type Assertion.

**Example**

1. let empCode: any = 111;
2. let employeeCode = **<number>** code;

### **Using as Keyword**

TypeScript provides another way to show Type Assertion by using "**as**" keyword.

**Example**

1. let empCode: any = 111;
2. let employeeCode = code as number;

## Type Assertion with object

Sometimes, we might have a situation where we have an object which is declared without any properties. For this, the compiler gives an error. But, by using type assertion we can avoid this situation. We can understand it with the following example.

**Example**

1. let student = { };
2. student.name = "Rohit"; //Compiler Error: Property 'name' doesn?t exist on type '{}'
3. student.code = 123; //Compiler Error: Property 'code' doesn?t exist on type '{}'

In the above example, we will get a compilation error, because the compiler assumes that the type of student is {} without properties. We can avoid this situation by using a type assertion, which can be shown below.

1. interface Student {
2. name: string;
3. code: number;
4. }
5. let student = **<Student>** { };
6. student.name = "Rohit"; // Correct
7. student.code = 123; // Correct

In the above example, we have created an interface **Student** with the properties **name** and **code**. Then, we used type assertion on the student, which is the correct way to use type assertion.

# **TypeScript Arrays**

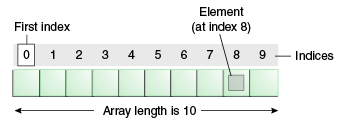
An array is a homogenous collection of similar type of elements which have a contiguous memory location.

An array is a user-defined data type.

An array is a type of data structure where we store the elements of a similar data type. In an array, we can store only a fixed set of elements. We can also use it as an object.

The array is index-based storage, where the first element stored at index 0. The below structure helps to understand the structure of an array.

Backward Skip 10sPlay VideoForward Skip 10s



## Characteristics of an Array

1. An array stores elements which have the same data type.
2. Array elements stored in contiguous memory locations.
3. The storage of 2-D array elements is rowed by row in a contiguous memory location.
4. Array name represents the address of the starting element.
5. The size of an array should be initialized at the declaration time.
6. Array size should be a constant expression and not a variable.
7. We can retrieve array elements by specifying the element's corresponding index value.

## Advantage

**Code Optimization:** An array helps to make the code optimized, which increases the speed and performance of the program. It allows us to retrieve or sort the array data more efficiently.

**Random access:** It provides the ability to access any data of an array in constant time (independent of its position and size). Thus, we can get any data of an array located at any index position directly.

## Disadvantage

**Size Limit:** An array allows us to store only the fixed number of elements. Once the array is declared, we cannot alter its size. Hence, if we want to insert more element than declared, it is not possible.

## Array declaration

Just like JavaScript, TypeScript also supports arrays. There are two ways to declare an array:

**1. Using square brackets.**

1. let array\_name[:datatype] = [val1,val2,valn..]

**Example:**

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

**2. Using a generic array type.**

1. let array\_name: Array**<elementType>** = [val1,val2,valn..]

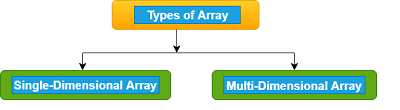
**Example:**

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

## Types of the array in TypeScript

There are two types of an array:

1. Single-Dimensional Array
2. Multi-Dimensional Array



### **Single-Dimensional Array**

A single-dimensional array is a type of linear array, which contains only one row for storing data. It has a single set of the square bracket ("[]"). We can access its elements either using row or column index.

**Syntax**

1. let array\_name[:datatype];

**Initialization**

1. array\_name = [val1,val2,valn..]

**Example**

1. let arr:number[];
2. arr = [1, 2, 3, 4]
3. console.log("Array[0]: " +arr[0]);
4. console.log("Array[1]: " +arr[1]);

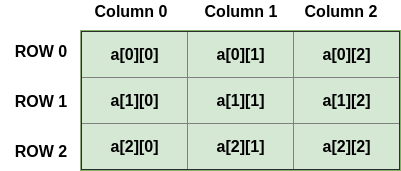
**Output:**

Array[0]: 1

Array[1]: 2

### **Multi-Dimensional Array**

A multi-dimensional array is an array which contains one or more arrays. In the multi-dimensional array, data is stored in a row and column-based index (also known as matrix form). A two-dimensional array (2-D array) is the simplest form of a multi-dimensional array.



**Syntax**

1. let arr\_name:datatype[][] = [ [a1,a2,a3], [b1,b2,b3] ];

**Initialization**

1. let arr\_name:datatype[initial\_array\_index][referenced\_array\_index] = [ [val1,val2,val 3], [v1,v2,v3]];

**Example**

1. var mArray:number[][] = [[1,2,3],[5,6,7]] ;
2. console.log(mArray[0][0]);
3. console.log(mArray[0][1]);
4. console.log(mArray[0][2]);
5. console.log();
6. console.log(mArray[1][0]);
7. console.log(mArray[1][1]);
8. console.log(mArray[1][2]);

**Output:**

1

2

3

5

6

7

## Array Object

Array objects allow us to store multiple values in a single variable. We can create an array by using the Array object. The Array constructor is used to pass the following arguments for array creation.

* A numeric value which represents the size of an array or
* A list of comma-separated values.

**Syntax**

1. let arr\_name:datatype[] = new Array(values);

**Example**

1. //array by using the Array object.
2. let arr:string[] = new Array("JavaTpoint","2200","Java","Abhishek");
3. for(var i = 0;i**<arr.length**;i++) {
4. console.log(arr[i]);
5. }

**Output:**

JavaTpoint

2200

Java

Abhishek

### **Array Traversal by using a for...in loop**

**Example**

1. let i:any;
2. let arr:string[] = ["JavaTpoint", "2300", "Java", "Abhishek"];
3. for(i in arr) {
4. console.log(arr[i])
5. }

**Output:**

JavaTpoint

2300

Java

Abhishek

### **Passing Arrays to Functions**

We can pass arrays to functions by specifying the array name without an index.

**Example**

1. let arr:string[] = new Array("JavaTpoint", "2300", "Java", "Abhishek");
2. //Passing arrays in function
3. function display(arr\_values:string[]) {
4. for(let i = 0;i**<arr\_values.length**;i++) {
5. console.log(arr[i]);
6. }
7. }
8. //Calling arrays in function
9. display(arr);

**Output:**

JavaTpoint

2300

Java

Abhishek

## TypeScript Spread operator

The spread operator is used to initialize arrays and objects from another array or object. We can also use it for object de-structuring. It is a part of the ES 6 version.

**Example**

1. let arr1 = [ 1, 2, 3];
2. let arr2 = [ 4, 5, 6];
3. //Create new array from existing array
4. let copyArray = [...arr1];
5. console.log("CopiedArray: " +copyArray);
6. //Create new array from existing array with more elements
7. let newArray = [...arr1, 7, 8];
8. console.log("NewArray: " +newArray);
9. //Create array by merging two arrays
10. let mergedArray = [...arr1, ...arr2];
11. console.log("MergedArray: " +mergedArray);

**Output:**

CopiedArray: 1,2,3

NewArray: 1,2,3,7,8

MergedArray: 1,2,3,4,5,6

## Array Methods

The list of array methods with their description is given below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1. | concat() | It is used to joins two arrays and returns the combined result. |
| 2. | copyWithin() | It copies a sequence of an element within the array. |
| 3. | every() | It returns true if every element in the array satisfies the provided testing function. |
| 4. | fill() | It fills an array with a static value from the specified start to end index. |
| 5. | indexOf() | It returns the index of the matching element in the array, otherwise -1. |
| 6. | includes() | It is used to check whether the array contains a certain element or not. |
| 7. | Join() | It is used to joins all elements of an array into a string. |
| 8. | lastIndexOf() | It returns the last index of an element in the array. |
| 9. | Pop() | It is used to removes the last elements of the array. |
| 10. | Push() | It is used to add new elements to the array. |
| 11. | reverse() | It is used to reverse the order of an element in the array. |
| 12. | Shift() | It is used to removes and returns the first element of an array. |
| 13. | slice() | It returns the section fo an array in the new array. |
| 14. | sort() | It is used to sort the elements of an array. |
| 15. | splice() | It is used to add or remove the elements from an array. |
| 16. | toString() | It returns the string representation of an array. |
| 17. | unshift() | It is used to add one or more elements to the beginning of an array. |

# **TypeScript Tuples**

We know that an array holds multiple values of the same data type. But sometimes, we may need to store a collection of values of different data types in a single variable. Arrays will not provide this feature, but TypeScript has a data type called Tuple to achieve this purpose. A Tuple is an array which store multiple fields belong to different data types. It is similar to the structures in the C programming language.

A tuple is a data type which can be used like any other variables. It represents the heterogeneous collection of values and can also be passed as parameters in a function call.

In abstract mathematics, the term tuple is used to denote a multi-dimensional coordinate system. JavaScript does not have tuple as data type, but tuples are available in TypeScript. The order of elements in a tuple is important.

### **Syntax**

1. let tuple\_name = [val1,val2,val3, ...val n];

### **Example**

1. let arrTuple = [101, "JavaTpoint", 105, "Abhishek"];
2. console.log(arrTuple);

**Output:**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

[101, 'JavaTpoint', 105, 'Abhishek']

We can also declare and initialize a tuple separately by initially declaring the tuple as an empty tuple in Typescript.

### **Example**

1. let arrTuple = [];
2. arrTuple[0] = 101
3. arrTuple[1] = 105

## Accessing tuple Elements

We can read or access the fields of a tuple by using the index, which is the same as an array. In Tuple, the index starts from zero.

### **Example**

1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
2. console.log("Name of the Employee is : "+empTuple [0]);
3. console.log("Age of the Employee is : "+empTuple [1]);
4. console.log(empTuple [0]+" is working in "+empTuple [2]);

**Output:**

Name of the Employee is: Rohit Sharma

Age of the Employee is: 25

Rohit Sharma is working in JavaTpoint

## Operations on Tuple

A tuple has two operations:

1. Push()
2. Pop()

### **Push()**

The push operation is used to add an element to the tuple.

**Example**

1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
2. console.log("Items: "+empTuple);
3. console.log("Length of Tuple Items before push: "+empTuple.length);   // returns the tuple size
4. empTuple.push(10001);   // append value to the tuple
5. console.log("Length of Tuple Items after push: "+empTuple.length);
6. console.log("Items: "+empTuple);

**Output:**

Items: Rohit Sharma, 25, JavaTpoint

Length of Tuple Items before push: 3

Length of Tuple Items after push: 4

Items: Rohit Sharma, 25, JavaTpoint, 10001

### **Pop()**

The pop operation is used to remove an element from the tuple.

### **Example**

1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint", 10001];
2. console.log("Items: "+empTuple);
3. console.log("Length of Tuple Items before pop: "+empTuple.length);   // returns the tuple size
4. empTuple.pop();   // removed value to the tuple
5. console.log("Length of Tuple Items after pop: "+empTuple.length);
6. console.log("Items: "+empTuple);

**Output:**

Items: Rohit Sharma,25, JavaTpoint, 10001

Length of Tuple Items before pop: 4

Length of Tuple Items after pop: 3

Items: Rohit Sharma, 25, JavaTpoint

## Update or Modify the Tuple Elements

Tuples are mutable, which means we can update or change the values of tuple elements. To modify the fields of a Tuple, we need to use the index of the fields and assignment operator. We can understand it with the following example.

### **Example**

1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
2. empTuple[1] = 30;
3. console.log("Name of the Employee is: "+empTuple [0]);
4. console.log("Age of the Employee is: "+empTuple [1]);
5. console.log(empTuple [0]+" is working in "+empTuple [2]);

**Output:**

Name of the Employee is: Rohit Sharma

Age of the Employee is: 30

Rohit Sharma is working in JavaTpoint

## Clear the fields of a Tuple

We cannot delete the tuple variable, but its fields could be cleared. To clear the fields of a tuple, assign it with an empty set of tuple field, which is shown in the following example.

### **Example**

1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
2. empTuple = [];
3. console.log(empTuple);

**Output:**

[]

## Destructuring the Tuple

Destructuring allows us to break up the structure of an entity. TypeScript used destructuring in the context of a tuple.

### **Example**

1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
2. let [emp, student] = empTuple;
3. console.log(emp);
4. console.log(student);

**Output:**

Rohit Sharma

25

## Passing Tuple to Functions

We can pass a tuple to functions, which can be shown in the below example.

### **Example**

1. //Tuple Declaration
2. let empTuple = ["JavaTpoint", 101, "Abhishek"];
3. //Passing tuples in function
4. function display(tuple\_values:any[]) {
5. for(let i = 0;i**<empTuple.length**;i++) {
6. console.log(empTuple[i]);
7. }
8. }
9. //Calling tuple in function
10. display(empTuple);

**Output:**

JavaTpoint

101

Abhishek

# **TypeScript Union**

In TypeScript, we can define a variable which can have multiple types of values. In other words, TypeScript can combine one or two different types of data (i.e., number, string, etc.) in a single type, which is called a union type. Union types are a powerful way to express a variable with multiple types. Two or more data types can be combined by using the pipe ('|') symbol between the types.

### **Syntax**

1. (type1 | type2 | type3 | ........ | type-n)

### **Example**

1. let value: number|string;
2. value = 120;
3. console.log("The Numeric value of a value is: "+value);
4. value = "Welcome to JavaTpoint";
5. console.log("The String value of a value is: "+value);

**Output:**

The Numeric value of the value is: 120

The String value of the value is: Welcome to JavaTpoint

## Passing Union Type in Function Parameter

In function, we can pass a union type as a parameter. We can understand it from the below example.

### **Example**

1. function display(value: (number | string))
2. {
3. if(typeof(value) === "number")
4. console.log('The given value is of type number.');
5. else if(typeof(value) === "string")
6. console.log('The given value is of type string.');
7. }
8. display(123);
9. display("ABC");

**Output:**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

The given value is of type number.

The given value is of type of string.

## Passing Union Type to Arrays

TypeScript allows passing a union type to an array. We can understand it from the below example.

### **Example**

1. let arrType:number[]|string[];
2. let i:number;
3. arrType = [1,2,3,4];
4. console.log("Numeric type array:")
6. for(i = 0;i**<arrType.length**;i++){
7. console.log(arrType[i]);
8. }
9. arrType = ["India","America","England"];
10. console.log("String type array:")
12. for(i = 0;i**<arrType.length**;i++){
13. console.log(arrType[i]);
14. }

**Output:**

Numeric type array:

1

2

3

4

String type array:

India

America

England

## The union can replace enums.

Enums are used to create types that contain a list of constants. By default, enums have index values (0, 1, 2, 3, etc.). We can see the enums in the following example, which contains the list of colors.

### **Example**

1. export enum Color {RED, BLUE, WHITE}

Instead of enums, we can use union types and can get similar benefits in a much shorter way.

### **Example**

1. export type Color = 'red' | 'white' | 'blue';
2. const myColor: Color = 'red';
3. console.log(myColor.toUpperCase());

**Output:**

RED

# **TypeScript String**

In TypeScript, the string is an object which represents the sequence of character values. It is a primitive data type which is used to store text data. The string values are surrounded by single quotation mark or double quotation mark. An array of characters works the same as a string.

### **Syntax**

1. let var\_name = new String(string);

### **Example**

1. let uname = new String("Hello JavaTpoint");
2. console.log("Message: " +uname);
3. console.log("Length: "+uname.length);

**Output:**

Message: Hello JavaTpoint

Length: 16

There are three ways in which we can create a string.

### **1. Single quoted strings**

It enclosed the string in a single quotation mark, which is given below.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

**Example**

1. var studentName: String = 'Peter';

### **2. Double quoted strings**

It enclosed the string in double quotation marks, which is given below.

**Example**

1. var studentName: String = "Peter";

### **3. Back-ticks strings**

It is used to write an expression. We can use it to embed the expressions inside the string. It is also known as Template string. TypeScript supports Template string from ES6 version.

**Example**

1. let empName:string = "Rohit Sharma";
2. let compName:string = "JavaTpoint";
3. // Pre-ES6
4. let empDetail1: string = empName + " works in the " + compName + " company.";
5. // Post-ES6
6. let empDetail2: string = `${empName} works in the ${compName} company.`;
7. console.log("Before ES6: " +empDetail1);
8. console.log("After ES6: " +empDetail2);

**Output:**

Before ES6: Rohit Sharma works in the JavaTpoint company.

After ES6: Rohit Sharma works in the JavaTpoint company.

## Multi-Line String

ES6 provides us to write the multi-line string. We can understand it from the below example.

### **Example**

1. let multi = 'hello ' +
2. 'world ' +
3. 'my ' +
4. 'name ' +
5. 'is ' +
6. 'Rohit';

If we want that each line in the string contains "new line" characters, then we have to add **"\n"** at the end of each string.

### **Example**

1. let multi = ' hello\n ' +
2. 'JavaTpoint\n ' +
3. 'my\n ' +
4. 'name\n ' +
5. 'is\n ' +
6. 'Rohit Sharma';
7. console.log(multi);

**Output:**

hello

JavaTpoint

my

name

is

Rohit Sharma

## String Literal Type

A string literal is a sequence of characters enclosed in double quotation marks (" "). It is used to represent a sequence of character which forms a null-terminated string. It allows us to specify the exact string value specified in the "string literal type." It uses **"pipe" or " | "** symbol between different string value.

### **Syntax**

1. Type variableName = "value1" | "value2" | "value3"; // upto N number of values

String literal can be used in two ways-

### **1. Variable Assignment**

We can assign only allowed values to a literal type variable. Otherwise, it will give the compile-time error.

**Example**

1. type Pet = 'cat' | 'dog' | 'Rabbit';
2. let pet: Pet;
3. if(pet = 'cat'){
4. console.log("Correct");
5. };
6. if(pet = 'Deer')
7. {
8. console.log("compilation error");
9. };

**Output:**

Correct

compilation error

### **2. Function Parameter**

We can pass only defined values to literal type argument. Otherwise, it will give the compile-time error.

**Example**

1. type FruitsName = "Apple" | "Mango" | "Orange";
2. function showFruitName(fruitsName: FruitsName): void {
3. console.log(fruitsName);
4. }
5. showFruitName('Mango');   //OK - Print 'Mango'
6. //Compile Time Error
7. showFruitName('Banana');

**Output:**

Mango

Banana

## String Methods

The list of string methods with their description is given below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1. | charAt() | It returns the character of the given index. |
| 2. | concat() | It returns the combined result of two or more string. |
| 3. | endsWith() | It is used to check whether a string ends with another string. |
| 4. | includes() | It checks whether the string contains another string or not. |
| 5. | indexOf() | It returns the index of the first occurrence of the specified substring from a string, otherwise returns -1. |
| 6. | lastIndexOf() | It returns the index of the last occurrence of a value in the string. |
| 7. | match() | It is used to match a regular expression against the given string. |
| 8. | replace() | It replaces the matched substring with the new substring. |
| 9. | search() | It searches for a match between a regular expression and string. |
| 10. | slice() | It returns a section of a string. |
| 11. | split() | It splits the string into substrings and returns an array. |
| 12. | substring() | It returns a string between the two given indexes. |
| 13. | toLowerCase() | It converts the all characters of a string into lower case. |
| 14. | toUpperCase() | It converts the all characters of a string into upper case. |
| 15. | trim() | It is used to trims the white space from the beginning and end of the string. |
| 16. | trimLeft() | It is used to trims the white space from the left side of the string. |
| 17. | trimRight() | It is used to trims the white space from the right side of the string. |
| 18. | valueOf() | It returns a primitive value of the specified object. |

### **Example**

1. //String Initialization
2. let str1: string = 'Hello';
3. let str2: string = 'JavaTpoint';
4. //String Concatenation
5. console.log("Combined Result: " +str1.concat(str2));
6. //String charAt
7. console.log("Character At 4: " +str2.charAt(4));
8. //String indexOf
9. console.log("Index of T: " +str2.indexOf('T'));
10. //String replace
11. console.log("After Replacement: " +str1.replace('Hello', 'Welcome to'));
12. //String uppercase
13. console.log("UpperCase: " +str2.toUpperCase());

**Output:**

Combined Result: HelloJavaTpoint

Character At 4: T

Index of T: 4

After Replacement: Welcome to

UpperCase: JAVATPOINT

# **TypeScript Numbers**

Like JavaScript, all the numbers in TypeScript are stored as floating-point values. These numeric values are treated like a number data type. The number is used to represents both integers as well as floating-point values. The number type converts the numeric literal to an instance of the number class. The number class acts as a wrapper and manipulate the numeric literals as they were objects. TypeScript also supports Binary(Base 2), Octal(Base 8), Decimal(Base 10), and Hexadecimal(Base 16) literals.

