TypeScript

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.

6.7M

170

C++ vs Java

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 Installation

### 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://nodejs.org/en/download/

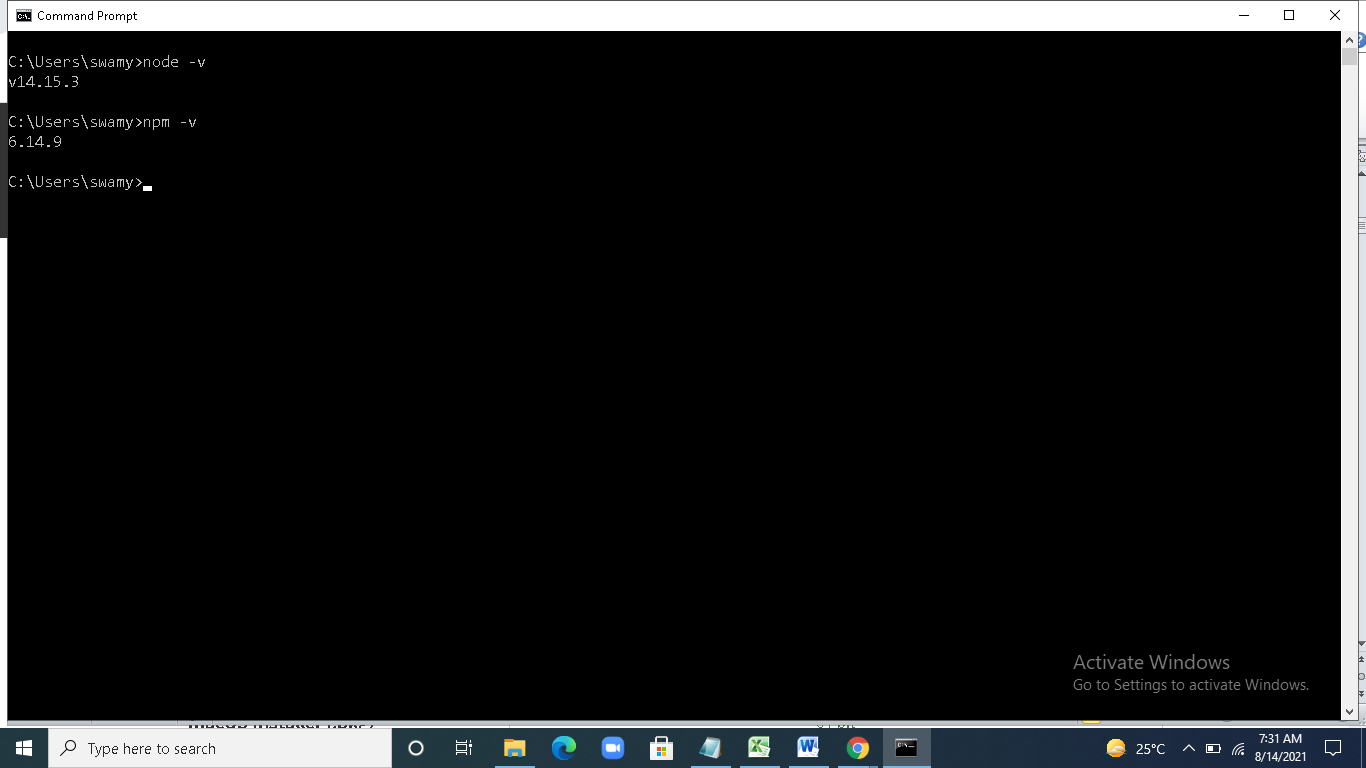
6.9M

151

HTML Tutorial

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

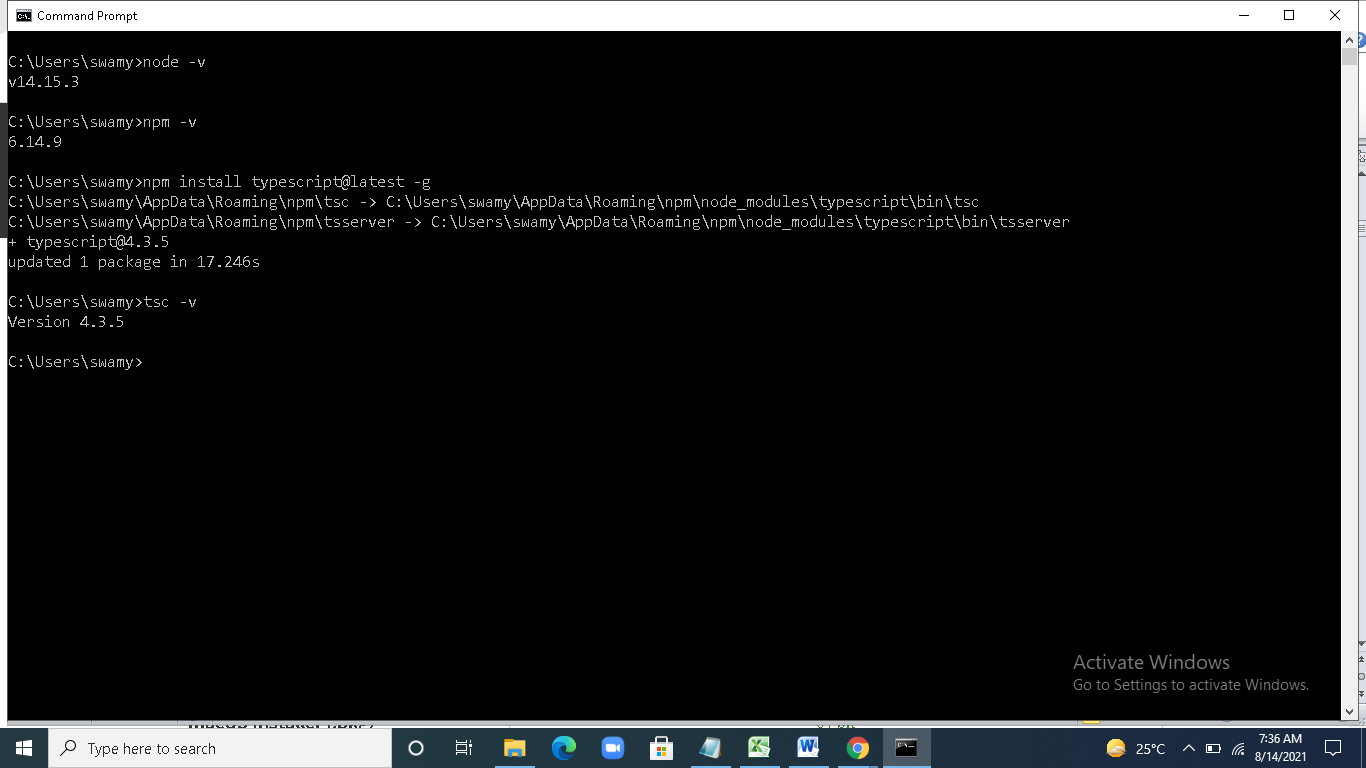
1. node -v
2. npm -v