### **Syntax**

1. let identifier: number = value;

### **Example**

1. let first: number = 12.0;             // number
2. let second: number = 0x37CF;          // hexadecimal
3. let third: number = 0o377 ;           // octal
4. let fourth: number = 0b111001;        // binary
6. console.log(first);           // 123
7. console.log(second);          // 14287
8. console.log(third);           // 255
9. console.log(fourth);          // 57

## Number Properties

The Number objects have the following set of properties:

|  |  |  |
| --- | --- | --- |
| **SN** | **Property\_Name** | **Description** |
| 1. | MAX\_VALUE | It returns the largest possible value of a number in JavaScript and can have 1.7976931348623157E+308. |
| 2. | MIN\_VALUE | It returns the smallest possible value of a number in JavaScript and can have 5E-324. |
| 3. | NEGATIVE\_INFINITY | It returns a value that is less than MIN\_VALUE. |
| 4. | POSITIVE\_INFINITY | It returns a value that is greater than MAX\_VALUE. |
| 5. | NaN | When some number calculation is not representable by a valid number, then TypeScript returns a value NaN. It is equal to a value that is not a number. |
| 6. | prototype | It is a static property of the Number object. It is used to assign new properties and methods to the Number object in the current document. |

### **Example**

1. console.log("Number Properties: ");
2. console.log("A number variable can hold maximum value: " + Number.MAX\_VALUE);
3. console.log("A number variable can hold minimum value: " + Number.MIN\_VALUE);
4. console.log("Value of Negative Infinity: " + Number.NEGATIVE\_INFINITY);
5. console.log("Value of Positive Infinity:" + Number.POSITIVE\_INFINITY);
6. console.log("Example of NaN: " +Math.sqrt(-5)); // NaN

**Output:**

Number Properties:

A number variable can hold maximum value: 1.7976931348623157e+308

A number variable can hold minimum value: 5e-324

Value of Negative Infinity: -Infinity

Value of Positive Infinity: Infinity

Example of NaN: NaN

## Number Methods

The list of Number methods with their description is given below.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

|  |  |  |
| --- | --- | --- |
| **SN** | **Method\_Name** | **Description** |
| 1. | toExponential() | It is used to return the exponential notation in string format. |
| 2. | toFixed() | It is used to return the fixed-point notation in string format. |
| 3. | toLocaleString() | It is used to convert the number into a local specific representation of the number. |
| 4. | toPrecision() | It is used to return the string representation in exponential or fixed-point to the specified precision. |
| 5. | toString() | It is used to return the string representation of the number in the specified base. |
| 6. | valueOf() | It is used to return the primitive value of the number. |

### **Example**

1. let myNumber: number = 12345;
2. let myNumber\_1: number = 12.8789;
3. let myNumber\_2: number = 12667.976;
4. let myNumber\_3: number = 12.5779;
5. let myNumber\_4: number = 1234;
6. let myNumber\_5 = new Number(123);
8. console.log("Number Method: toExponential()");
9. console.log(myNumber.toExponential());
10. console.log(myNumber.toExponential(2));
12. console.log("Number Method: toString()");
13. console.log(myNumber.toString());
14. console.log(myNumber.toString(4));
16. console.log("Number Method: toFixed()");
17. console.log(myNumber\_1.toFixed());
18. console.log(myNumber\_1.toFixed(3));
20. console.log("Number Method: toLocaleString()");
21. console.log(myNumber\_2.toLocaleString()); // returns in US English
23. console.log("Number Method: toPrecision()");
24. console.log(myNumber\_3.toPrecision(1));
25. console.log(myNumber\_3.toPrecision(3));
27. console.log("Number Method: tovalueOf()");
28. console.log(myNumber\_5)
29. console.log(myNumber\_5.valueOf())
30. console.log(typeof myNumber)

**Output:**

Number Method: toExponential()

1.2345e+4

1.23e+4

Number Method: toString()

12345

3000321

Number Method: toFixed()

13

12.879

Number Method: toLocaleString()

12,667.976

Number Method: toPrecision()

1e+1

12.6

Number Method: tovalueOf()

[Number: 123]

123

number

# **Decision Making**

The decision making in a programming language is similar to decision making in real life. In a programming language, the programmer uses decision making for specifying one or more conditions to be evaluated by the program. The decision making always returns the Boolean result true or false.

There are various types of Decision making in TypeScript:

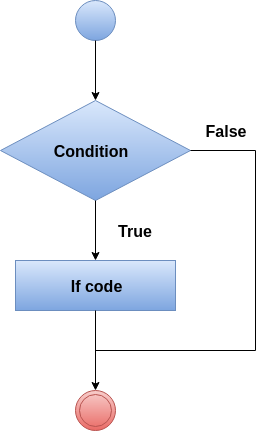
* if statement
* if-else statement
* if-else-if ladder
* nested if statement

## if statement

It is a simple form of decision making. It decides whether the statements will be executed or not, i.e., it checks the condition and returns true if the given condition is satisfied.

### **Syntax**

1. if(condition) {
2. // code to be executed
3. }



### **Example**

1. let a = 10, b = 20;
2. if (a **<** **b**)
3. {
4. console.log('a is less than b.');
5. }

**Output:**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

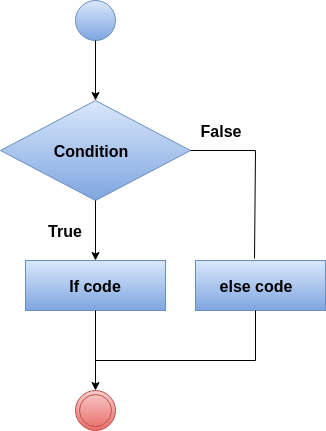
a is less than b.

## if-else statement

The if statement only returns the result when the condition is true. But if we want to returns something when the condition is false, then we need to use the if-else statement. The if-else statement tests the condition. If the condition is true, it executes if block and if the condition is false, it executes the else block.

### **Syntax**

1. if(condition) {
2. // code to be executed
3. } else {
4. // code to be executed
5. }



### **Example**

1. let n = 10
2. if (n **>** 0) {
3. console.log("The input value is positive Number: " +n);
4. } else {
5. console.log("The input value is negative Number: " +n);
6. }

**Output:**

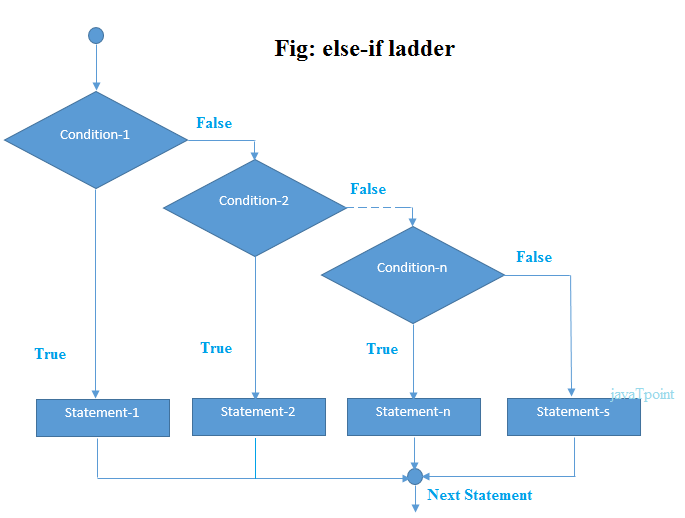
The input value is positive Number: 10

## if-else-if ladder

Here a user can take decision among multiple options. It starts execution in a top-down approach. When the condition gets true, it executes the associated statement, and the rest of the condition is bypassed. If it does not find any condition true, it returns the final else statement.

### **Syntax**

1. if(condition1){
2. //code to be executed if condition1 is true
3. }else if(condition2){
4. //code to be executed if condition2 is true
5. }
6. else if(condition3){
7. //code to be executed if condition3 is true
8. }
9. else{
10. //code to be executed if all the conditions are false
11. }



### **Example**

1. let marks = 95;
2. if(marks**<50**){
3. console.log("fail");
4. }
5. else if(marks**>**=50 && marks**<60**){
6. console.log("D grade");
7. }
8. else if(marks**>**=60 && marks**<70**){
9. console.log("C grade");
10. }
11. else if(marks**>**=70 && marks**<80**){
12. console.log("B grade");
13. }
14. else if(marks**>**=80 && marks**<90**){
15. console.log("A grade");
16. }else if(marks**>**=90 && marks**<100**){
17. console.log("A+ grade");
18. }else{
19. console.log("Invalid!");
20. }

**Output:**

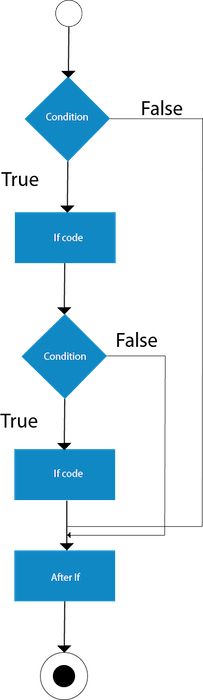
A+ grade

## Nested if statement

Here, the if statement targets another if statement. The nested if statement means if statement inside the body of another if or else statement.

### **Syntax**

1. if(condition1) {
2. //Nested if else inside the body of "if"
3. if(condition2) {
4. //Code inside the body of nested "if"
5. }
6. else {
7. //Code inside the body of nested "else"
8. }
9. }
10. else {
11. //Code inside the body of "else."
12. }



### **Example**

1. let n1 = 10, n2 = 22, n3 = 25
2. if (n1 **>**= n2) {
3. if (n1 **>**= n3) {
4. console.log("The largest number is: " +n1)
5. }
6. else {
7. console.log("The largest number is: " +n3)
8. }
9. }
10. else {
11. if (n2 **>**= n3) {
12. console.log("The largest number is: " +n2)
13. }
14. else {
15. console.log("The largest number is: " +n3)
16. }
17. }

**Output:**

The largest number is: 25

# **TypeScript Switch Statement**

The TypeScript switch statement executes one statement from multiple conditions. It evaluates an expression based on its value that could be Boolean, number, byte, short, int, long, enum type, string, etc. A switch statement has one block of code corresponding to each value. When the match is found, the corresponding block will be executed. A switch statement works like the if-else-if ladder statement.

**The following points must be remembered in a switch statement:**

* There can be N number of cases inside a switch statement.
* The case values must be unique.
* The case values must be constant.
* Each case statement has a break statement at the end of the code. The break statement is optional.
* The switch statement has a default block which is written at the end. The default statement is optional.

### **Syntax**

1. switch(expression){
3. case expression1:
4. //code to be executed;
5. break;  //optional
7. case expression2:
8. //code to be executed;
9. break;  //optional
10. ........
12. default:
13. //when no case is matched, this block will be executed;
14. break;  //optional
15. }

The switch statement contains the following things. There can be any number of cases inside a switch statement.

**Case:** The case should be followed by only one constant and then a semicolon. It cannot accept another variable or expression.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

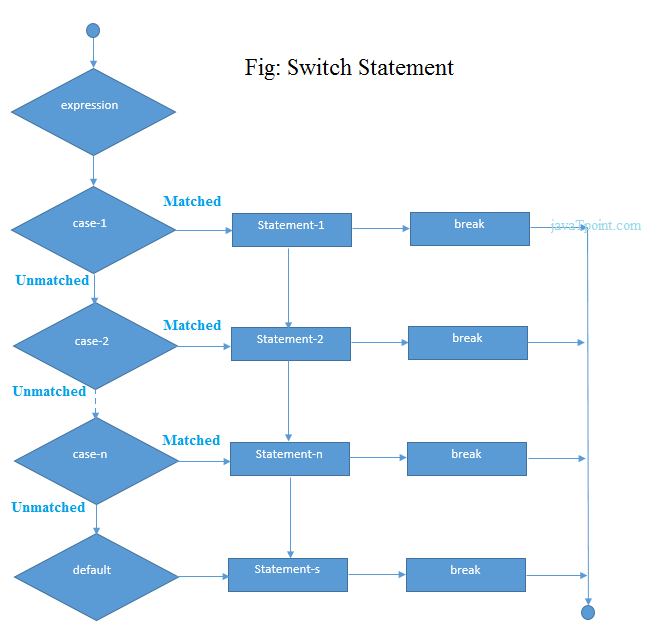
Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

**Break:** The break should be written at the end of the block to come out from the switch statement after executing a case block. If we do not write break, the execution continues with the matching value to the subsequent case block.

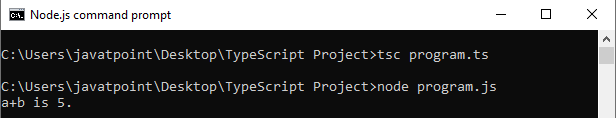
**Default:** The default block should be written at the end of the switch statement. It executes when there are no case will be matched.



### **Example**

1. let a = 3;
2. let b = 2;
4. switch (a+b){
5. case 1: {
6. console.log("a+b is 1.");
7. break;
8. }
9. case 2: {
10. console.log("a+b is 5.");
11. break;
12. }
13. case 3: {
14. console.log("a+b is 6.");
15. break;
16. }
18. default: {
19. console.log("a+b is 5.");
20. break;
21. }
22. }

**Output:**

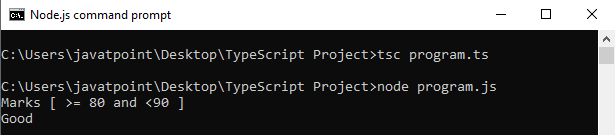


## Switch case with String

1. let grade: string = "A";
2. switch (grade)
3. {
4. case'A+':
5. console.log("Marks **>**= 90"+"\n"+"Excellent");
6. break;
8. case'A':
9. console.log("Marks [ **>**= 80 and **<90** ]"+"\n"+"Good");
10. break;
12. case'B+':
13. console.log("Marks [ **>**= 70 and **<80** ]"+"\n"+"Above Average");
14. break;
16. case'B':
17. console.log("Marks [ **>**= 60 and **<70** ]"+"\n"+"Average");
18. break;
20. case'C':
21. console.log("Marks **<** **60**"+"\n"+"Below Average");
22. break;
24. default:
25. console.log("Invalid Grade.");
26. }

In this example, we have a string variable grade. The switch statement evaluates grade variable value and match with case clauses and then execute its associated statements.

**Output:**



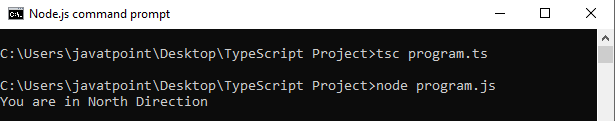
## Switch Case with Enum

In TypeScript, we can use the switch case with Enum in the following ways.

### **Example**

1. enum Direction {
2. East,
3. West,
4. North,
5. South
6. };
7. var dir: DirectionDirection = Direction.North;
9. function getDirection() {
10. switch (dir) {
11. case Direction.North: console.log('You are in North Direction');
12. break;
13. case Direction.East: console.log('You are in East Direction');
14. break;
15. case Direction.South: console.log('You are in South Direction');
16. break;
17. case Direction.West: console.log('You are in West Direction');
18. break;
19. }
20. }
21. getDirection();

**Output:**



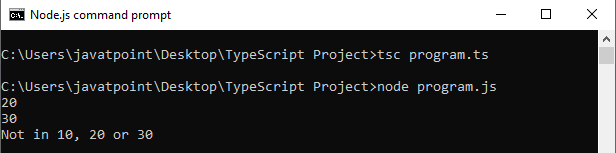
## TypeScript Switch Statement is fall-through.

The TypeScript switch statement is fall-through. It means if a break statement is not present, then it executes all statements after the first match case.

### **Example**

1. let number = 20;
2. switch(number)
3. {
4. //switch cases without break statements
5. case 10: console.log("10");
6. case 20: console.log("20");
7. case 30: console.log("30");
8. default: console.log("Not in 10, 20 or 30");
9. }

**Output:**



# **TypeScript Indefinite Loops**

In a programming language, loops are the sequence of instructions which continually repeated until a specific condition is not found. It makes the code compact. We can mostly use it with the array. Below is the general structure of the loop statement:

TypeScript Indefinite Loops

We can classify the loops into two types:

TypeScript Indefinite Loops

1. [Indefinite](https://www.javatpoint.com/typescript-indefinite-loops#indefinite)
2. [Definite](https://www.javatpoint.com/typescript-definite-loop)

## Indefinite Loop

In Indefinite loops, the number of iterations is not known before beginning the execution of the block of statements. There are two indefinite loops:

1. while loop
2. do-while loop

### **TypeScript while loop**

The TypeScript while loop iterates the elements for the infinite number of times. It executes the instruction repeatedly until the specified condition evaluates to true. We can use it when the number of iteration is not known. The while loop syntax is given below.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

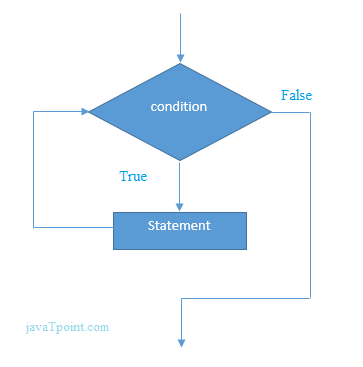
Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

**Syntax**

1. while (condition)
2. {
3. //code to be executed
4. }



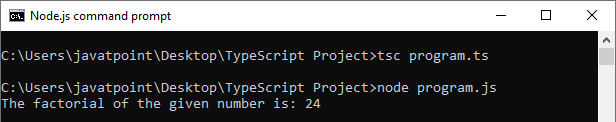
The explanation of while loop syntax is:

While loop starts the execution with checking the condition. If the condition evaluates to true, the loop body statement gets executed. Otherwise, the first statement following the loop gets executed. If the condition becomes false, the loops get terminated, which ends the life-cycle of the loops.

**Example**

1. let num = 4;
2. let factorial = 1;
4. while(num **>**=1) {
5. factorialfactorial = factorial \* num;
6. num--;
7. }
8. console.log("The factorial of the given number is: "+factorial);

**Output:**

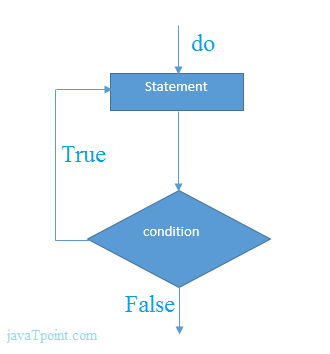


### **TypeScript do-while loop**

The TypeScript do-while loop iterates the elements for the infinite number of times similar to the while loop. But there is one difference from while loop, i.e., it gets executed at least once whether the condition is true or false. It is recommended to use do-while when the number of iteration is not fixed, and you have to execute the loop at least once. The do-while loop syntax is given below.

**Syntax**

1. do{
2. //code to be executed
3. }while (condition);



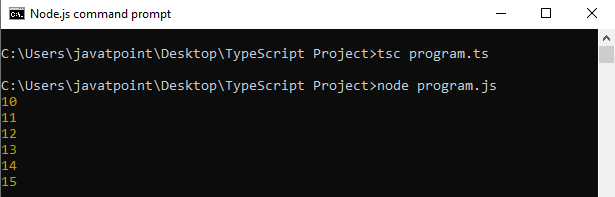
The explanation of do-while loop syntax is:

The do-while loop starts executing the statement without checking any condition for the first time. After the execution of the statement and update of the variable value, it starts evaluating the condition. If the condition is true, the next iteration of the loop starts execution. If the condition becomes false, the loops get terminated, which ends the life-cycle of the loops.

**Example**

1. let n = 10;
3. do {
4. console.log(n);
5. n++;
6. } while(n**<**=15);

**Output:**



# **TypeScript Definite Loop**

In this loop, we know about the number of iterations before the execution of the block of statements. A "**for loop**" is the best example of this loop. Here, we are going to discuss three types of the loop:

1. for loop
2. for..of loop
3. for..in loop

## TypeScript for loop

A for loop is a **repetition** control structure. It is used to execute the block of code to a specific number of times. A for statement contains the initialization, condition and increment/decrement in a single line which provides a shorter, and easy to debug structure of looping. The syntax of for loop is given below.

**Syntax**

1. for (first expression; second expression; third expression ) {
2. // statements to be executed repeatedly
3. }

Explanation of the flow of control in a "**for loop**" is:

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

The first expression is the **initialization** step, which is executed first, and only once. It allows us to declare and initialize the loop control variables.

The next expression evaluates the **condition**. If it is true, the body of the loop gets executed. If it is false, the loop does not execute, and the flow of control jumps to the next statement just after the "**for**" loop.

When the body of the "**for loop**" executes, the flow of control jumps to the increment/decrement statement. It allows us to update the loop control variables. It can be left blank, as long as a semicolon appears after the condition.