**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.

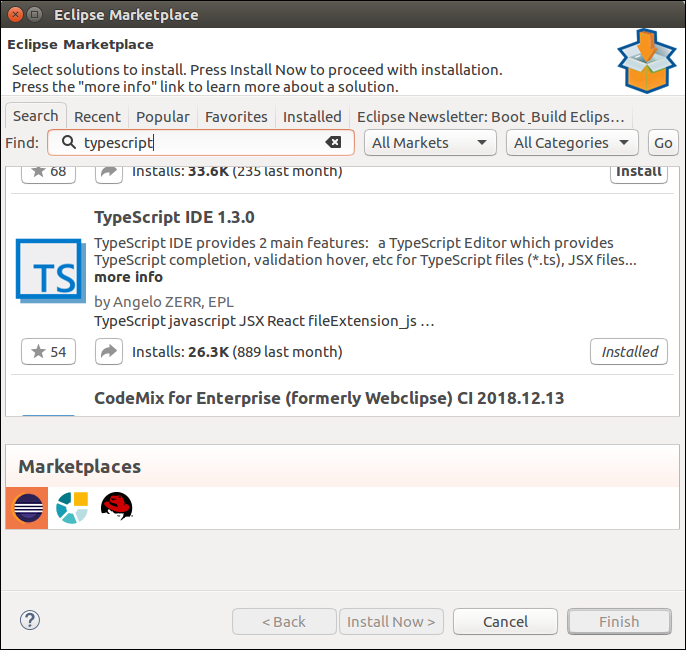


### Install TypeScript plug-in in your IDE

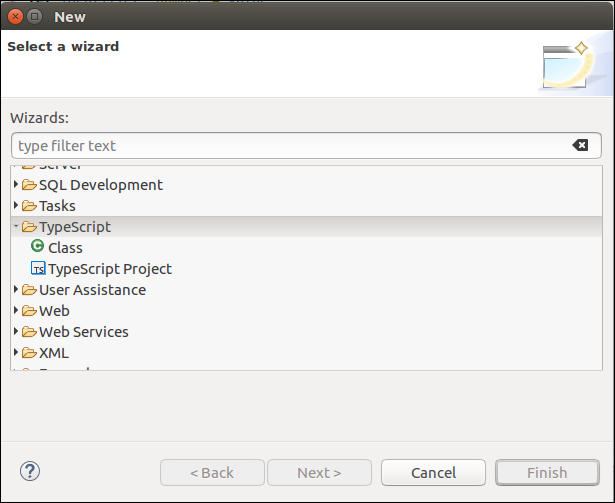
**Step-1** Install IDE like Eclipse, Visual Studio, WebStorm, Atom, Sublime Text, etc. Here, we install Eclipse. To install Eclipse, go to the following link:

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

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