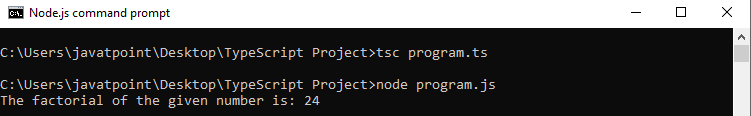
Now, the condition is **re-evaluated**. If it finds true, the loop executes, and the process repeats. When the condition becomes false, the "**for loop**" terminates which marks the end of the life-cycle.

TypeScript Definite Loop

**Example**

1. let num = 4;
2. let factorial = 1;
3. while (num **>**= 1) {
4. factorialfactorial = factorial \* num;
5. num--;
6. }
7. console.log("The factorial of the given number is: " + factorial);

**Output:**



## TypeScript for..of loop

The for..of loop is used to iterate and access the elements of an array, string, set, map, list, or tuple collection. The syntax of the for..of loop is given below.

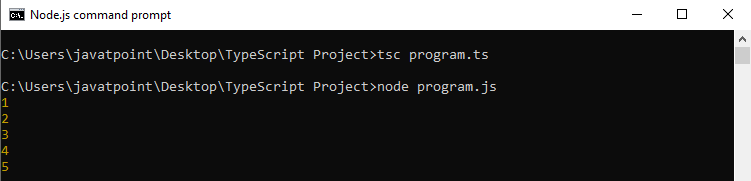
**Syntax**

1. for (var val of list) {
2. //statements to be executed
3. }

**Example**

1. let arr = [1, 2, 3, 4, 5];
3. for (var val of arr) {
4. console.log(val);
5. }

**Output:**



## TypeScript for..in loop

The for..in loop is used with an array, list, or tuple. This loop iterates through a list or collection and returns an index on each iteration. In this, the data type of "**val**" should be a string or any. The syntax of the for..in loop is given below.

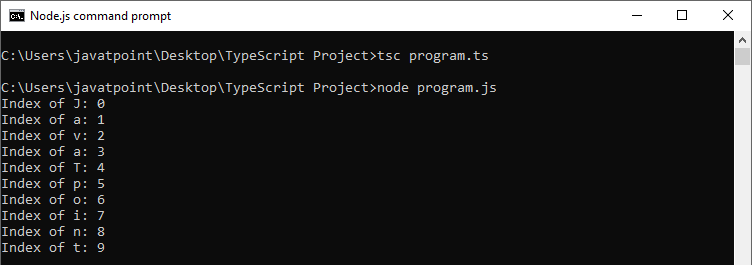
**Syntax**

1. for (var val in list) {
2. //statements
3. }

**Example**

1. let str:any = "JavaTpoint";
3. for (let index in str) {
4. console.log('Index of ${str[index]}: ${index}');
5. }

**Output:**



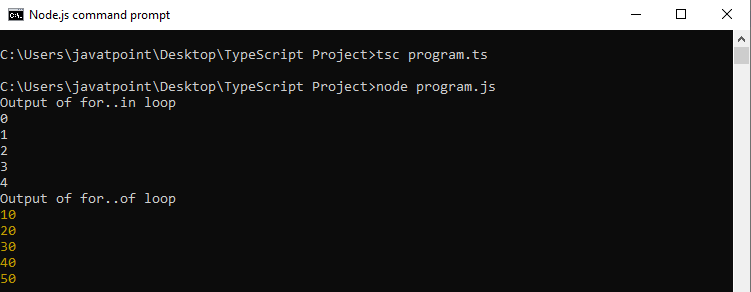
## for..of Vs. for..in Loop

Both the loops iterate over the lists, but their kind of iteration is different. The **for..in** loop returns a list of indexes on the object being iterated, whereas the **for..of** loop returns a list of values of the object being iterated.

Below example demonstrates these differences:

1. let myArray = [10, 20, 30, 40, 50,];
2. console.log("Output of for..in loop ");
3. for (let index in myArray) {
4. console.log(index);
5. }
6. console.log("Output of for..of loop ");
7. for (let val of myArray) {
8. console.log(val);
9. }

**Output:**



# **TypeScript Enums**

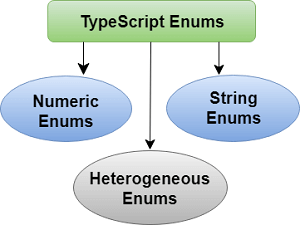
Enums stands for **Enumerations**. Enums are a new data type supported in TypeScript. It is used to define the set of **named constants**, i.e., a collection of related values. TypeScript supports both **numeric** and **string-based** enums. We can define the enums by using the **enum** keyword.

## Why Enums?

Enums are useful in TypeScript because of the following:

* It makes it easy to change values in the future.
* It reduces errors which are caused by transporting or mistyping a number.
* It exists only during compilation time, so it does not allocate memory.
* It saves runtime and compile-time with inline code in JavaScript.
* It allows us to create constants that we can easily relate to the program.
* It will enable developers to develop memory-efficient custom constants in JavaScript, which does not support enums, but TypeScript helps us to access them.

There are **three** types of Enums in TypeScript. These are:



* Numeric Enums
* String Enums
* Heterogeneous Enums

## Numeric Enums

Numeric enums are **number-based** enums, which store values as numbers. It means we can assign the number to an instance of the enum.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

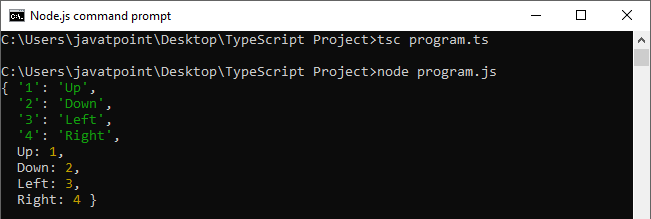
Backward Skip 10sPlay VideoForward Skip 10s

**Example**

1. enum Direction {
2. Up = 1,
3. Down,
4. Left,
5. Right,
6. }
7. console.log(Direction);

In the above example, we have a numeric enum named **Direction**. Here, we initialize **Up** with 1, and all of the following members are **auto-incremented** from that point. It means Direction.Up has the value 1, **Down** has 2, **Left** has 3, and **Right** has 4.

**Output:**

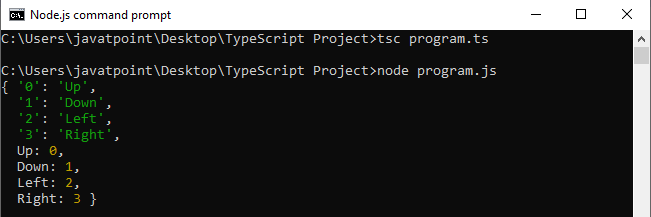


According to our need, it also allows us to leave off the initialization of enumeration. We can declare the enum without initialization as below.

1. enum Direction {
2. Up,
3. Down,
4. Left,
5. Right,
6. }
8. console.log(Direction);

Here, Up have the value 0, and all of the following members are auto-incremented from that point. It means Direction.Up has the value 0, Down has 1, Left has 2, and Right has 3. The auto-incrementing behavior is useful when there is no need to care about the member values themselves. But each value must be **distinct** from other values in the same enum.

**Output:**

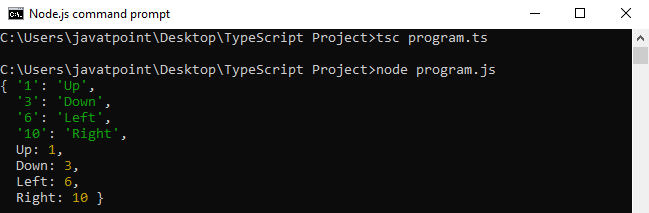


In TypeScript enums, it is not necessary to assign **sequential** values to enum members always. We can provide any values to the enum members, which looks like the below example.

**Example**

1. enum Direction {
2. Up=1,
3. Down=3,
4. Left=6,
5. Right=10,
6. }
7. console.log(Direction);

**Output:**



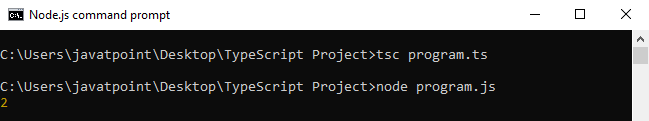
## Enum as a function argument

We can also use an enum as a **function type** or **return type**, which we can see in the below example.

1. enum AppStatus {
2. ACTIVE,
3. INACTIVE,
4. ONHOLD
5. }
6. function checkStatus(status: AppStatus): void {
7. console.log(status);
8. }
9. checkStatus(AppStatus.ONHOLD);

In the above example, we have declared an enum **AppStatus**. Next, we create a function **checkStatus()** that takes an input parameter status which returns an enum AppStatus. In the function, we **check** for the type of status. If status name matches, we get the matched enum member.

**Output:**



Here, we can see that the value printed '2' in the last statement is not much useful in most of the scenarios. That's why it is **preferred** and **recommended** to use **string-based** enums.

## String Enums

String enums are a similar concept to numeric enums, except that the enum has some subtle runtime differences. In a string enum, each enum values are **constant-initialized** with a string literal, or with another string enum member rather than numeric values.

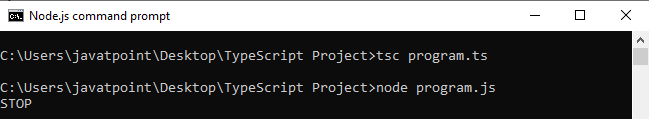
String enums do not have **auto-incrementing** behavior. The benefits of using this enum is that string enums provides better **readability**. If we were debugging a program, string enums allow us to give a meaningful and readable value when our code runs, independent of the name of the enum member itself.

Consider the following example of a numeric enum, but it is represented as a string enum:

**Example**

1. enum AppStatus {
2. ACTIVE = 'ACT',
3. INACTIVE = 'INACT',
4. ONHOLD = 'HLD',
5. ONSTOP = 'STOP'
6. }
7. function checkStatus(status: AppStatus): void {
9. console.log(status);
10. }
11. checkStatus(AppStatus.ONSTOP);

**Output:**



In the above example, we have declared a string enum **AppStatus** with the same values as the numeric enum above. But string enum is different from numeric enum where string enum values are initialized with **string literals**. The difference between these enums is that the numeric enum values are auto-incremented, whereas string enum values need to be initialized individually.

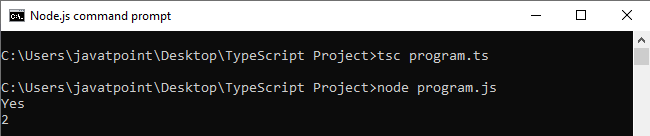
## Heterogeneous Enums

The heterogeneous enums are enums, which contains both **string** and **numeric** values. But it is advised that you don't do this unless there is a need to take advantage of JavaScript runtime behavior.

**Example**

1. enum AppStatus {
2. ACTIVE = 'Yes',
3. INACTIVE = 1,
4. ONHOLD = 2,
5. ONSTOP = 'STOP'
6. }
7. console.log(AppStatus.ACTIVE);
8. console.log(AppStatus.ONHOLD);

**Output:**



## Computed and constant members

We know that each enum members has a value associated with it. These values can be either constant or computed. We can consider enum member as **constant** if:

**1.** It is the first member of the enum and has no initializer value. In this case, it is assigned the value 0.

**Example**

1. // Name.Abhishek is constant:
2. enum Name {
3. Abhishek
4. }
5. console.log(Name);

**2.** It has no initializer value, and the preceding enum member is a numeric constant. In this case, the value of the current enum member will be the value of the preceding enum member plus one.

1. // All enum members in 'Name' and 'Profile' are constant.
2. enum Name {
3. Abhishek,
4. Ravi,
5. Ajay
6. }
7. enum Profile {
8. Engineer=1,
9. Leader,
10. Businessman
11. }

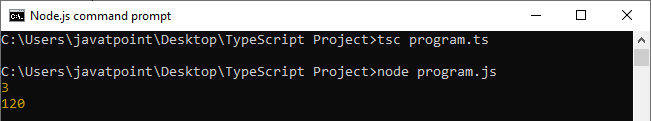
In TypeScript, we can say that an expression is a constant enum expression if it is:

* A literal enum expression.
* A reference to the previously defined constant enum member.
* A parenthesized constant enum expression.
* It is one of the +, -, ~ unary operators which is applied to constant enum expression.
* +, -, \*, /, %, <<, >>, >>>, &, |, ^ binary operators with constant enum expressions as operands.

In all other cases, the enum member is considered **computed**. The following enum example includes enum members with computed values.

1. enum Weekend {
2. Friday = 1,
3. Saturday = getDate('Dominoz'),
4. Sunday = Saturday \* 40
5. }
7. function getDate(day : string): number {
8. if (day === 'Dominoz') {
9. return 3;
10. }
11. }
12. console.log(Weekend.Saturday);
13. console.log(Weekend.Sunday);

**Output:**



## Reverse mapping

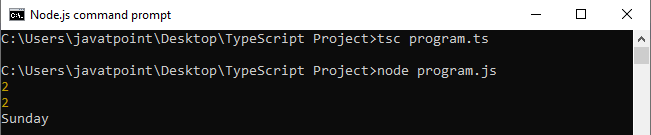
TypeScript enums also support reverse mapping. It means we can access the value of an enum member, and also can access a member name from its value. We can understand the reverse mapping from the below example.

#### Note:**The string enum does not support reverse mapping.**

**Example**

1. enum Weekend {
2. Friday = 1,
3. Saturday,
4. Sunday
5. }
6. console.log(Weekend.Saturday);
7. console.log(Weekend["Saturday"]);
8. console.log(Weekend[3]);

**Output:**



## Enums at runtime

Enums are the real objects which exist at runtime. We can understand it from the below example.

1. enum E {
2. A, B, C
3. }

It can actually be passed around to functions, which we can see in the below example.

1. function f(obj: { A: number }) {
2. return obj.A;
3. }
4. // Works, since 'E' has a property named 'A' which is a number.
5. f(E);

## Ambient enums

We can use ambient enums for describing the **shape** of already existing enum types.

1. Declare enum Enum{
2. X=1,
3. Y,
4. Z=2
5. }

There is mainly one **difference between ambient and non-ambient enums**. In regular enums, members that do not have an initializer is considered as **constant** if its preceding enum member is considered constant. But, an ambient (and non-const) enum member that does not have initializer is always considered **computed** enums.

# **TypeScript forEach**

The forEach() method is an array method which is used to execute a function on **each item in an array**. We can use it with the JavaScript data types like Arrays, Maps, Sets, etc. It is a useful method for displaying elements in an array.

### **Syntax**

We can declare the forEach() method as below.

1. array.forEach(callback[, thisObject]);

The forEach() method executes the provided **callback** once for each element present in the array in **ascending order**.

### **Parameter Details**

**1. callback:** It is a function used to test for each element. The callback function accepts **three arguments**, which are given below.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

* **Element value:** It is the current value of the item.
* **Element index:** It is the index of the current element processed in the array.
* **Array:** It is an array which is being iterated in the forEach() method.

#### Note:**These three arguments are optional.**

**2. thisObject:** It is an object to use as this when executing the callback.

### **Return Value**

It will return the created array.

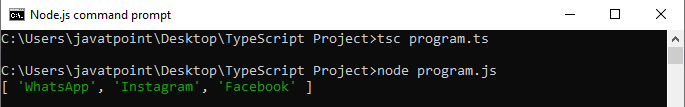
### **Example with string**

1. let apps = ['WhatsApp', 'Instagram', 'Facebook'];
2. let playStore = [];
4. apps.forEach(function(item){
5. playStore.push(item)
6. });
8. console.log(playStore);

The corresponding JavaScript code is:

1. var apps = ['WhatsApp', 'Instagram', 'Facebook'];
2. var playStore = [];
3. apps.forEach(function (item) {
4. playStore.push(item);
5. });
6. console.log(playStore);

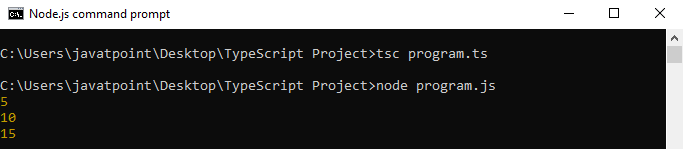
**Output:**



### **Example with number**

1. var num = [5, 10, 15];
2. num.forEach(function (value) {
3. console.log(value);
4. });

**Output:**



### **Disadvantage of forEach()**

The following are the disadvantages of the use of the forEach() method:

1. It does not provide a way to stop or **break** the forEach() loop.
2. It only **works** with arrays.

# **TypeScript Map**

TypeScript map is a new data structure added in **ES6** version of JavaScript. It allows us to store data in a **key-value pair** and remembers the original **insertion order** of the keys similar to other programming languages. In TypeScript map, we can use any value either as a **key** or as a **value**.

## Create Map

We can create a map as below.

1. var map = new Map();

## Map methods

The TypeScript map methods are listed below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Methods** | **Descriptions** |
| **1.** | map.set(key, value) | It is used to add entries in the map. |
| **2.** | map.get(key) | It is used to retrieve entries from the map. It returns undefined if the key does not exist in the map. |
| **3.** | map.has(key) | It returns true if the key is present in the map. Otherwise, it returns false. |
| **4.** | map.delete(key) | It is used to remove the entries by the key. |
| **5.** | map.size() | It is used to returns the size of the map. |
| **6.** | map.clear() | It removes everything from the map. |

**Example**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

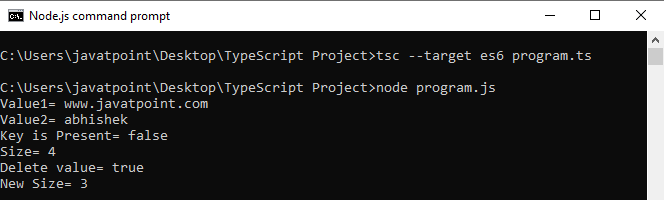
Backward Skip 10sPlay VideoForward Skip 10s

We can understand the map methods from the following example.

1. let map = new Map();
3. map.set('1', 'abhishek');
4. map.set(1, 'www.javatpoint.com');
5. map.set(true, 'bool1');
6. map.set('2', 'ajay');
8. console.log( "Value1= " +map.get(1)   );
9. console.log("Value2= " + map.get('1') );
10. console.log( "Key is Present= " +map.has(3) );
11. console.log( "Size= " +map.size );
12. console.log( "Delete value= " +map.delete(1) );
13. console.log( "New Size= " +map.size );

**Output:**

When we execute the above code snippet, it returns the following output.



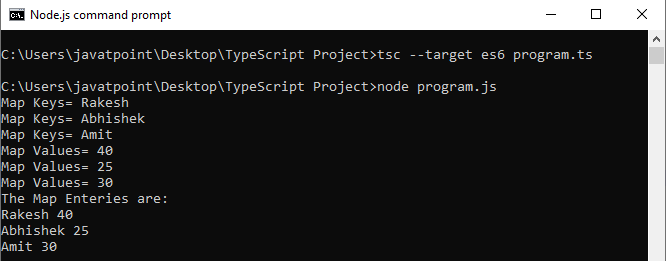
## Iterating Map Data

We can iterate over map keys or values or entries by using '**for...of**' loop. The following example helps to understand it more clearly.

**Example**

1. let ageMapping = new Map();
3. ageMapping.set("Rakesh", 40);
4. ageMapping.set("Abhishek", 25);
5. ageMapping.set("Amit", 30);
7. //Iterate over map keys
8. for (let key of ageMapping.keys()) {
9. console.log("Map Keys= " +key);
10. }
11. //Iterate over map values
12. for (let value of ageMapping.values()) {
13. console.log("Map Values= " +value);
14. }
15. console.log("The Map Enteries are: ");
16. //Iterate over map entries
17. for (let entry of ageMapping.entries()) {
18. console.log(entry[0], entry[1]);
19. }

**Output:**



# **TypeScript Set**

TypeScript set is a new data structure added in **ES6** version of JavaScript. It allows us to store **distinct data** (each value occur only once) into the **List** similar to other programming languages. Sets are a bit similar to **maps**, but it stores only **keys**, not the **key-value** pairs.

### **Create Set**

We can create a **set** as below.

1. let mySet = new Set();

### **Set methods**

The TypeScript set methods are listed below.

|  |  |  |
| --- | --- | --- |
| **SN** | **Methods** | **Descriptions** |
| **1.** | set.add(value) | It is used to add values in the set. |
| **2.** | set.has(value) | It returns true if the value is present in the set. Otherwise, it returns false. |
| **3.** | set.delete() | It is used to remove the entries from the set. |
| **4.** | set.size() | It is used to returns the size of the set. |
| **5.** | set.clear() | It removes everything from the set. |

**Example**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

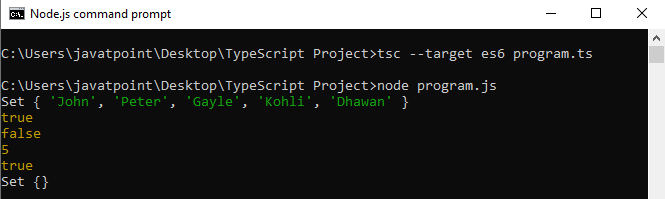
Backward Skip 10sPlay VideoForward Skip 10s

We can understand the set methods from the following example.

1. let studentEntries = new Set();
3. //Add Values
4. studentEntries.add("John");
5. studentEntries.add("Peter");
6. studentEntries.add("Gayle");
7. studentEntries.add("Kohli");
8. studentEntries.add("Dhawan");
10. //Returns Set data
11. console.log(studentEntries);
13. //Check value is present or not
14. console.log(studentEntries.has("Kohli"));
15. console.log(studentEntries.has(10));
17. //It returns size of Set
18. console.log(studentEntries.size);
20. //Delete a value from set
21. console.log(studentEntries.delete("Dhawan"));
23. //Clear whole Set
24. studentEntries.clear();
26. //Returns Set data after clear method.
27. console.log(studentEntries);

**Output:**

When we execute the above code snippet, it returns the following output.



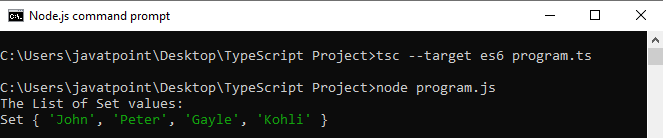
### **Chaining of Set Method**

TypeScript set method also allows the chaining of **add()** method. We can understand it from the below example.

**Example**

1. let studentEntries = new Set();
3. //Chaining of add() method is allowed in TypeScript
4. studentEntries.add("John").add("Peter").add("Gayle").add("Kohli");
6. //Returns Set data
7. console.log("The List of Set values:");
8. console.log(studentEntries);

**Output:**



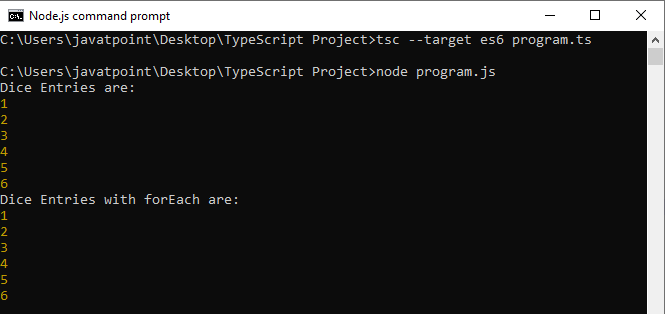
### **Iterating Set Data**

We can iterate over set values or entries by using '**for...of**' loop. The following example helps to understand it more clearly.

**Example**

1. let diceEntries = new Set();
3. diceEntries.add(1).add(2).add(3).add(4).add(5).add(6);
5. //Iterate over set entries
6. console.log("Dice Entries are:");
7. for (let diceNumber of diceEntries) {
8. console.log(diceNumber);
9. }
11. // Iterate set entries with forEach
12. console.log("Dice Entries with forEach are:");
13. diceEntries.forEach(function(value) {
14. console.log(value);
15. });

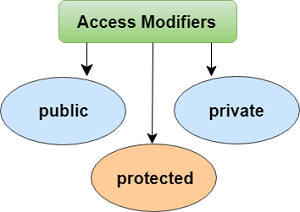
**Output:**



# **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.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

|  |  |  |  |
| --- | --- | --- | --- |
| **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

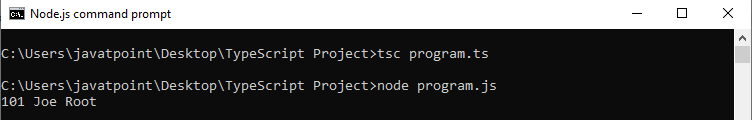
In TypeScript by default, all the members (properties and methods) of a class are public. So, there is no need to prefix members with this keyword. We can access this data member anywhere without any restriction.

**Example**

1. class Student {
2. public studCode: number;
3. studName: string;
4. }
6. let stud = new Student();
7. stud.studCode = 101;
8. stud.studName = "Joe Root";
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:**



## Private

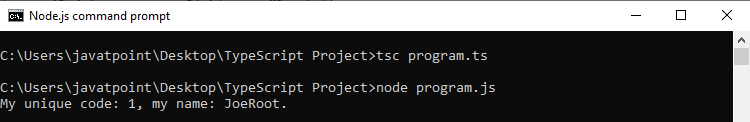
The private access modifier cannot be accessible outside of its containing class. It ensures that the class members are visible only to that class in which it is containing.

**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. }
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:**



## Protected

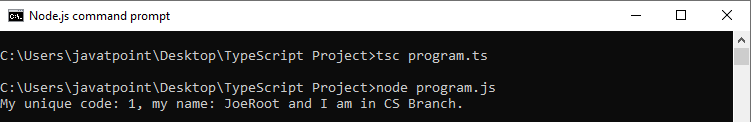
A Protected access modifier can be accessed only within the class and its subclass. We cannot access it from the outside of a class in which it is containing.

**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;
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:**



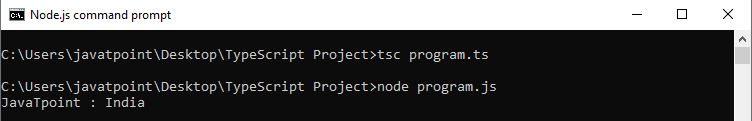
## Readonly Modifier

* We can make the properties of the class, type, or interface readonly by using the readonly modifier.
* This modifier needs to be initialized at their declaration time or in the constructor.
* We can also access readonly member from the outside of a class, but its value cannot be changed.

**Example**

1. class Company {
2. readonly country: string = "India";
3. readonly name: string;
5. constructor(contName: string) {
6. this.name = contName;
7. }
8. showDetails() {
9. console.log(this.name + " : " + this.country);
10. }
11. }
13. let comp = new Company("JavaTpoint");
14. comp.showDetails(); // JavaTpoint : India
16. comp.name = "TCS"; //Error, name can be initialized only within constructor

**Output:**



# **TypeScript Accessor**

In TypeScript, the accessor property provides a method to access and set the class members. It has two methods which are given below.

1. getter
2. setter

## getter

The getter accessor property is the conventional method which is used for retrieving the value of a variable. In object literal, the getter property denoted by "**get**" keyword. It can be public, private, and protected.

**Syntax**

1. get propName() {
2. // getter, the code executed on getting obj.propName
3. },

**Example**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

1. class MyDrawing {
2. length: number = 20;
3. breadth: string = 15;
5. get rectangle() {
6. return this.length \* this.breadth;
7. }
8. }
9. console.log(new MyDrawing().square);

## Setter

The setter accessor property is the conventional method which is used for updating the value of a variable. In object literal, the setter property is denoted by "**set**" keyword.

**Syntax**

1. set propName(value) {
2. // setter, the code executed on setting obj.propName = value
3. }

**Example**

1. set displayFullName {
2. const parts = value.split ('');
3. this.pname = firstname[0];
4. this.pname = firstname[1];
5. }
6. person displayFullName = "Abhishek Mishra"
7. console.log(student);

Note:

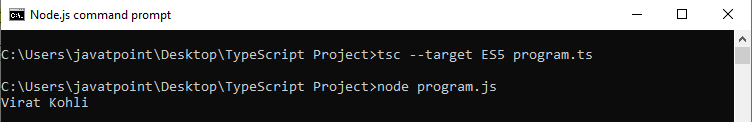
* The **getter** and **setter** give us a way of having finer control over how a member is accessed on each object.
* The TypeScript accessors require us to set the compiler to output ECMAScript 5 or higher. It does not support below ECMAScript 5.
* The accessor which has a get property without any set property is automatically assumed to be read-only. It is helpful when we are generating a **.d.ts** file from our code.

We can understand the concept of getter and setter from the below example.

### **Example**

1. let passcode = "secret passcode";
3. class Student {
4. private \_fullName: string;
6. get fullName(): string {
7. return this.\_fullName;
8. }
10. set fullName(newName: string) {
11. if (passcode && passcode == "secret passcode") {
12. this.\_fullName = newName;
13. }
14. else {
15. console.log("Unauthorized update of student detail!");
16. }
17. }
18. }
20. let stud = new Student();
21. stud.fullName = "Virat Kohli";
22. if (stud.fullName) {
23. console.log(stud.fullName);
24. }

**Output:**



Now, if we change the first line: let passcode = "secret\_passcode";

Then, Output: Unauthorized update of student detail!

**Naming convention for getter and setter**

The naming convention of the setter and getter method should be as follows:

            getXX() and setXX()

Here, XX is the name of the variable. For example:

1. private String name;

Then the setter and getter will be:

1. public void setName(String name) { }
2. public String getName() { }

# **TypeScript Function**

Functions are the fundamental building block of any applications in JavaScript. It makes the code readable, maintainable, and reusable. We can use it to build up layers of abstraction, mimicking classes, information hiding, and modules. In TypeScript, however, we have the concept of classes, namespaces, and modules, but functions still are an integral part in describing how to do things. TypeScript also allows adding new capabilities to the standard JavaScript functions to make the code easier to work.

## Advantage of function

These are the main advantages of functions.

* **Code reusability:** We can call a function several times without writing the same block of code again. The code reusability saves time and reduces the program size.
* **Less coding:** Functions makes our program compact. So, we don't need to write many lines of code each time to perform a common task.
* **Easy to debug:** It makes the programmer easy to locate and isolate faulty information.

## Function Aspects

There are three aspects of a function.

* **Function declaration:** A function declaration tells the compiler about the function name, function parameters, and return type. The syntax of the function declaration is:

1. function functionName( [arg1, arg2, ...argN] );

* **Function definition:** It contains the actual statements which are going to executes. It specifies what and how a specific task would be done. The syntax of the function definition is:

1. function functionName( [arg1, arg2, ...argN] ){
2. //code to be executed
3. }

* **Function call:** We can call a function from anywhere in the program. The parameter/argument cannot differ in function calling and a function declaration. We must pass the same number of functions as it is declared in the function declaration. The syntax of the function call is:

1. FunctionName();

## Function Creation

We can create a function in two ways. These are:

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

* Named function
* Anonymous function

### **Named function**

When we declare and call a function by its given name, then this type of function is known as a **named** function.

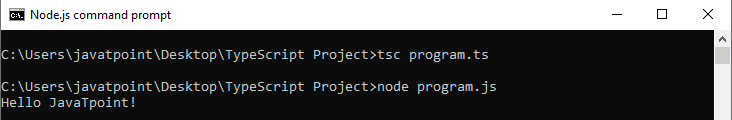
**Syntax**

1. functionName( [arguments] ) { }

**Example**

1. //Function Definition
2. function display() {
3. console.log("Hello JavaTpoint!");
4. }
5. //Function Call
6. display();

**Output:**



### **Anonymous function**

A function without a name is known as an anonymous function. These type of functions are dynamically declared at runtime. It is defined as an expression. We can store it in a variable, so it does not need function names. Like standard function, it also accepts inputs and returns outputs. We can invoke it by using the variable name, which contains function.

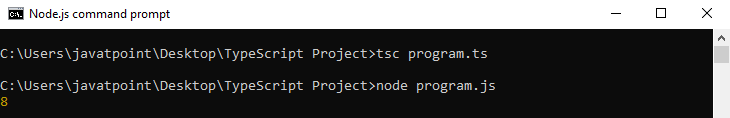
**Syntax**

1. let res = function( [arguments] ) { }

**Example**

1. // Anonymous function
2. let myAdd = function (x: number, y: number) : number {
3. return x + y;
4. };
5. // Anonymous function call
6. console.log(myAdd(5,3));

**Output:**



## Function Parameter

Parameters are the values or arguments that passed to a function. In TypeScript, the compiler accepts the same number and type of arguments as defined in the function signature. If the compiler does not match the same parameter as in the function signature, then it will give the compilation error.

**Function parameter can be categories into the following:**

* Optional Parameter
* Default Parameter
* Rest Parameter

### **Optional Parameter**

In JavaScript, we can call a function without passing any arguments. Hence, in a JavaScript function, the parameter is optional, and when we do this, each parameter value is undefined.

Unlike JavaScript, the TypeScript compiler will throw an error if we try to invoke a function without providing the exact number and types of parameters as declared in its function signature. To overcome this problem, we can use optional parameters by using the question mark sign ('?'). It means that the parameters which may or may not receive a value can be appended with a "?" sign to mark them as optional. In below example, **e\_mail\_id** is marked as an optional parameter.

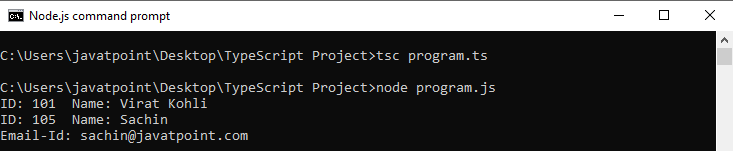
**Syntax**

1. function function\_name(parameter1[:type], parameter2[:type], parameter3 ? [:type]) { }

**Example**

1. function showDetails(id:number,name:string,e\_mail\_id?:string) {
2. console.log("ID:", id, " Name:",name);
3. if(e\_mail\_id!=undefined)
4. console.log("Email-Id:",e\_mail\_id);
5. }
6. showDetails(101,"Virat Kohli");
7. showDetails(105,"Sachin","sachin@javatpoint.com");

**Output:**



### **Default Parameter**

TypeScript provides an option to set default values to the function parameters. If the user does not pass a value to an argument, TypeScript initializes the default value for the parameter. The behavior of the default parameter is the same as an optional parameter. For the default parameter, if a value is not passed in a function call, then the default parameter must follow the required parameters in the function signature. However, if a function signature has a default parameter before a required parameter, we can still call a function which marks the default parameter is passed as an undefined value.

#### Note:**We cannot make the parameter**optional**and**default**at the same time.**

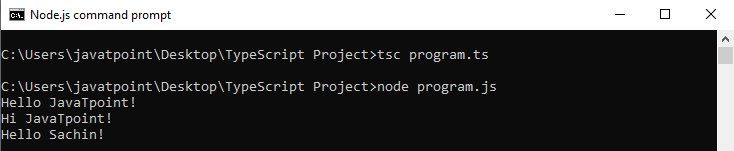
**Syntax**

1. function function\_name(parameter1[:type], parameter2[:type] = default\_value) { }

**Example**

1. function displayName(name: string, greeting: string = "Hello") : string {
2. return greeting + ' ' + name + '!';
3. }
4. console.log(displayName('JavaTpoint'));   //Returns "Hello JavaTpoint!"
5. console.log(displayName('JavaTpoint', 'Hi'));   //Returns "Hi JavaTpoint!".
6. console.log(displayName('Sachin'));    //Returns "Hello Sachin!"

**Output:**



### **Rest Parameter**

The rest parameter is used to pass **zero or more** values to a function. We can declare it by prefixing the **three "dot"** characters ('...') before the parameter. It allows the functions to have a different number of arguments without using the arguments object. The TypeScript compiler will create an array of arguments with the rest parameter so that all array methods can work with the rest parameter. The rest parameter is useful, where we have an undetermined number of parameters.

**Rules to follow in rest parameter:**

* Only one rest parameter is allowed in a function.
* It must be an array type.
* It must be the last parameter in a parameter list.

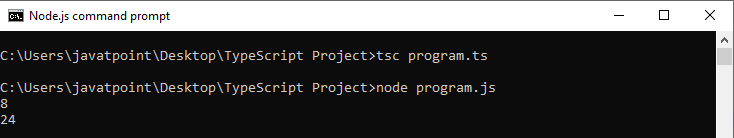
**Syntax**

1. function function\_name(parameter1[:type], parameter2[:type], ...parameter[:type]) { }

**Example**

1. function sum(a: number, ...b: number[]): number {
2. let result = a;
3. for (var i = 0; i **<** **b.length**; i++) {
4. result += b[i];
5. }
6. return result;
7. }
8. let result1 = sum(3, 5);
9. let result2 = sum(3, 5, 7, 9);
10. console.log(result1 +"\n" + result2);

**Output:**



# **TypeScript Arrow function**

ES6 version of TypeScript provides an arrow function which is the **shorthand** syntax for defining the anonymous function, i.e., for function expressions. It omits the function keyword. We can call it fat arrow (because -> is a thin arrow and => is a "**fat**" arrow). It is also called a **Lambda function**. The arrow function has lexical scoping of "**this**" keyword.

The motivation for arrow function is:

* When we don't need to keep typing function.
* It lexically captures the meaning of this keyword.
* It lexically captures the meaning of arguments.

### **Syntax**

We can split the syntax of an Arrow function into three parts:

* **Parameters:** A function may or may not have parameters.
* **The arrow notation/lambda notation** (=>)
* **Statements:** It represents the function's instruction set.

1. (parameter1, parameter2, ..., parameterN) =**>** expression;

If we use the **fat arrow (=>)** notation, there is no need to use the **function** keyword. Parameters are passed in the brackets (), and the function expression is enclosed within the curly brackets {}.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

There are two ways of writing a function in ES5 and ES6 style of coding.

1. // ES5: Without arrow function
2. var getResult = function(username, points) {
3. return username + ' scored ' + points + ' points!';
4. };
6. // ES6: With arrow function
7. var getResult = (username: string, points: number): string =**>** {
8. return `${ username } scored ${ points } points!`;
9. }

## Arrow function with parameter

The following program is an example of arrow function with parameters.

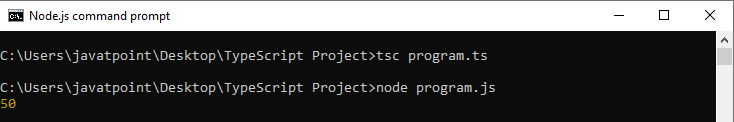
1. let sum = (a: number, b: number): number =**>** {
2. return a + b;
3. }
4. console.log(sum(20, 30)); //returns 50

In the above example, the **sum** is an arrow function, "**a: number, b: number**" is a parameter type, "**: number**" is the return type, the arrow notation => separates the function parameter and the function body.

After compiling the above TypeScript program, the corresponding JavaScript code is:

1. let sum = (a, b) =**>** {
2. return a + b;
3. };
4. console.log(sum(20, 30)); //returns 50

**Output:**

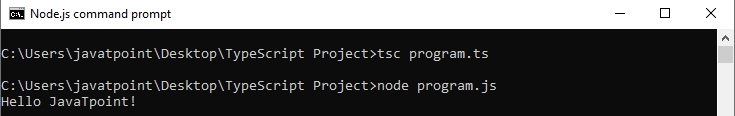


## Arrow function without a parameter

The following program is an example of arrow function without parameters.

1. let Print = () =**>** console.log("Hello JavaTpoint!");
2. Print();

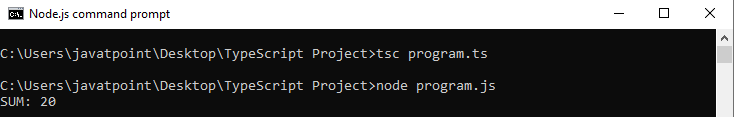
**Output:**



In the arrow function, if the function body consists of only one statement, then there is no need of the curly brackets and the return keyword. We can understand it from the below example.

1. let sum = (a: number, b: number) =**>** a + b;
2. console.log("SUM: " +sum(5, 15));

**Output:**

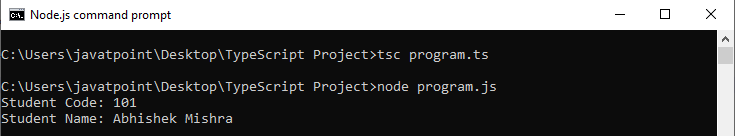


## Arrow function in a class

We can include the arrow function as a property in a class. The following example helps to understand it more clearly.

1. class Student {
2. studCode: number;
3. studName: string;
4. constructor(code: number, name: string) {
5. this.studName = name;
6. this.studCode = code;
7. }
8. showDetail = () =**>** console.log("Student Code: " + this.studCode + '\nStudent Name: ' + this.studName)
9. }
10. let stud = new Student(101, 'Abhishek Mishra');
11. stud.showDetail();

**Output:**



# **TypeScript Function Overloading**

Function overloading is a mechanism or ability to create multiple methods with the **same name** but different parameter types and **return type**. However, it can have the same number of parameters. Function overloading is also known as method overloading.