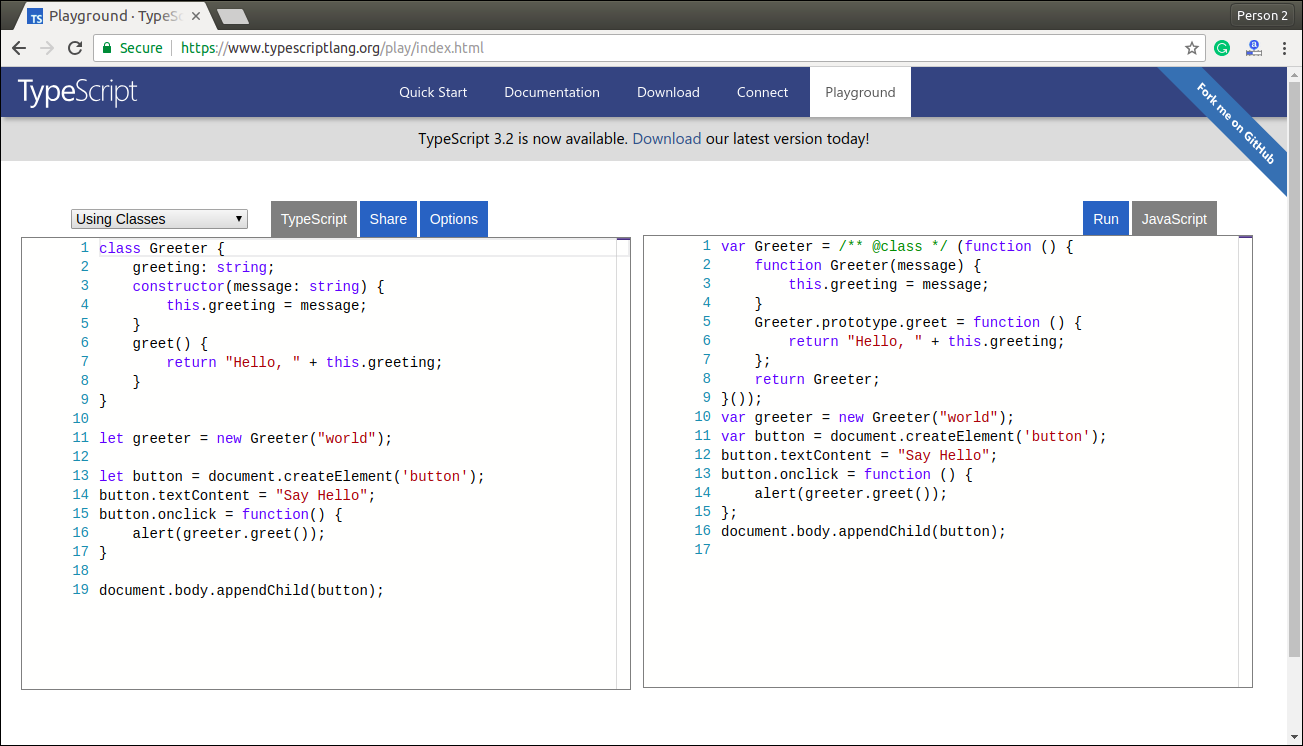
* In the next window, select **Features** which you want to install, and click **Confirm**.
* A new window will open, select **Accept Terms and Condition**, Click **Next**, and follow the on-screen instructions.
* Now **restart** Eclipse. To verify the TypeScript, go to **New->Other->TypeScript**. Once 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 FirsApplication