The Function/Method overloading is allowed when:

* The function name is the same
* The number of parameters is different in each overloaded function.
* The number of parameters is the same, and their type is different.
* The all overloads function must have the same return type.

Suppose we have to perform **multiplication** of the numbers, which has a different number of parameters. We write the **two** methods such as mul\_a(number, number) for **two parameters**, and mul\_b(number, number, number) for **three parameters**. Now, it may be difficult for us as well as other programmers to understand the behavior of the method because its name **differs**. In that case, we need to use function overloading, which increases the readability of the program.

## Advantage of function overloading

* It saves the memory space so that program execution becomes fast.
* It provides code reusability, which saves time and efforts.
* It increases the readability of the program.
* Code maintenance is easy.

**Example**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

1. //Function with string type parameter
2. function add(a:string, b:string): string;
3. //Function with number type parameter
4. function add(a:number, b:number): number;
5. //Function Definition
6. function add(a: any, b:any): any {
7. return a + b;
8. }
9. //Result
10. console.log("Addition: " +add("Hello ", "JavaTpoint"));
11. console.log("Addition: "+add(30, 20));

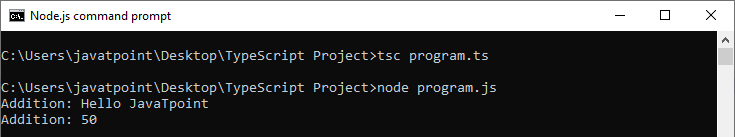
In the above example:

* The first **two** lines are the function overload **declaration**. It has two overloads:
  + A Function which accepts a **string** parameter.
  + A Function which accepts a **number** parameter.
* The third line is the **function definition**, where the data type of the parameters is set to **any**.
* The last two statements **invoke** the overloaded function.

After compiling the above TypeScript program, we will get the following JavaScript code.

1. //Function Definition
2. function add(a, b) {
3. return a + b;
4. }
5. //Result
6. console.log("Addition: " + add("Hello ", "JavaTpoint"));
7. console.log("Addition: " + add(30, 20));

**Output:**



## Function overloading in a class

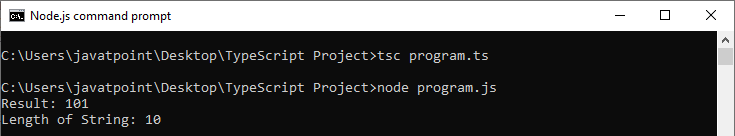
The following example helps to understand the use of method overloading in a class.

1. class A
2. {
3. public foo(s: string): number;
4. public foo(n: number): string;
5. public foo(arg: any): any
6. {
7. if (typeof(arg) === 'number')
8. return arg.toString();
9. if (typeof(arg) === 'string')
10. return arg.length;
11. }
12. }
13. let obj = new A();
14. console.log("Result: " +obj.foo(101));
15. console.log("Length of String: " +obj.foo("JavaTpoint"));

After compiling the above TypeScript program, we will get the following JavaScript code.

1. class A {
2. foo(arg) {
3. if (typeof (arg) === 'number')
4. return arg.toString();
5. if (typeof (arg) === 'string')
6. return arg.length;
7. }
8. }
9. let obj = new A();
10. console.log("Result: " + obj.foo(101));
11. console.log("Length of String: " + obj.foo("JavaTpoint"));

**Output:**



Function overloading with a different number of parameters and different types along with the same function name is not supported.

### **Example**

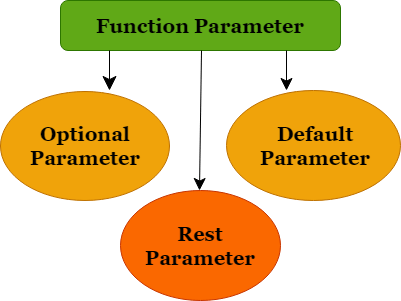
1. function display(x:number, y:number):void //Compiler Error: Duplicate function implementation
2. {
3. console.log(x + x);
4. }
6. function display(x:string): void //Compiler Error: Duplicate function implementation
7. {
8. console.log(x);
9. }

# **TypeScript Function Parameter**

Functions are the basic building block of any application which holds some business logic. The process of creating a function in TypeScript is similar to the process in JavaScript.

In functions, parameters are the values or arguments that passed to a function. The TypeScript, compiler accepts the same number and type of arguments as defined in the function signature. If the compiler does not match the same parameter as in the function signature, then it will give the compilation error.

**We can categorize the function parameter into the three types:**



* Optional Parameter
* Default Parameter
* Rest Parameter

## Optional Parameter

JavaScript allows us to call a function without passing any arguments. Hence, in a JavaScript function, the parameter is optional. If we declare the function without passing arguments, then each parameter value is undefined.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

Unlike JavaScript, the TypeScript compiler will throw an error if we try to invoke a function without providing the exact number and types of parameters as declared in its function signature. To overcome this problem, TypeScript introduces **optional parameter**. We can use optional parameters by using the **question mark sign ('?')**. It means that the parameters which may or may not receive a value can be appended with a "?" sign to mark them as optional. In the below example, **e\_mail\_id** is marked as an optional parameter.

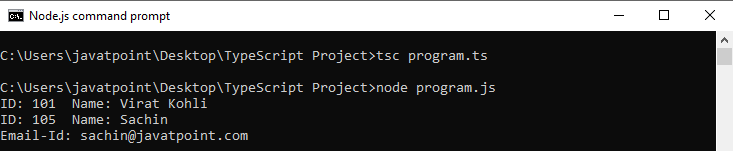
**Syntax**

1. function function\_name(parameter1[:type], parameter2[:type], parameter3 ? [:type]) { }

**Example**

1. function showDetails(id:number,name:string,e\_mail\_id?:string) {
2. console.log("ID:", id, " Name:",name);
3. if(e\_mail\_id!=undefined)
4. console.log("Email-Id:",e\_mail\_id);
5. }
6. showDetails(101,"Virat Kohli");
7. showDetails(105,"Sachin","sachin@javatpoint.com");

**Output:**



## Default Parameter

TypeScript provides an option to set default values to the function parameters. If the user does not pass a value to an argument, TypeScript initializes the default value for the parameter. The behavior of the default parameter is the same as an optional parameter. For the default parameter, if a value is not passed in a function call, then the default parameter must follow the required parameters in the function signature. However, if a function signature has a default parameter before a required parameter, we can still call a function which marks the default parameter is passed as an undefined value.

#### Note:**We cannot make the parameter**default**and**optional**at the same time.**

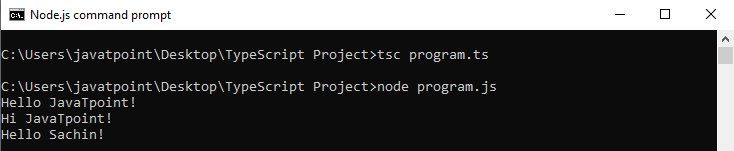
**Syntax**

1. function function\_name(parameter1[:type], parameter2[:type] = default\_value) { }

**Example**

1. function displayName(name: string, greeting: string = "Hello") : string {
2. return greeting + ' ' + name + '!';
3. }
4. console.log(displayName('JavaTpoint'));   //Returns "Hello JavaTpoint!"
5. console.log(displayName('JavaTpoint', 'Hi'));   //Returns "Hi JavaTpoint!".
6. console.log(displayName('Sachin'));    //Returns "Hello Sachin!"

**Output:**



## Rest Parameter

It is used to pass **zero or more** values to a function. We can declare it by prefixing the **three "dot"** characters ('...') before the parameter. It allows the functions to have a different number of arguments without using the arguments object. The TypeScript compiler will create an array of arguments with the rest parameter so that all array methods can work with the rest parameter. The rest parameter is useful, where we have an undetermined number of parameters.

**Rules to follow in rest parameter:**

* We can use only one rest parameter in a function.
* It must be an array type.
* It must be the last parameter in a parameter list.

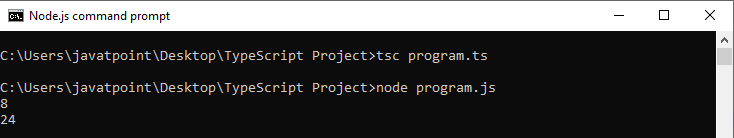
**Syntax**

1. function function\_name(parameter1[:type], parameter2[:type], ...parameter[:type]) { }

**Example**

1. function sum(a: number, ...b: number[]): number {
2. let result = a;
3. for (var i = 0; i **<** **b.length**; i++) {
4. result += b[i];
5. }
6. return result;
7. }
8. let result1 = sum(3, 5);
9. let result2 = sum(3, 5, 7, 9);
10. console.log(result1 +"\n" + result2);

**Output:**



# **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 JavaSript. 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:

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

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

### **Example**

1. class Student {
2. studCode: number;
3. studName: string;
5. constructor(code: number, name: string) {
6. this.studName = name;
7. this.studCode = code;
8. }
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)
2. **new keyword:** it is used for instantiating the object in memory.
3. 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();
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.

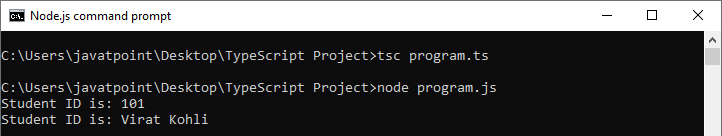
**Advantage of Method**

* Code Reusability
* Code Optimization

**Example**

1. //Defining a Student class.
2. class Student {
3. //defining fields
4. id: number;
5. name:string;
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. }
14. //Creating an object or instance
15. let obj = new Student();
16. obj.id = 101;
17. obj.name = "Virat Kohli";
18. obj.display();

**Output:**



### **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.**

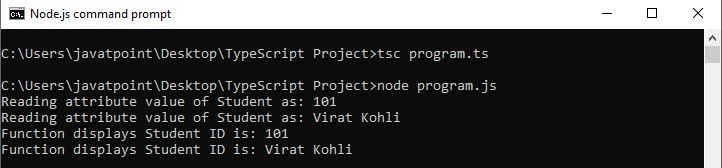
**Example**

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

**Example with constructor, method and object:**

1. //Defining a Student class.
2. class Student {
3. //defining fields
4. id: number;
5. name:string;
7. //defining constructor
8. constructor(id: number, name:string) {
9. this.id = id;
10. this.name = name;
11. }
13. //creating method or function
14. display():void {
15. console.log("Function displays Student ID is: "+this.id)
16. console.log("Function displays Student ID is: "+this.name)
17. }
18. }
20. //Creating an object or instance
21. let obj = new Student(101, "Virat Kohli")
23. //access the field
24. console.log("Reading attribute value of Student as: " +obj.id,)
25. console.log("Reading attribute value of Student as: " +obj.name)
27. //access the method or function
28. obj.display()

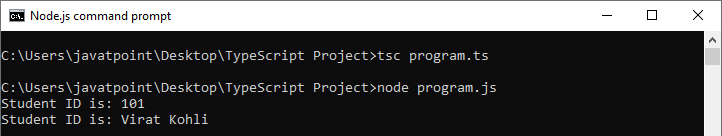
**Output:**



**Example without constructor**

1. //Defining a Student class.
2. class Student {
3. //defining fields
4. id: number;
5. name:string;
6. }
8. //Creating an object or instance
9. let obj = new Student();
11. // Initializing an object
12. obj.id = 101;
13. obj.name = "Virat Kohli";
15. //access the field
16. console.log("Student ID: " +obj.id,);
17. console.log("Student Name: " +obj.name);

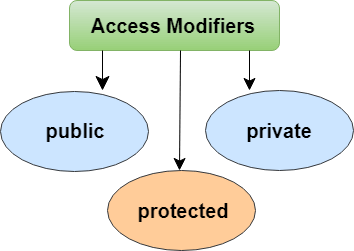
**Output:**



## Data Hiding

It is a technique which is used to hide the internal object details. A class can control the visibility of its data members from the members of the other classes. This capability is termed as encapsulation or data-hiding. OOPs uses the concept of access modifier to implement the encapsulation. The access modifier defines the visibility of class data member outside its defining class.

TypeScript supports the three types of access modifier. These are:



# **TypeScript Inheritance**

Inheritance is an aspect of OOPs languages, which provides the ability of a program to create a new class from an existing class. It is a mechanism which acquires the **properties** and **behaviors** of a class from another class. The class whose members are inherited is called the **base class**, and the class that inherits those members is called the **derived/child/subclass**. In child class, we can override or modify the behaviors of its parent class.

Before ES6, JavaScript uses **functions** and **prototype-based** inheritance, but TypeScript supports the **class-based** inheritance which comes from ES6 version. The TypeScript uses class inheritance through the **extends** keyword. TypeScript supports only **single** inheritance and **multilevel** inheritance. It doesn't support multiple and hybrid inheritance.

**Syntax**

We can declare a class inheritance as below.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

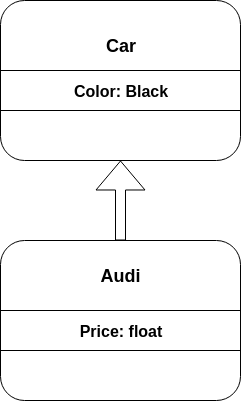
Backward Skip 10sPlay VideoForward Skip 10s

1. class sub\_class\_name extends super\_class\_name
2. {
3. // methods and fields
4. {

## Why use inheritance?

* We can use it for Method Overriding (so runtime polymorphism can be achieved).
* We can use it for Code Reusability.

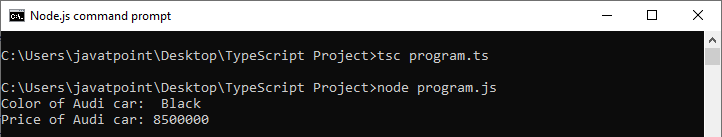
**Inheritance Example**



As displayed in the above figure, **Audi** is the subclass and **Car** is the superclass. The relationship between the two classes is **Audi IS-A Car**. It means that Audi is a type of Car.

1. class Car {
2. Color:string
3. constructor(color:string) {
4. this.Color = color
5. }
6. }
7. class Audi extends Car {
8. Price: number
9. constructor(color: string, price: number) {
10. super(color);
11. this.Price = price;
12. }
13. display():void {
14. console.log("Color of Audi car: " + this.Color);
15. console.log("Price of Audi car: " + this.Price);
16. }
17. }
18. let obj = new Audi(" Black", 8500000 );
19. obj.display();

**Output:**

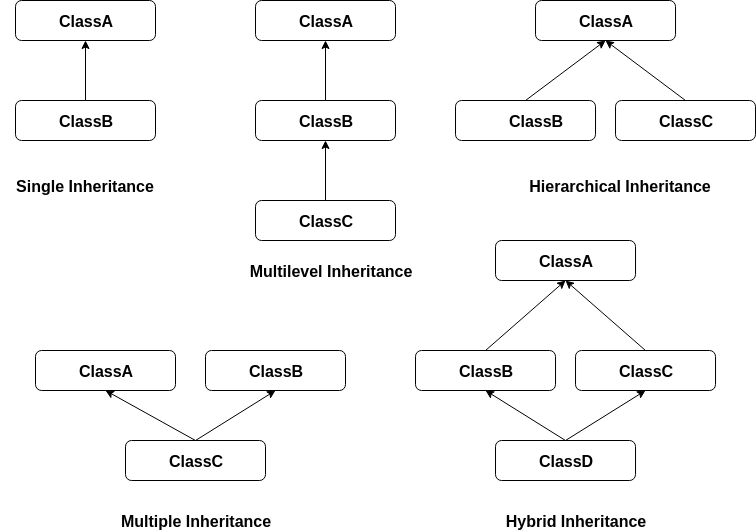


In the above example, the **Audi class** extends the **Car class** by using the **extends** keyword. It means the Audi class can include all the members of the Car class. The constructor of the Audi class initializes its own members as well as the parent class's properties by using a special keyword '**super**.' The super keyword is used to call the parent constructor and its values.

## Types of Inheritance

We can classify the inheritance into the five types. These are:

* Single Inheritance
* Multilevel Inheritance
* Multiple Inheritance
* Hierarchical Inheritance
* Hybrid Inheritance



#### Note:**TypeScript supports only single and multilevel inheritance. It doesn't support multiple, hierarchical, and hybrid inheritance.**

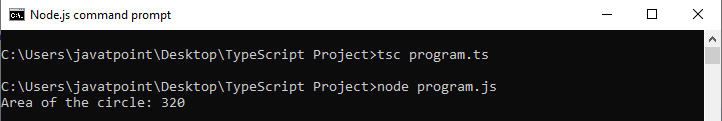
### **Single Inheritance**

Single inheritance can inherit properties and behavior from at most **one parent class**. It allows a derived/subclass to inherit the properties and behavior of a base class that enable the **code reusability** as well as we can add new features to the existing code. The single inheritance makes the code less repetitive.

**Example**

1. class Shape {
2. Area:number
3. constructor(area:number) {
4. this.Area = area
5. }
6. }
7. class Circle extends Shape {
8. display():void {
9. console.log("Area of the circle: "+this.Area)
10. }
11. }
12. var obj = new Circle(320);
13. obj.display()

**Output:**



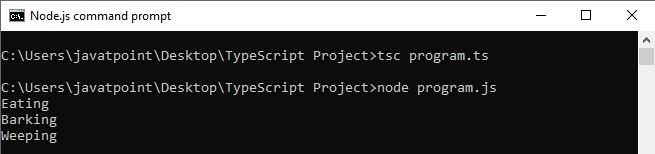
### **Multilevel Inheritance**

When a derived class is derived from another derived class, then this type of inheritance is known as **multilevel inheritance**. Thus, a multilevel inheritance has more than one parent class. It is similar to the **relation** between Grandfather, Father, and Child.

**Example**

1. class Animal {
2. eat():void {
3. console.log("Eating")
4. }
5. }
6. class Dog extends Animal {
7. bark():void {
8. console.log("Barking")
9. }
10. }
11. class BabyDog extends Dog{
12. weep():void {
13. console.log("Weeping")
14. }
15. }
16. let obj = new BabyDog();
17. obj.eat();
18. obj.bark();
19. obj.weep()

**Output:**



### **Multiple Inheritance**

When an object or class inherits the **characteristics** and **features** form more than one parent class, then this type of inheritance is known as **multiple inheritance**. Thus, a multiple inheritance acquires the properties from more than one parent class. TypeScript does not support multiple inheritance.

### **Hierarchical Inheritance**

When more than one subclass is inherited from a single base class, then this type of inheritance is known as **hierarchical inheritance**. Here, all features which are common in sub-classes are included in the base class. TypeScript does not support hierarchical inheritance.

### **Hybrid Inheritance**

When a class inherits the characteristics and features from more than one **form of inheritance**, then this type of inheritance is known as **Hybrid inheritance**. In other words, it is a **combination** of multilevel and multiple inheritance. We can implement it by combining more than one type of inheritance. TypeScript does not support hybrid inheritance.

# **TypeScript Interface**

An Interface is a structure which acts as a **contract** in our application. It defines the syntax for classes to follow, means a class which implements an interface is bound to implement all its members. We cannot instantiate the interface, but it can be referenced by the class object that implements it. 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.

When the Typescript compiler compiles it into JavaScript, then the interface will be disappeared from the JavaScript file. Thus, its purpose is to help in the development stage only.

## Interface Declaration

We can declare an interface as below.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

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**

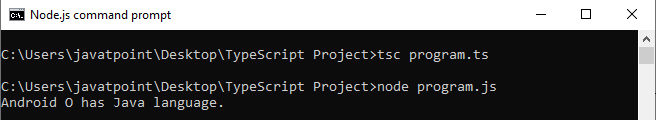
1. interface OS {
2. name: String;
3. language: String;
4. }
5. let OperatingSystem = (type: OS): void =**>** {
6. console.log('Android ' + type.name + ' has ' + type.language + ' language.');
7. };
8. let Oreo = {name: 'O', language: 'Java'}
9. OperatingSystem(Oreo);

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);

**Output:**



## Use of Interface

We can use the interface for the following things:

* Validating the specific structure of properties
* Objects passed as parameters
* Objects returned from functions.

### **Interface Inheritance**

We can inherit the interface from the other interfaces. In other words, Typescript allows an interface to be inherited from zero or more **base types**.

The base type can be a **class** or **interface**. We can use the "**extends**" keyword to implement inheritance among interfaces.

The following example helps us to understand the interface inheritance more clearly.

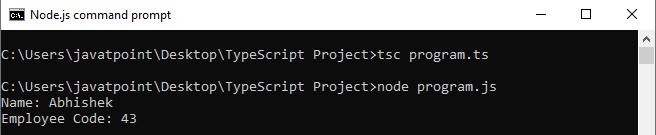
**Syntax**

1. child\_interface extends parent interface{
2. }

**Example**

1. interface Person {
2. name:string
3. age:number
4. }
5. interface Employee extends Person {
6. gender:string
7. empCode:number
8. }
9. let empObject = **<Employee>**{};
10. empObject.name = "Abhishek"
11. empObject.age = 25
12. empObject.gender = "Male"
13. empObject.empCode = 43
14. console.log("Name: "+empObject.name);
15. console.log("Employee Code: "+empObject.empCode);

**Output:**

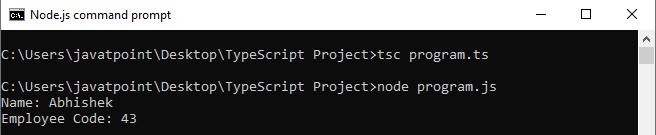


Let us take the above example, which helps us to understand the **multiple interface** inheritance.

**Example**

1. interface Person {
2. name:string
3. }
4. interface PersonDetail {
5. age:number
6. gender:string
7. }
8. interface Employee extends Person, PersonDetail {
9. empCode:number
10. }
11. let empObject = **<Employee>**{};
12. empObject.name = "Abhishek"
13. empObject.age = 25
14. empObject.gender = "Male"
15. empObject.empCode = 43
16. console.log("Name: "+empObject.name);
17. console.log("Employee Code: "+empObject.empCode);

**Output:**



## Array Type Interface

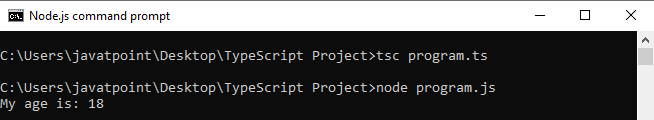
We can also use interfaces to describe the array type. The following example helps us to understand the Array Type Interface.

**Example**

1. // Array which return string
2. interface nameArray {
3. [index:number]:string
4. }
5. // use of the interface
6. let myNames: nameArray;
7. myNames = ['Virat', 'Rohit', 'Sachin'];
9. // Array which return number
10. interface ageArray {
11. [index:number]:number
12. }
13. var myAges: ageArray;
14. myAges =[10, 18, 25];
15. console.log("My age is: " +myAges[1]);

In the above example, we have declared **nameArray** that returns **string** and **ageArray** that returns **number**. The type of index in the array is always **number** so that we can retrieve array elements with the use of its **index position** in the array.

**Output:**



## Interface in a class

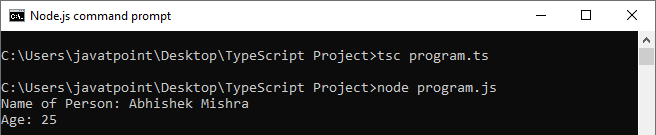
TypeScript also allows us to use the interface in a class. A class implements the interface by using the **implements** keyword. We can understand it with the below example.

**Example**

1. // defining interface for class
2. interface Person {
3. firstName: string;
4. lastName: string;
5. age: number;
6. FullName();
7. GetAge();
8. }
9. // implementing the interface
10. class Employee implements Person {
11. firstName: string;
12. lastName: string;
13. age:number;
14. FullName() {
15. return this.firstName + ' ' + this.lastName;
16. }
17. GetAge() {
18. return this.age;
19. }
20. constructor(firstN: string, lastN: string, getAge: number) {
21. this.firstName = firstN;
22. this.lastName = lastN;
23. this.age = getAge;
24. }
25. }
26. // using the class that implements interface
27. let myEmployee = new Employee('Abhishek', 'Mishra', 25);
28. let fullName = myEmployee.FullName();
29. let Age = myEmployee.GetAge();
30. console.log("Name of Person: " +fullName + '\nAge: ' + Age);

In the above example, we have declared **Person** interface with **firstName**, **lastName** as property and **FullName** and **GetAge** as **method/function**. The **Employee** class implements this interface by using the **implements** keyword. After implementing an interface, we must declare the properties and methods in the class. If we do not implement those properties and methods, it throws a **compile-time** error. We have also declared a constructor in the class. So when we instantiate the class, we need to pass the necessary parameters otherwise it throws an error at compile time.

**Output:**



## Difference between interface and inheritance

|  |  |  |
| --- | --- | --- |
| **SN** | **Interface** | **Inheritance** |
| 1. | An Interface is a structure which acts as a contract in our application. It defines the required functions, and the class is responsible for implementing it to meet that contract. | Inheritance is object-oriented programming that allows similar objects to inherit the functionality and data from each other. |
| 2. | In an interface, we can only declare properties and methods. | In inheritance, we can use a superclass to declare and defines variables and methods. |
| 3. | An interface type objects cannot declare any new methods or variables. | In this, we can declare and define its own variables and methods of a subclass that inherits a superclass. |
| 4. | Interface enforces the variables and methods which have to be present in an object. | A subclass extends the capability of a superclass to suit the "**custom**" needs. |
| 5. | Interface are classes that contain body-less structure (abstract or virtual functions). So, we have to derive the interface and then implement all of the functions in the subclass. | Inheritance is the process where one subclass acquires the properties of its superclass. |

# **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**.

#### NOTE:**A namespace can**span**in multiple files and allow to**concatenate**each file using "**-outFile**" as they were all defined in one place. It makes the code easier to maintain.**

## Namespace Declaration

We can create a namespace by 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. We can declare the namespace as below.

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.

1. namespaceName.className;
2. namespaceName.functionName;

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

1. /// **<** **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**

1. namespace studentCalc{
2. export function AnualFeeCalc(feeAmount: number, term: number){
3. return feeAmount \* term;
4. }
5. }

Main File: **app.ts**

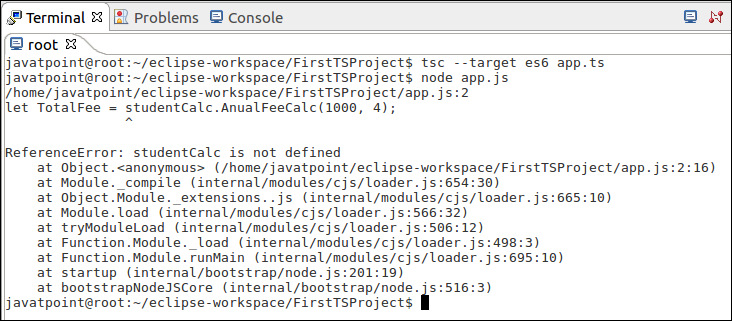
1. /// **<reference** path = "./studentCalc.ts" **/>**
3. let TotalFee = studentCalc.AnualFeeCalc(1500, 4);
5. console.log("Output: " +TotalFee);

**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

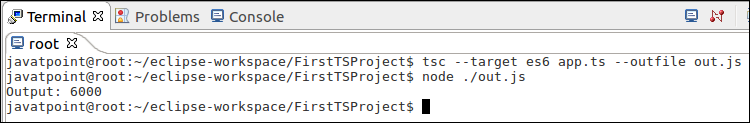
We will see the output below: **studentCalc** is not defined.



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

1. $ tsc --target es6 app.ts --outfile out.js
2. $ node ./out.js

Now, we can see the following output.



## Nested Namespaces

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.

**Example**

Nested NameSpace file: **StoreCalc**

1. namespace invoiceCalc {
2. export namespace invoiceAccount {
3. export class Invoice {
4. public calculateDiscount(price: number) {
5. return price \* .60;
6. }
7. }
8. }
9. }

Main File: **app.ts**

1. /// **<reference** path = "./StoreCalc.ts" **/>**
3. let invoice = new invoiceCalc.invoiceAccount.Invoice();
5. console.log("Output: " +invoice.calculateDiscount(400));

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

1. $ tsc --target es6 app.ts --outfile out.js
2. $ node ./out.js

It produces the following output.

Output: 240

# **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:

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

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:**

1. module Sum {
2. export function add(a, b) {
3. console.log("Sum: " +(a+b));
4. }
5. }

**Internal Module Syntax from ECMAScript 2015:**

1. namespace Sum {
2. export function add(a, b) {
3. console.log("Sum: " +(a+b));
4. }
5. }

## 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.

1. //FileName : EmployeeInterface.ts
2. export interface Employee {
3. //code declarations
4. }

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**.

1. import { class/interface name } from 'module\_name';

**Example**

Let us understands the module with the following example.

Module Creation: **addition.ts**

1. export class Addition{
2. constructor(private x?: number, private y?: number){
3. }
4. Sum(){
5. console.log("SUM: " +(this.x + this.y));
6. }
7. }

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

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

## Compiling and Executing Modules

Open the **terminal** and go to the location where you stored your **project**. Now, type the following command in the terminal window.

1. $ tsc --module commonjs app.ts
2. $ node ./app.js

**Output:**

Sum: 30

### **Importing multiple modules in single file**

We can define multiple modules in a single file. The syntax for multiple module declaration is given below.

1. import { export\_name1, export\_name2 } from 'module\_name';

**Example**

Let us understands the module with the following example.

Module Creation: **Modules.ts**

1. export class Addition{
2. constructor(private x?: number, private y?: number){
3. }
4. Sum(){
5. console.log("SUM: " +(this.x + this.y));
6. }
7. }
8. export class Substraction{
9. constructor(private a?: number, private b?: number){
10. }
11. Substract(){
12. console.log("SUBSTRACTION: " +(this.a - this.b));
13. }
14. }

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

1. import  {Addition, Substraction} from './Modules';
3. let addObject = new Addition(10, 20);
4. let subObject = new Substraction(20, 10);
6. addObject.Sum();
7. subObject.Substract();

### **Compiling and Executing Multiple Modules**

Open the **terminal** and go to the location where you stored your **project**. Now, type the following command in the terminal window.

1. $ tsc --module commonjs app.ts
2. $ node ./app.js

**Output:**

SUM: 30

SUBTRACTION: 10

## Re-exports

In TypeScript, sometimes modules extend other modules, and partially expose some of their features. A re-export does not import it locally or introduce a local variable. In this case, we can re-export some of their features either using their original name or rename it.

**Example**

Let us understands the re-export concept of a module with the following example.

Module Creation: **Modules.ts**

1. export class Addition{
2. constructor(private x?: number, private y?: number){
3. }
4. Sum(){
5. console.log("SUM: " +(this.x + this.y));
6. }
7. }

Create re-exports module: **re-exports.ts**

1. // Re-exporting types Addition as plus from Modules file
2. export { Addition as plus } from "./Modules";

In the below example, the name of **Addition** export class is changed to **plus** using {Addition as plus}. The re-exporting is useful in assigning a more meaningful name to an export which increases the readability of our program.

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

1. //importing the exporting types from re-exports file
2. import {plus as Addition} from './re-exports';  //importing plus as Addition
4. let addObject = new Addition(10, 20);
6. addObject.Sum();

### **Compiling and Executing Modules**

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

1. $ tsc --module commonjs app.ts
2. $ node ./app.js

**Output:**

Sum: 30

# **Difference between Namespaces and Modules**

## Namespace

A namespace is a way that is used for **logical grouping** of functionalities. It allows us to organize our code in a much cleaner way. A namespace can 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**.

A namespace can **span** in multiple files and allow to **concatenate** each file using **"--outFile"** as they were all defined in one place.

**Namespace Declaration**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

FileName: **StoreCalc.ts**

1. namespace invoiceCalc {
2. export namespace invoiceAccount {
3. export class Invoice {
4. public calculateDiscount(price: number) {
5. return price \* .60;
6. }
7. }
8. }
9. }

**Accessing Namespace**

1. ///**<reference** path="./StoreCalc.ts"**/>**
2. let invoice = new invoiceCalc.invoiceAccount.Invoice();
3. console.log("Output: "" +invoice.calculateDiscount(400));

To read more information, [**click here**](https://www.javatpoint.com/typescript-namespaces).

## 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.

Namespaces vs Modules

**Module Declaration**

FileName: **addition.ts**

1. export **class** Addition{
2. constructor(**private** x?: number, **private** y?: number){
3. }
4. Sum(){
5. console.log("SUM: " +(**this**.x + **this**.y));
6. }
7. }

**Accessing Modules**

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

To read more information, [**click here**](https://www.javatpoint.com/typescript-module).

## Module vs. Namespace

|  |  |  |
| --- | --- | --- |
| **SN** | **Module** | **Namespace** |
| 1. | 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. |
| 2. | 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. |
| 3. | 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. |
| 4. | We can use a module in other modules by using the import keyword. | The namespace must be included in a file by using triple-slash (///) reference syntax. e.g.  ***/// <reference path="path to namespace file"/>*** |
| 5. | It is also known as an external module. | It is also known as an internal module. |
| 6. | We can compile the module by using the "--module" command. | We can compile the namespace by using the "--outFile" command. |
| 7. | Modules import another module by using a module loader API, which was specified at the time of compilation, e.g., CommonJS, require.js, etc. | In namespace, there is no need for a module loader. Include the .js file of a namespace using the <script> tag in the HTML page. |
| 8. | A module can declare their dependencies. | Namespaces cannot declare their dependencies. |
| 9. | In modules, we can re-export some of their features either using their original name or rename it. | In namespaces, we cannot re-export their features or rename it. |
| 10. | **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"/>  let invoice = new invoiceCalc.invoiceAccount.Invoice();  console.log("Output: "" +invoice.calculateDiscount(400)); |

# **TypeScript Generics**

TypeScript Generics is a tool which provides a way to create **reusable** components. It creates a component that can work with a **variety of data types** rather than a single data type. It allows users to consume these components and use their own types. Generics ensures that the program is flexible as well as scalable in the long term.

Generics provides type safety without compromising the performance, or productivity. TypeScript uses generics with the type variable which denotes types. The type of generic functions is just like non-generic functions, with the type parameters listed first, similarly to function declarations.

In generics, we need to write a **type parameter** between the **open (<)** and **close (>)** brackets, which makes it strongly typed collections. Generics use a special kind of type variable **<T>** that denotes **types**. The generics collections contain only similar types of objects.

In TypeScript, we can create generic classes, generic functions, generic methods, and generic interfaces. TypeScript Generics is almost similar to C# and Java generics.

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

**Example**

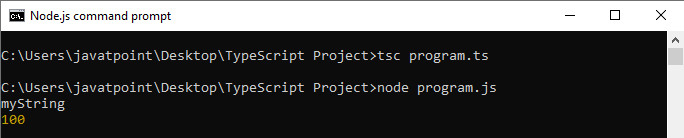
The below example helps us to understand the generics clearly.

1. function identity**<T>**(arg: T): T {
2. return arg;
3. }
4. let output1 = identity**<string>**("myString");
5. let output2 = identity**<number>**( 100 );
6. console.log(output1);
7. console.log(output2);

When we compile the above file, it returns the corresponding JavaScript file as below.

1. function identity(arg) {
2. return arg;
3. }
4. var output1 = identity("myString");
5. var output2 = identity(100);
6. console.log(output1);
7. console.log(output2);

**Output:**



### **Advantage of Generics**

There are mainly three advantages of generics. They are as follows:

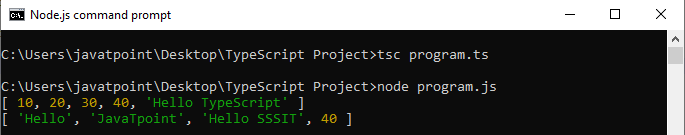
1. **Type-safety:** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
2. **Typecasting is not required:** There is no need to typecast the object.
3. **Compile-Time Checking:** It is checked at compile time so the problem will not occur at runtime.

## Why need Generics?

We can understand the need for generics by using the following example.

1. function getItems(items: any[] ) : any[] {
2. return new Array().concat(items);
3. }
4. let myNumArr = getItems([10, 20, 30]);
5. let myStrArr = getItems(["Hello", "JavaTpoint"]);
6. myNumArr.push(40); // Correct
7. myNumArr.push("Hello TypeScript"); // Correct
8. myStrArr.push("Hello SSSIT"); // Correct
9. myStrArr.push(40); // Correct
10. console.log(myNumArr); // [10, 20, 30, 40, "Hello TypeScript"]
11. console.log(myStrArr); // ["Hello", "JavaTpoint", "Hello SSSIT", 40]

**Output:**



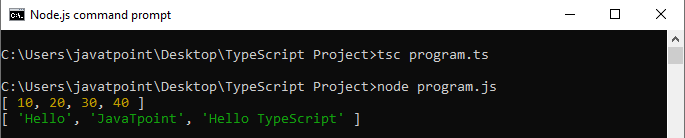
In the above example, the **getItems()** function accepts an array which is of type **any**. The getItems() function creates a new array of type **any**, concatenates items to it and returns this new array. Since we have used any data type, we can pass any type of items to the function. But, this may not be the correct way to add items. We have to add **numbers** to **number array** and the **strings** to the **string array**, but we do not want to add numbers to the string array or vice-versa.

To solve this, TypeScript introduced generics. In generics, the type variable only accepts the particular type that the user provides at declaration time. It is also preserving the type checking information.

So, we can write the above function in generic function as below.

1. function getItems**<T>**(items : T[] ) : T[] {
2. return new Array**<T>**().concat(items);
3. }
4. let arrNumber = getItems**<number>**([10, 20, 30]);
5. let arrString = getItems**<string>**(["Hello", "JavaTpoint"]);
6. arrNumber.push(40); // Correct
7. arrNumber.push("Hi! Javatpoint"); // Compilation Error
8. arrString.push("Hello TypeScript"); // Correct
9. arrString.push(50); // Compilation Error
10. console.log(arrNumber);
11. console.log(arrString);

**Output:**



In the above example, the type variable T specifies the function in the angle brackets **getItems<T>**. This variable also specifies the type of the arguments and the return value. It ensures that data type specified at the time of a function call will also be the data type of the arguments and the return value.

The generic function **getItems()** accepts the numbers array and the strings array. When we call the function **getItems<number>([10, 20, 30])**, then it will replace **T** with the number. So, the type of the arguments and the return value will be number array. Similarly, for function **getItems<string>(["Hello", "JavaTpoint"])**, the arguments type and the return value will be string array. Now, if we try to add a string in **arrNumber** or a number in **arrString** array, the compiler will show an error. Thus, it preserves the type checking advantage.

In TypeScript, we can also call a generic function without specifying the type variable. The TypeScript compiler will set the value of T on the function based on the data type of argument values.

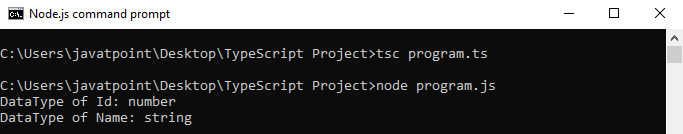
## Multi-type variables

In TypeScript Generics, we can define multi-type variables with a different name. We can understand it with the following example.

**Example**

1. function displayDataType**<T**, U**>**(id:T, name:U): void {
2. console.log("DataType of Id: "+typeof(id) + "\nDataType of Name: "+ typeof(name));
3. }
4. displayDataType**<number**, string**>**(101, "Abhishek");

**Output:**



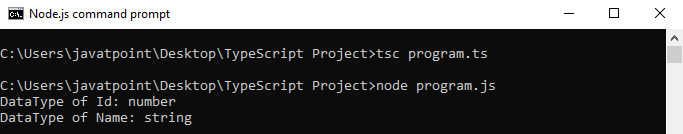
## Generic with non-generic Type

We can also use generic types with other non-generic types.

**Example**

1. function displayDataType**<T>**(id:T, name:string): void {
2. console.log("DataType of Id: "+typeof(id) + "\nDataType of Name: "+ typeof(name));
3. }
4. displayDataType**<number>**(1, "Abhishek");

**Output:**



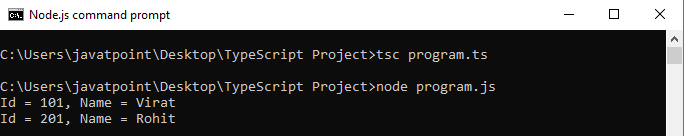
## Generics Classes

TypeScript also supports generic classes. The generic type parameter is specified in angle brackets (<>) following the name of the class. A generic class can have generic fields or methods.

**Example**

1. class StudentInfo**<T**,U**>**
2. {
3. private Id: T;
4. private Name: U;
5. setValue(id: T, name: U): void {
6. this.Id = id;
7. this.Name = name;
8. }
9. display():void {
10. console.log(`Id = ${this.Id}, Name = ${this.Name}`);
11. }
12. }
13. let st = new StudentInfo**<number**, string**>**();
14. st.setValue(101, "Virat");
15. st.display();
16. let std = new StudentInfo**<string**, string**>**();
17. std.setValue("201", "Rohit");
18. std.display();

**Output:**



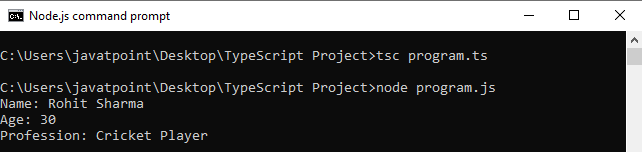
## Generics Interface

The generic type can also be used with the interface. We can understand the generic interface with the following example.