function greeter(person) {

return "Hello, " + person;

}

let user = 'Welcome to TP';

console.log(greeter(user));

-----------------------------------------------------

--Save the above file as ".ts" extension

--Compile the TypeScript code.

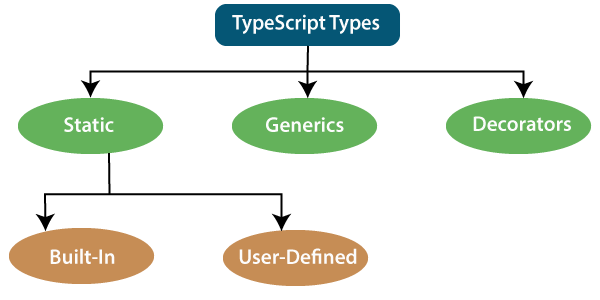
>tsc filename.ts (for VisualStudioCode Terminal--🡪 tsc.cmd filename.ts)

>node filename.js

# 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

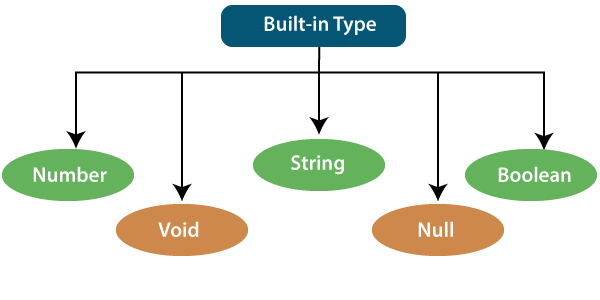
In the context of type systems, static types mean "at compile time" or "without running a program." In a statically typed language, variables, parameters, and objects have types that the compiler knows at compile time. The compiler used this information to perform the type checking.