**Example**

1. interface People {
2. name: string
3. age: number
4. }
5. interface Celebrity extends People {
6. profession: string
7. }
8. function printName**<T** extends Celebrity**>**(theInput: T): void {
9. console.log(`Name: ${theInput.name} \nAge: ${theInput.age} \nProfession: ${theInput.profession}`);
10. }
11. let player: Celebrity = {
12. name: 'Rohit Sharma', age: 30, profession: 'Cricket Player'
13. }
14. printName(player);

**Output:**

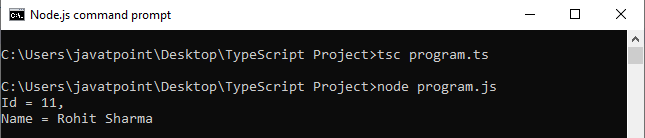


## Generics Interface as Function Type

We can also use generics interface as function types. The following example can understand it.

**Example**

1. interface StudentInfo**<T**, U**>**
2. {
3. (id: T, value: U): void;
4. };
5. function studentData(id: number, value:string):void {
6. console.log('Id = '+ id + ', \nName = ' + value)
7. }
8. let std: StudentInfo**<number**, string**>** = studentData;
9. std(11, "Rohit Sharma");



## Generic Constraints

As we know, the TypeScript Generics Types allows working with any and all data type. However, we can restrict it to certain types by using constraints. In the following example, we will create an interface that has a single .length property. We will use this interface, and the "**extends**" keyword to denote our constraint.

**Example**

1. interface Lengthwise {
2. length: number;
3. }
4. function loggingIdentity**<T** extends Lengthwise**>**(arg: T): T {
5. console.log("Length: " +arg.length);  // It has a .length property, so no more error found
6. return arg;
7. }
8. loggingIdentity({length: 10, value: 9});
9. loggingIdentity(3);  // Compilation Error, number doesn't have a .length property

**Output:**

Length: 10

Length: undefined

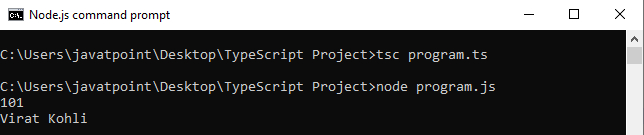
## Generic Constraints with class

A more advanced example of Generic constraints relationships between the constructor function and the instance side of class types is given below.

**Example**

1. class Student {
2. Id: number;
3. Name: string;
5. constructor(id:number,  name:string) {
6. this.Id = id;
7. this.Name = name;
8. }
9. }
10. function display**<T** extends Student**>**(per: T): void {
11. console.log(`${ st.Id} ${st.Name}` );
12. }
13. var st = new Student(101, "\nVirat Kohli");
14. display(st);

**Output:**



# **TypeScript Decorators**

A Decorator is a special kind of declaration that can be applied to classes, methods, accessor, property, or parameter. Decorators are simply functions that are prefixed **@expression** symbol, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

#### **Note: Decorators are an experimental feature proposed for ES7. It is already in use by some of the JavaScript frameworks including Angular 2. The Decorators may change in future releases.**

To enable experimental support for decorators, we must enable the **experimentalDecorators** compiler option either on the **command line** or in our **tsconfig.json:**

## Command Line

1. $tsc --target ES5 --experimentalDecorators

**tsconfig.json**

1. {
2. "compilerOptions": {
3. "target": "ES5",
4. "experimentalDecorators": true
5. }
6. }

## Purpose

TypeScript Decorators serves the purpose of adding both annotations and metadata to the existing code in a declarative way.

Backward Skip 10sPlay VideoForward Skip 10s

## Decorator Factories

To customize decorator how it is applied to a declaration, we can write a decorator factory. A decorator factory is a function which returns the expression that will be called by the decorator at runtime.

A decorator factory can be written in the following manner:

1. function color(value: string) { // this is the decorator factory
2. return function (target) { // this is the decorator
3. // do something with 'target' and 'value'...
4. }
5. }

## Decorator Composition

We can apply multiple decorators to a declaration. The following examples help to understand it.

## On a single line

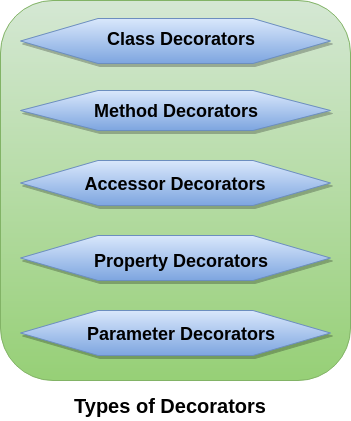
1. @f @g x

## On multiple lines

1. @f
2. @g
3. x

## Types of Decorators

TypeScript uses the following types of Decorators:



1. [Class Decorators](https://www.javatpoint.com/typescript-decorators#class-decorators)
2. [Method Decorators](https://www.javatpoint.com/typescript-decorators#method-decorators)
3. [Accessor Decorators](https://www.javatpoint.com/typescript-decorators#accessor-decorators)
4. [Property Decorators](https://www.javatpoint.com/typescript-decorators#property-decorators)
5. [Parameter Decorators](https://www.javatpoint.com/typescript-decorators#parameter-decorators)

### **1. Class Decorators**

A class decorator is defined just before the class declaration, and it tells about the class behaviors. A class decorator is applied to the constructor of the class. A class decorator can be used to observe, modify, or replace a class definition. If the class decorator returns a value, it will replace the class declaration with the given constructor function.

### **Example:**

1. @sealed
2. class Person {
3. msg: string;
4. constructor(message: string) {
5. this.msg = message;
6. }
7. show() {
8. return "Hello, " + this.msg;
9. }
10. }

In the above example, when **@sealed** decorator is executed, it will seal both the constructor and its prototype so that we cannot inherit the **Person** class.

### **2. Method Decorators**

A Method Decorator is defined just before a method declaration. It is applied to a property descriptor for the method. It can be used to observe, modify, or replace a method definition. We cannot use method decorator in a declaration file.

The expression for the method decorator function accepts three arguments. They are:

1. Either the constructor function of the class for a static member or the prototype of the class for an instance member.
2. The member name.
3. The Property Descriptor for the member.

### **Example:**

In the below example, the **@log** decorator will log the new item entry.

1. class Item {
2. itemArr: Array;
3. constructor() {
4. this.itemArr = [];
5. }
6. @log
7. Add(item: string): void {
8. this.itemArr.push(item);
9. }
10. GetAll(): Array {
11. return this.itemArr;
12. }
13. }

### **3. Accessor Decorators**

An Accessor Decorator is defined just before an accessor declaration. It is applied to the property descriptor for the accessor. It can be used to observe, modify, or replace an accessor's definitions.

#### **Note: An accessor is a getter and setter property of the class declaration.**

The expression for the accessor decorator function accepts three arguments. They are:

1. Either the constructor function of the class for a static member or the prototype of the class for an instance member.
2. The member name.
3. The Property Descriptor for the member.

### **Example:**

In the below example, an accessor decorator **(@configurable)** is applied to a member of the **Employee** class.

1. class Employee {
2. private \_salary: number;
3. private \_name: string;
5. @configurable(false)
6. get salary() { return 'Rs. ${this.\_salary}'; }
7. set salary(salary: any) { this.\_salary = +salary; }
9. @configurable(true)
10. get name() { return 'Sir/Madam, ${this.\_name}'; }
11. set name(name: string) { this.\_name = name; }
12. }

### **4. Property Decorators**

A property decorator is defined just before a property declaration. It is similar to the method decorators. The only difference between property decorators and method decorators is that they do not accept property descriptor as an argument and do not return anything.

The expression for the property decorator function accepts two arguments. They are:

1. Either the constructor function of the class for a static member or the prototype of the class for an instance member.
2. The member name.

### **Example:**

In the below example, the **@ReadOnly** decorator will make the name property as read-only, so we can't change its value.

1. class Company {
2. @ReadOnly
3. name: string = "JavaTpoint.com";
4. }
5. let comp = new Company();
6. comp.name = 'SSSIT.com'; // Here, we can't change company name.
7. console.log(comp.name); // 'JavaTpoint.com'

### **5. Parameter Decorators**

A parameter decorator is defined just before a parameter declaration. It is applied to the function for a class constructor or method declaration. It cannot be used in a declaration file or in any other ambient context (such as in a declared class).

The expression for the parameter decorator function accepts three arguments. They are:

1. Either the constructor function of the class for a static member or the prototype of the class for an instance member.
2. The member name.
3. The index of the parameter in the function?s arguments list.

### **Example:**

In the below example, a parameter decorator **(@required)** is applied to the parameter of a member of the **Person** class.

1. class Person {
2. msg: string;
3. constructor(message: string) {
4. this.msg = message;
5. }
6. @validate
7. show(@required name: string) {
8. return "Hello " + name + ", " + this.msg;
9. }
10. }

# **TypeScript Date Object**

The Date object represents a **date** and **time** functionality in TypeScript. It allows us to get or set the year, month and day, hour, minute, second, and millisecond.

If we create a date without any argument passed to its constructor, by default, it contains the date and time of the user's computer.

The Date object also provides the functions which deal with **Coordinated Universal Time (UTC) time**, also known as **Greenwich Mean Time (GMT)**. The World Time Standard is based on UTC time.

## Creating Date Object

There are four ways to create a new date object:

**1. new Date():** It creates a new date object with the current **date** and **time**.

**Example**

1. let date: Date = new Date();
2. console.log("Date = " + date); //Date = Tue Feb 05 2019 12:05:22 GMT+0530 (IST)

**2. new Date(milliseconds):** It creates a new date object as **zero** time plus **milliseconds**.

**Example**

1. let date: Date = new Date(500000000000);
2. console.log("Date = " + date); //Date = Tue Nov 05 1985 06:23:20 GMT+0530 (IST)

**3. new Date(datestring):**It creates a new date object from a date string.

**Example**

1. let date: Date = new Date("2019-01-16");
2. console.log("Date = " + date); //Date = Wed Jan 16 2019 05:30:00 GMT+0530 (IST)

**4. new Date ( year, month, date[, hour, minute, second, millisecond ]):**It creates a new date object with a specified date and time.

**Example**

1. let date: Date = new Date(2018, 0O5, 0O5, 17, 23, 42, 11);
2. console.log("Date = " + date); //Date = Tue Jun 05 2018 17:23:42 GMT+0530 (IST)

## Date Object Properties

|  |  |
| --- | --- |
| **Property** | **Description** |
| constructor | It specifies the function that creates an object's prototype. |
| prototype | It allows to add properties and methods to an object. |

## Date Object Methods

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1. | Date() | It is used to returns the current date and time. |
| 2. | getDate() | It is used to returns day of the month for the specified date according to local time. |
| 3. | getDate() | It is used to returns day of the week for the specified date according to local time. |
| 4. | getFullYear() | It is used to returns year of the specified date according to local time. |
| 5. | getHours() | It is used to returns hours in the specified date according to local time. |
| 6. | getMilliseconds() | It is used to returns milliseconds in the specified date according to local time. |
| 7. | getMinutes() | It is used to returns minutes in the specified date according to local time. |
| 8. | getMonth() | It is used to returns month in the specified date according to local time. |
| 9. | getSeconds() | It is used to returns seconds in the specified date according to local time. |
| 10. | getTime() | It is used to returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC. |
| 11. | getTimezoneOffset() | It is used to returns the time-zone offset in minutes for the current locale. |
| 12. | getUTCDate() | It is used to returns the day(date) of the month in the specified date according to universal time. |
| 13. | getUTCDay() | It is used to returns day of the week in the specified date according to universal time. |
| 14. | getUTCFullYear() | It is used to returns the year in the specified date according to universal time. |
| 15. | getUTCHours() | It is used to returns hours in the specified date according to universal time. |
| 16. | getUTCMilliseconds() | It is used to returns milliseconds in the specified date according to universal time. |
| 17. | getUTCMinutes() | It is used to returns the minutes in the specified date according to universal time. |
| 18. | getUTCMonth() | It is used to returns the month in the specified date according to universal time. |
| 19. | getUTCSeconds() | It is used to returns the seconds in the specified date according to universal time. |
| 20. | setDate() | It is used to sets the day of the month for a specified date according to local time. |
| 21. | setFullYear() | It is used to sets the full year for a specified date according to local time. |
| 22. | setHours() | It is used to sets the hours for a specified date according to local time. |
| 23. | setMilliseconds() | It is used to sets the milliseconds for a specified date according to local time. |
| 24. | setMinutes() | It is used to sets the minutes for a specified date according to local time. |
| 25. | setMonth() | It is used to sets the month for a specified date according to local time. |
| 26. | setSeconds() | It is used to sets the seconds for a specified date according to local time. |
| 27. | setTime() | It is used to sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC. |
| 28. | setUTCDate() | It is used to sets the day(date) of the month for a specified date according to universal time. |
| 29. | setUTCFullYear() | It is used to sets the full year in the specified date according to universal time. |
| 30. | setUTCHours() | It is used to sets the hours for a specified date according to universal time. |
| 31. | setUTCMilliseconds() | It is used to sets the milliseconds for a specified date according to universal time. |
| 32. | setUTCMinutes() | It is used to sets the minutes for a specified date according to universal time. |
| 33. | setUTCMonth() | It is used to sets the month for a specified date according to universal time. |
| 34. | setUTCSeconds() | It is used to sets the seconds for a specified date according to universal time. |
| 35. | toDateString() | It is used to returns the "date" portion of the date as a human-readable string. |
| 36. | toLocaleDateString() | It is used to returns the "date" portion of the Date as a string, using the current locale's conventions. |
| 37. | toLocaleFormat() | It converts a date to a string, using a format string. |
| 38. | toLocaleString() | It converts a date to a string, using the current locale's conventions. |
| 39. | toLocaleTimeString() | It is used to returns the "time" portion of the Date as a string, using the current locale's conventions. |
| 40. | toSource() | It is used to returns a string representing the source for an equivalent Date object; you can use this value to create a new object. |
| 41. | toString() | It is used to returns a string representing the specified Date object. |
| 42. | toTimeString() | It is used to returns the "time" portion of the Date as a human-readable string. |
| 43. | toUTCString() | It converts a date to a string, using the universal time convention. |
| 44. | valueOf() | It is used to returns the primitive value of a Date object. |

**Example**

1. let date: Date = new Date(2017, 4, 4, 17, 23, 42, 11);
2. date.setDate(13);
3. date.setMonth(13);
4. date.setFullYear(2013);
5. date.setHours(13);
6. date.setMinutes(13);
7. date.setSeconds(13);
8. console.log("Year = " + date.getFullYear());
9. console.log("Date = " + date.getDate());
10. console.log("Month = " + date.getMonth());
11. console.log("Day = " + date.getDay());
12. console.log("Hours = " + date.getHours());
13. console.log("Minutes = " + date.getMinutes());
14. console.log("Seconds = " + date.getSeconds());

**output**

Year = 2013

Date = 13

Month = 1

Day = 3

Hours = 13

Minutes = 13

Seconds = 13

# **TypeScript Duck-Typing**

According to TypeScript, Duck-Typing is a method/rule used to check the **type compatibility** for more complex variable types.

TypeScript uses the **duck-typing** method to compare one object with other objects by checking that both objects have the **same type matching** names or not. It means we cannot change the signature of a variable. For example, if we assign an object that has **two** properties like name, address and next time we assign an object which contains more properties or fewer properties or both properties are not (name, address), then the TypeScript compiler will generate the **compile-time error**. The concept is known as **Duck typing**.

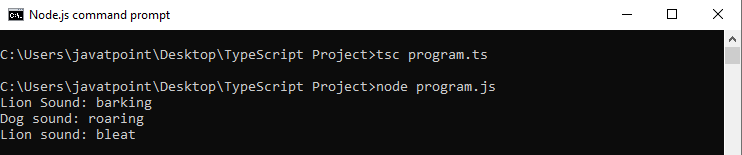
The duck-typing feature provides type safety in TypeScript code.

Through the duck-typing rule TypeScript compiler checks that an object is same as other object or not.

According to the duck-typing method, both objects must have the same properties/variables types.

1. class Dog {
2. sound = "barking";
3. }
4. class Lion {
5. sound = "roaring";
6. }
7. class Goat {
8. sound = "bleat";
9. swim(){
10. console.log("Cannot Swim!");
11. }
12. }
13. let lion: Lion = new Dog(); // substitutes
14. let dog: Dog = new Lion(); // substitutes
15. let lionTwo: Lion = new Goat();
16. //let goat: Goat = new Lion(); // IDE & compiler error
17. console.log("Lion Sound: "+lion.sound);
18. console.log("Dog sound: "+dog.sound);
19. console.log("Lion sound: "+lionTwo.sound);

**output**



In the above example, we can see that it does not allow substitution of a **Lion** for **a Goat** because the Goat class has an additional method (so Lion fails duck typing). **Dog** and Lion are substitutable in duck typing because there's nothing a lion can do that a dog cannot, and vice versa.

# **TypeScript Ambients Declarations**

TypeScript provides the way to safely and easily use existing JavaScript libraries like jQuery, AngularJS, Node.js, etc. The Ambient declarations allow us to use existing popular JavaScript libraries safely.

Ambient declarations tell the TypeScript compiler about the actual source code (like variable/functions) existing elsewhere. If our TypeScript code needs to use a third-party library that was written in plain JavaScript libraries like jQuery/AngularJS/Node.js, we can always write ambient declarations. The ambient declaration describes the types that would have been there and will be written in TypeScript.

## Ambients Declarations

Ambient declarations files need to save with the extension **(d.ts)**. A file with extension .d.ts must have the **declare** keyword prefixed to each root level definition. It makes clear to the author that there will be no code emitted by TypeScript. The author needs to ensure that the declared item will exist at runtime.

Ambient declarations tell the compiler about the actual source code exist elsewhere. If these source codes do not exist at runtime and we try to use them, then it will break without warning.

Ambient declarations files are like **docs** file. If the source changes, the docs need to be kept updated also. If the ambient declaration file is not updated, it returns compilation errors.

1. Test.d.ts

We cannot **trans-compiled** the above file into JavaScript. We can use the above file for **type-safety** and **IntelliSense**.

We can declare the ambient variables and methods by using the **declare** keyword. The syntax for the ambient declaration is like below.

**Syntax**

1. declare module module\_name{
2. }

**Syntax to access ambient files**

1. /// **<reference** path = "AmbientFileName.d.ts" **/>**

**Example**

We can understand the ambient declaration with the following example. Here, we are using a third-party JavaScript library with the below code.

**Addition.js**

1. var TestSum;
2. (function (TestSum) {
3. var Calc = (function () {
4. function Calc() {
5. }
6. Calc.prototype.doSum = function (a, b) {
7. return a + b;
8. }
9. })
10. })

Above is a **JS file** and we have not much time to re-write this library to TypeScript. But still, if we need to use the **doSum()** function with type safety, then we can do this by using **ambient declaration**. Let us create an ambient declaration file.

**CalcSum.d.ts**

1. declare module TestSum {
2. export class Calc {
3. doSum(a:number, b:number) : number;
4. }
5. }

Now, include this ambient declaration file **(CalcSum.d.ts)** into our TypeScript file.

**Main.ts**

1. /// **<reference** path = "CalcSum.d.ts" **/>**
2. var obj = new TestSum.Calc();
3. console.log("Sum: " +obj.doSum(15,25));

Compile and executed the **Main.ts** file by using the following command on the console.

1. $ tsc main.ts
2. $ node Main.js

We will get the following output.

Sum: 40

# **Typescript Compilation Context**

The compilation context is a term for a grouping of the TypeScript files that will parse and analyze to determine what is valid and what is not valid. The compilation context contains the information about which compiler options are in use. We can define this logical grouping of TypeScript files by using a tsconfig.json file.

We can compile the TypeScript files by using the tsc **<file name>.ts** command. When we use **'$tsc'** command to compile TypeScript code, compiler searches for configurations which are loaded in the **tsconfig.json** file. TypeScript also provides an option to compile multiple **.ts** files in the large project.

## tsconfig.json

The tsconfig.json file is a file which is in **JSON** format. In the tsconfig.json file, we can specify various options which tell the compiler how to compile the current project.

The first step in any new TypeScript project is to add a tsconfig.json file. To create tsconfig.json file open the folder where you want to store your source file and add a new file named **tsconfig.json.**

PlayNext

Unmute

Current TimeÂ 0:00

/

DurationÂ 18:10

Loaded: 0.37%

Â

Fullscreen

Backward Skip 10sPlay VideoForward Skip 10s

We can compile a typescript project in one of the following ways:

* **By invoking tsc with no input files:** In this case, the compiler searches the tsconfig.json file starting in the **current directory** following the **parent directory** chain.
* **By invoking tsc with no input files and a --project (or just -p) command:** In this case, the compiler specifies the path of a directory which contains a tsconfig.json file. It also specifies a path to a valid **.json** file which contains the configurations.

## Create tsconfig.json file

We can create a tsconfig.json file in our project's **root** folder by using the following command.

1. $ tsc --init

If we execute the above command, a default **tsconfig.json** file will be created.

1. {
2. "compilerOptions": {
3. "declaration": true,
4. "emitDecoratorMetadata": false,
5. "experimentalDecorators": false,
6. "module": "none",
7. "moduleResolution": "node",
8. "noFallthroughCasesInSwitch": false,
9. "noImplicitAny": false,
10. "noImplicitReturns": false,
11. "removeComments": false,
12. "sourceMap": false,
13. "strictNullChecks": false,
14. "target": "es3"
15. },
16. "compileOnSave": true
17. }

## Include and Exclude properties

An **include** and **exclude** properties allows us to take an array pattern to add or remove a list of TypeScript files from the compilation process.

1. {
2. "compilerOptions": {
3. "module": "system",
4. "noImplicitAny": true,
5. "removeComments": true,
6. "preserveConstEnums": true,
7. "outFile": "../../built/local/tsc.js",
8. "sourceMap": true
9. },
10. "include": [
11. "src/\*\*/\*"
12. ],
13. "exclude": [
14. "node\_modules",
15. "\*\*/\*.spec.ts"
16. ]
17. }

## compilerOptions Property

We can customize the compiler options properties by using **compilerOptions.** It allows specifying additional options to the typescript compiler. Some of the important compiler options are summarized in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Option** | **Type** | **Default** | **Description** |
| --allowJs | boolean | false | Allow JavaScript files to be compiled. |
| --alwaysStrict | boolean | false | Parse in strict mode and emit "use strict" for each source file. |
| --baseUrl | string |  | It is base directory to resolve non-directory module names. |
| --build--b | boolean | false | It is used to build a project and all of its dependencies specified by project references. |
| --declaration--d | boolean | false | It generates a corresponding .d.ts file. |
| --diagnostics | boolean | false | It shows diagnostic information. |
| --experimentalDecorators | boolean | false | It enables the experimental support for ES decorators. |
| --isolatedModules | boolean | false | It is used to transpile each file as a separate module. |
| --module--m | string | target === "ES3" or "ES5" ? "CommonJS" : "ES6" | The output module type, e.g. "CommonJS", ?AMD?, "System", "ES6", "ES2015" or "ESNext." Default value is CommonJS if the target attribute is ES3 or ES5; else default is ES6. |
| --moduleResolution | string | module === "AMD" or "System" or "ES6" ? "Classic" : "Node" | It determines how modules get resolved. Either "Node" for Node.js/io.js style resolution, or "Classic." |
| --noEmitOnError | boolean | false | Do not emit outputs if any errors were reported. |
| --outDir | string |  | Redirect output structure to the directory. |
| --sourceMap | boolean | false | Generates a corresponding .map file. It helps in debugging. |
| --target--t | string | "ES3" | Specify ECMAScript target version: "ES3" (default), "ES5", "ES6"/"ES2015", "ES2016", "ES2017" or "ESNext". |
| --watch--w |  |  | It runs the compiler in watch mode. It means that whenever any of the source files are changed, then the compiling process is re-triggered to generate the transpiled files again. |

To see the complete list of compiler options go to the [official page](https://www.typescriptlang.org/docs/handbook/compiler-options.html).

## compileOnSave

It is the property which is used to set the **IDE** to compile the included TypeScript files and generate the output automatically. It signals to the IDE to generate all files for a given tsconfig.json upon saving.

1. {
2. "compileOnSave": true,
3. "compilerOptions": {
4. "noImplicitAny" : true
5. }
6. }

# **TypeScript Build Tools**

Build tools are programming utilities which help to **automate** the transformation and **bundling** of our source code into a single file. A build tool utility is used to build a new version of a program. Building means compiling, linking, and packaging the code into the executable form.

The Build tools are usually run on the **command line**, either in IDE or completely separate from it.

Build tools or build automation is the act of scripting or automating a variety of tasks that developers do in their day-to-day activities. These are:

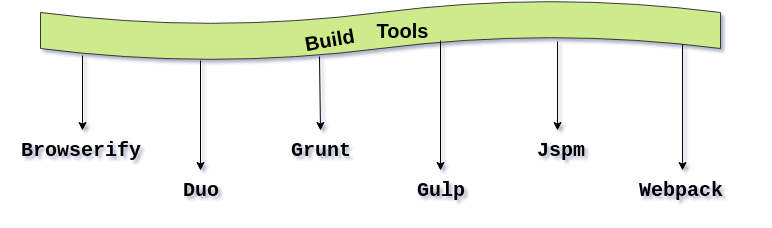
* Downloading dependencies.
* Compiling source code into binary code.
* Packaging that binary code.
* Running tests.
* Deployment to production systems.

## Use of build tools

In small projects, the software developers manually invoke the build process, which is not a good practice for larger projects. It is because, in larger projects, it is very hard to keep track of what needs to be built, in what sequence and what should be the dependencies in the building process. So we use an automation tool which allows the build process to be more consistent.

Some of the standard build tools that can be integrated with TypeScript are:

1. [Browserify](https://www.javatpoint.com/typescript-build-tools#Browserify)
2. [Duo](https://www.javatpoint.com/typescript-build-tools#Duo)
3. [Grunt](https://www.javatpoint.com/typescript-build-tools#Grunt)
4. [Gulp](https://www.javatpoint.com/typescript-build-tools#Gulp)
5. [Jspm](https://www.javatpoint.com/typescript-build-tools#Jspm)
6. [Webpack](https://www.javatpoint.com/typescript-build-tools#Webpack)



## 1. Browserify

Use the Browserify plugin **Tsify** for compiling TypeScript files.

**Install**

Install **Tsify** by using the following command:

1. $npm install Tsify

**Using Command Line Interface**

By using the following command, compile your code that saves the result in a file named **bundle.js.**

1. browserify main.ts -p [ tsify --noImplicitAny ] **>** bundle.js

**Using API**

1. var browserify = require("browserify");
2. var tsify = require("tsify");
4. browserify()
5. .add("main.ts")
6. .plugin("tsify", { noImplicitAny: true })
7. .bundle()
8. .pipe(process.stdout);

## 2. Duo

**Install**

Install the **Duo plugin** by using the following command:

1. $npm install duo-typescript

**Using Command Line Interface**

By using the following command, compile your code that saves the result in a file named **entry.ts.**

1. $duo --use duo-typescript entry.ts

**Using API**

1. var Duo = require("duo");
2. var fs = require("fs")
3. var path = require("path")
4. var typescript = require("duo-typescript");
6. var out = path.join(\_\_dirname, "output.js")
8. Duo(\_\_dirname)
9. .entry("entry.ts")
10. .use(typescript())
11. .run(function (err, results) {
12. if (err) throw err;
13. // Write compiled result to output file
14. fs.writeFileSync(out, results.code);
15. });

## 3. Grunt

Use **grunt-ts** plugin from Grunt for compiling TypeScript files.

**Install**

Install **grunt-ts** by using the following command:

1. $npm install grunt-ts

Now, you need to include the Grunt **config file** named **gruntfile.js** in your project.

1. module.exports = function(grunt) {
2. grunt.initConfig({
3. ts: {
4. default: {
5. src: ["\*\*/\*.ts", "!node\_modules/\*\*/\*.ts"]
6. }
7. }
8. });
9. grunt.loadNpmTasks("grunt-ts");
10. grunt.registerTask("default", ["ts"]);
11. };

## 4. Gulp

Use a **gulp-typescript** plugin for compiling TypeScript files.

**Install**

Install **gulp-typescript** by using the following command:

1. $npm install gulp-typescript

Now, you need to include the **Gulp config** file named **gulpfile.js** in your project.

1. var gulp = require("gulp");
2. var ts = require("gulp-typescript");
4. gulp.task("default", function () {
5. var tsResult = gulp.src("src/\*.ts")
6. .pipe(ts({
7. noImplicitAny: true,
8. out: "output.js"
9. }));
10. return tsResult.js.pipe(gulp.dest("built/local"));
11. });

## 5. Jspm

Use **jspm plugin** for compiling TypeScript files.

**Install**

Install **jspm** by using the following command:

1. $npm install -g jspm@beta

#### **Note: Currently TypeScript support in jspm is in 0.16beta**

## 6. Webpack

Use **ts-loader** plugin for compiling TypeScript files.

**Install**

Install **webpack** by using the following command:

1. $npm install ts-loader --save-dev

Now, you need to include the Webpack **config file** named **webpack.config.js** in your project.

1. module.exports = {
2. entry: "./src/index.tsx",
3. output: {
4. filename: "bundle.js"
5. },
6. resolve: {
7. // Add '.ts' and '.tsx' as a resolvable extension.
8. extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
9. },
10. module: {
11. loaders: [
12. // all files with a '.ts' or '.tsx' extension will be handled by 'ts-loader'
13. { test: /\.tsx?$/, loader: "ts-loader" }
14. ]
15. }
16. }

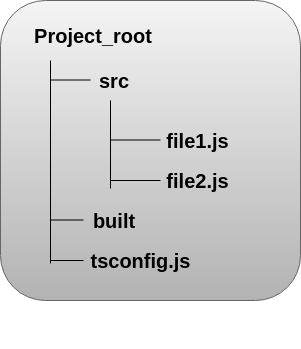
# **Migrating from JavaScript to TypeScript**

The following things are necessary to migrate from JavaScript to TypeScript:

* You know JavaScript.
* You know patterns and build tools used in the project.

Let us assume we have JavaScript files that need to be converted into TypeScript. We know that, when we compile a TypeScript file, it produces corresponding JavaScript file with the same name. Here, we need to ensure that our original JavaScript file which acts as input cannot have in the same directory so that TypeScript does not override them.

From the above point, we are going to assume that our **directory** is set up like the below structure. Here, we kept all the output files in an output directory called "**built**."



We can use the following process to migrate from JavaScript to TypeScript:

1. Add a tsconfig.json file to project.
2. Integrate with a build tool.
3. Moving all .js files to .ts files.
4. Check for errors.
5. sing third-party JavaScript libraries.

## 1. Add tsconfig.json file to project

First, we need to add a **tsconfig.json** file in our project. TypeScript uses a tsconfig.json file for managing our project's compilation options, such as which files we want to **include** and **exclude.**

1. {
2. "compilerOptions": {
3. "outDir": "./built",
4. "allowJs": true,
5. "target": "es5"
6. },
7. "include": [
8. "./src/\*\*/\*"
9. ]
10. }

In the above file, we are specifying a few things to TypeScript:

* The **include** option reads all files in the **src** directory.
* The **allowJs** option accept all JavaScript files as inputs.
* The **outDir** specifies that all of the **output** files should be redirected in the built folder.
* The **target** option specifies that all JavaScript constructs should be translated into an older version like **ECMAScript 5.**

## 2. Integrate with a build tool

We know, most JavaScript projects have an integrated build tool like **gulp** or **webpack.**

#### **Note: Each build tool is different.**

We can integrate projects with **webpack** in the following ways:

**a) Run the following command on terminal:**

1. $ npm install awesome-typescript-loader source-map-loader

In webpack integration, we use **awesome-typescript-loader** (a TypeScript loader) combined with **source-map-loader** for easier debugging of source code.

**b) Merge the module config property in our webpack.config.js file to include the following loaders:**

1. module.exports = {
2. entry: "./src/index.ts",
3. output: {
4. filename: "./dist/bundle.js",
5. },
6. // Enable sourcemaps for debugging webpack's output.
7. devtool: "source-map",
8. resolve: {
9. // Add '.ts' and '.tsx' as resolvable extensions.
10. extensions: ["", ".webpack.js", ".web.js", ".ts", ".tsx", ".js"]
11. },
12. module: {
13. loaders: [
14. // All files with a '.ts' or '.tsx' extension will be handled by 'awesome-typescript-loader'.
15. { test: /\.tsx?$/, loader: "awesome-typescript-loader" }
16. ],
17. preLoaders: [
18. // All output '.js' files will have any sourcemaps re-processed by 'source-map-loader'.
19. { test: /\.js$/, loader: "source-map-loader" }
20. ]
21. },
22. // Other options...
23. };

## 3. Moving all .js files to .ts files

In this section, we have to rename our **.js** file to **.ts** file. Similarly, if our file uses **JSX,** we will need to rename it to **.tsx.** Now, if we open that file in an editor which support TypeScript, some of our code might start giving **compilation errors.** So, converting files one by one allows handling compilation errors more easily. If TypeScript finds any compilation errors during conversion, it still able to translate the code just like Word will print our documents.

## 4. Check for errors

After moving js file to ts file, immediately, TypeScript will start **Type Checking** of our code. So, we get **diagnostic** errors in our JavaScript code. Some of the errors we may encounter are listed below:

**a) We can suppress errors with using any, e.g.:**

In the below code, we can delete the error by using the **type assertion.**

1. var foo = 123;
2. var bar = 'hey';
3. bar = foo; // ERROR: cannot assign a number to a string
4. bar = foo as any  //Ok

**b) Function with less or more arguments:**

1. function display(name, age, height) {
2. let str1 = "Person named " + name + ",  " + age + " years old";
3. let str2 = (height !== undefined) ? (" and " + height +  " feet tall") : '';
4. console.log(str1 + str2);
5. }
6. display( "Rohit", 32);

In the above code, the function **display()** takes **three** arguments: name, age, and height. We can call this function with two values: "Rohit" and 23. It is perfectly valid with JavaScript because in JavaScript if an expected argument to a function is missing, it assigns the value undefined to the argument.

But, the same code in TypeScript will give the **compilation error: Expected three arguments but got two**. To remove this error, we can add an **optional parameter sign** to the argument height and annotate our code as below:

1. function display(name: string, age: number, height?: number) {
2. let str1: string = "Person named " + name + ",  " + age + " years old";
3. let str2: string = (height !== undefined) ? (" and " + height +  " feet tall") : '';
4. console.log(str1 + str2);
5. }

**c) Sequentially Added Properties**

The following code is very common in JavaScript.

1. var options = {};
2. options.color = "red";
3. options.volume = 11;

In TypeScript, the type of options as {} is an empty object. So, **color** and **volume** doesn't exist and are not assignable. If we instead moved the declarations into the object literal themselves, we would not get any errors:

1. let options = {
2. color: "red",
3. volume: 11
4. };

We can also define the type of options and add a **type assertion** on the object literal.

1. interface Options { color: string; volume: number }
3. let options = {} as Options;
4. options.color = "red";
5. options.volume = 11;

## 5. Using third party JavaScript Libraries

JavaScript projects use third-party libraries like **jQuery** or **Lodash.** In order to compile files, TypeScript needs to know the types of all objects in these libraries. We know, TypeScript Type definition files for JavaScript libraries are already available at **DefinitelyTyped.** So, we don't need to install this type externally. We need to install only those types which are used in our project.

**For example**

For **jQuery**, install the definition:

1. $ npm install @types/jquery

For **Lodash**, install the definition:

1. $ npm install -S @types/lodash

Once, we made the changes to our JavaScript project, run the **build tool.** Now, we should have our TypeScript project compiled into plain JavaScript that we can run in the browser.

# **Difference between TypeScript and ES6**

## TypeScript

TypeScript is an **open-source** pure object-oriented programing language. It is a strongly typed **superset** of JavaScript which compiles to plain JavaScript. TypeScript is developed and maintained by **Microsoft** under the **Apache 2** license. It is not directly run on the browser. It needs a compiler to compile and generate in JavaScript file. TypeScript source file is in ".ts" extension. We can use any valid "**.js**" file by renaming it to ".ts" file. TypeScript is the ES6 version of JavaScript with some additional features.

### **History of TypeScript**

**Anders Hejlsberg** developed TypeScript. It was first introduced for the public in the month of **1 October 2012**. After two years of internal development at Microsoft, the new version of TypeScript 0.9 was released in 2013. The current version of TypeScript is **TypeScript 3.4.5** which was released on **24 April 2019**.

## ES6

ECMAScript (ES) is a **scripting language** specification standardized by **ECMA international**. It was created to standardize JavaScript. The ES scripting language contains many implementations, and the most popular is **JavaScript**. The developers use ECMAScript mostly for **client-side scripting** of World Wide Web (WWW).

The **sixth** edition of ECMAScript standard is ECMAScript6 or ES6 and later renamed as **ECMAScript 2015**. It is a major enhancement to the JavaScript language, which allows us to write programs for complex applications. It adds many features intended to make large-scale software development easier. The most common ES6 web-browsers are **Chrome** and **Firefox**. A **transpiler** converts the ES6 based code into **ES5** which is supported many browsers. TypeScript is a transpiler. Grunt, Gulp, and Babel are some other transpilers to compile the modules. Therefore, TypeScript supports ES6.

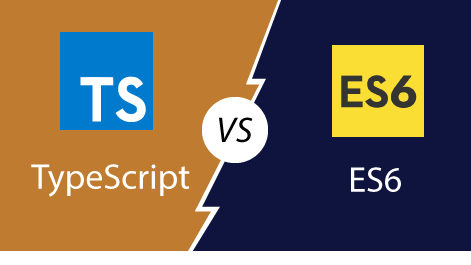
### **History**

JavaScript introduced in **May 1995** by **Brendan Eich**. It was initially called **Mocha**, a name chosen by **Netscape** founder **Marc Andreessen**, and later renamed to **LiveScript**. At the same time, **Sun Microsystems** owned the trademark for JavaScript. In **December 1995**, Netscape acquired a trademark license and renamed it to JavaScript.

In between **1996** and **1997**, Netscape took JavaScript to the ECMA standards organization to maintain a specification for the language. In **June 1997**, the ECMA Technical Committee 39 (TC39) was created to continue to evolve the language, eventually releasing **ECMA-262 Ed.1**.

The first standard version of JavaScript was ECMAScript 1 was released on **June 1997**. After a year later, ECMAScript 2 was released, which contains only minor changes to keep a parallel ISO standard for JavaScript. In **December 1999**, ECMAScript 3 was released, which introduces a lot of popular features of JavaScript. In **December 2009**, ECMAScript or ES6 was published and subsequently renamed to ECMAScript 2015.

## TypeScript vs. ES6



|  |  |  |
| --- | --- | --- |
|  | **TypeScript** | **ES6** |
| **Definition** | TypeScript is a free and open-source pure object-oriented programming language. It is developed and maintained by Microsoft. | ES6 is a version of ECMAScript (ES), which is a scripting language specification standardized by ECMA international. |
| **Explanation** | Typescript is to eradicate the development errors. | ES6 is comparatively more flexible in development time. |
| **Data-Types** | TypeScript supports all primitive data types. | ES6 does not support all data types. |
| **Features** | TypeScript contains features such as generics and type annotations, Inference, Enums, and Interfaces. | ES6 does not support these features. |
| **Scope** | Typescript has three scopes.   1. Global Scope 2. Class Scope 3. Local Scope | ES6 has two scopes.   1. Global Scope 2. Local Scope |
| **Decision-Making** | 1. if Statement 2. if-else Statement 3. else...if and nested if statements 4. switch Statement | 1. if Statement 2. if-else Statement 3. The else- if ladder/nested if statements. 4. switch?case Statement |
| **Modules** | TypeScript Modules are of two types:   1. Internal 2. External modules | We can classify the ES6 modules in two ways:   1. Importing a module 2. Exporting a module |
| **Loop** | Typescript and ES6 both are having same loops.   1. Definite 2. Indefinite | Typescript and ES6 both are having same loops.   1. Definite 2. Indefinite |
| **Why choose** | The developers choose TypeScript:   * Typesafe * JavaScript superset * Powerful type system, including generics & JS features. * Aligned with ES development for compatibility. * Structural, rather than nominal, subtyping. * Compile-time errors. * Starts and ends with JavaScript. | The developers choose ES6:   * ES6 code is shorter than traditional JS * Module System Standardized * Extremely compact * Destructuring Assignment |
| **Company using** | The list of companies which uses TypeScript are:   * Slack * Asana * CircleCI * Intuit * Swat.io * Avocode | The list of companies which uses ES6 are:   * Slack * StackShare * eBay * Asana * Intuit * Swat.io |