Static types can be further divided into two sub-categories:88

History of Java

### 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:**

let identifier: number = value;

**Examples:-**

let first: number = 12.0;             // number

let second: number = 0x37CF;          // hexadecimal

let third: number = 0o377 ;           // octal

let fourth: number = 0b111001;        // binary

console.log(first);           // 12

console.log(second);          // 14287

console.log(third);           // 255

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**

let identifier: string = " ";

                Or

let identifier: string = ' ';

**Examples**

let empName: string = "Rohan";

let empDept: string = "IT";

// Before-ES6

let output1: string = employeeName + " works in the " + employeeDept + " department.";

// After-ES6

let output2: string = `${empName} works in the ${empDept} department.`;

console.log(output1);//Rohan works in the IT department.

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**

let identifier: Boolean = Boolean value;

**Examples**

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**

let unusable: void = undefined;

**Examples**

 function helloUser(): void {

       alert("This is a welcome message");

       }

 let tempNum: void = undefined;

  tempNum = null;

  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**

let num: number = null;

let bool: boolean = null;

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**

let num: number = undefined;

let bool: boolean = undefined;

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**

let identifier: any = value;

**Examples**

1. let val: any = 'Hi';

      val = 555;   // OK

      val = true;   // OK

2. function ProcessData(x: any, y: any) {

                       return x + y;

            }

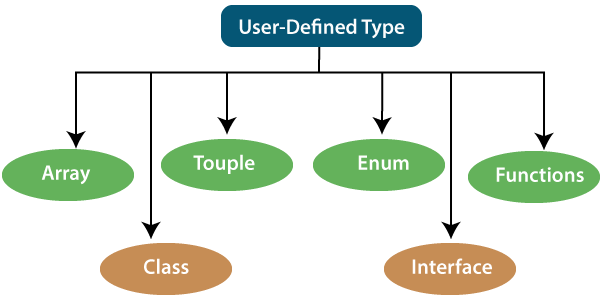
            let result: any;

            result = ProcessData("Hello ", "Any!"); //Hello Any!

            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:

var list : number[] = [1, 3, 5];

2. The second way uses a generic array type:

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

### Tuple

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:

// Declare a tuple

let a: [string, number];

// Initialize it

a = ["hi", 8]; // 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**

interface Calc {

    subtract (first: number, second: number): any;

}

let Calculator: Calc = {

    subtract(first: number, second: number) {

        return first - second;

    }

}

console.log(Calculator.subtract(10,5))

### 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**

class Student

{

    rollNo: number;

    name: string;

    constructor(rollNo: number, name: string)

    {

        this.rollNo = rollNo;

        this.name = name;

    }

    showDetails()

    {

        console.log(this.rollNo + " : " + this.name);

    }

}

let student=new Student(10,'vijay');

student.showDetails();

### 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**

enum Color {

    Red, Green, Blue

};

let c: Color;

c= Color.Blue;

console.log(c)

### 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**

//named function with number as parameters type and return type

function add(a: number, b: number): number {

    return a + b;

}

//anonymous function with number as parameters type and return type

let sum = function (a: number, y: number): number {

    return a + y;

};

console.log(sum(5,4))

## 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**

function identity**<T>**(arg: T): T {

    return arg;

}

let output1 = identity**<string>**("myString");

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**

function f() {

    console.log("f(): evaluated");

    return function (target, propertyKey: string, descriptor: PropertyDescriptor) {

        console.log("f(): called");

    }

}

class C {

    @f()

    method() {}

}

# 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

//Variable declared and assigned to null

var a = null;

console.log( a );   //output: null

console.log( typeof(a) );   //output: object

**Output:**

null

object

## 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

//Variable declaration without assigning any value to it

var a;

console.log(a);  //undefined

console.log(typeof(a));  //undefined

console.log(undeclaredVar);  //Uncaught ReferenceError: undeclaredVar is not defined

**Output:**

Error will come

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 re-assignment to a variable.

Variable Declaration

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

7.7M

141

Difference between JDK, JRE, and JVM

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

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

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

var [identifier] : [type-annotation];

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

var [identifier] = value;

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

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.

var x = 50;

We can also declare a variable inside the function:

function a() {

     var msg = " Welcome to TypeScript !! ";

     return msg;

}

a();

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

function a() {

    var x = 50;

    return function b() {

         var y = x+5;

         return y;

    }

}

var  b = a();

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**

function f()

{

    var X = 5; //Available globally inside f()

    if(true)

    {

        var Y = 10; //Available globally inside f()

        console.log(X); //Output 5

        console.log(Y); //Output 10

    }

    console.log(X); //Output 5

    console.log(Y); //Output 10

}

f();

console.log(X); //Returns undefined because value cannot accesses from outside function

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:

var declaration: var b = 50;

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.

function f(input: boolean) {

    let x = 100;

    if (input) {

        // "x" exists here

        let y = x + 1;

        return y;

    }

    // Error: "y" doesn't exist here

    return y;

}

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:

try {

    throw "Hi!!";

}catch (e) {

    console.log("Hello");

}

// 'e' doesn't exist here, so error will found

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:**

function f(a) {

    var a;

    var a;

    if (true) {

        var a;

    }

}

**Example with let keyword:**

let a = 10;

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**

var currencySymbol = "$";

function showMoney(amount) {

  var currencySymbol = "€";

  document.write(currencySymbol + amount);

}

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**

function get(x){

  console.log(a);  //printing x variable. Value is undefined

  //declared variable after console hoisted to the top at run time

  var a = x;

  //again printing x variable. Value is 4.

  console.log(a);

}

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**

{

  //program doesn't know about variable b so it will give me an error.

  console.log(b); // ReferenceError: b is not defined

  let b = 3;

}

## 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**

function constTest(){

  const VAR = 10;

  console.log("Value is: " +VAR);

}

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**

function constTest(){

  const VAR = 10;

  console.log("Output: " +VAR);  // Output: 10

  const VAR = 10;

  console.log("Output: " +VAR);  //Uncaught TypeError: Assignment to constant variable

}

constTest();

**Output:**

SyntaxError: Identifier 'VAR' has already been declared.

# 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

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(/).

7.7M

141

Difference between JDK, JRE, and JVM

**Next**

**Stay**

|  |  |  |  |
| --- | --- | --- | --- |
| **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'); //false |
| === | 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

let num = 16;

let result = (num **>** 0) ? "True":"False"

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 " + "TypeScript";
2. console.log("Result of String Operator: " +message);

**Output:**

Result of String Operator: Welcome to TypeScript

## 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 " + "TypeScript";  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.int

var age: number = 44;          // number variable

var name: string = "Rahul";     // string variable

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

function display(id:number, name:string)

{

    console.log("Id = " + id + ", Name = " + name);

}

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

:{ /\*Structure\*/ }

### Example

var student : {

    id: number;

    name: string;

};

student = {

  id: 100,

  name : "John"

}

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

Array declaration

1. Using square brackets.

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

Example:

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

2. Using a generic array type.

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

Example:

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

Example

let arr:number[];

arr = [1, 2, 3, 4]

console.log("Array[0]: " +arr[0]);

console.log("Array[1]: " +arr[1]);

Example

var mArray:number[][] = [[1,2,3],[5,6,7]] ;

console.log(mArray[0][0]);

console.log(mArray[0][1]);

console.log(mArray[0][2]);

console.log();

console.log(mArray[1][0]);

console.log(mArray[1][1]);

console.log(mArray[1][2]);

------------Array Object

Array objects allow us to store multiple values in a single variable

Syntax

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

Example

//array by using the Array object.

let arr:string[] = new Array("TP","2200","Java","Abhishek");

for(var i = 0;i<arr.length;i++) {

console.log(arr[i]); }

-------------------------------------------------------------

Array Traversal by using a for...in loop

Example

let i:any;

let arr:string[] = ["AB", "2300", "Java", "Abhishek"];

for(i in arr) {

console.log(arr[i])

}

---------------------------------------------------------------------

Passing Arrays to Functions

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

Example

let arr:string[] = new Array("Ja", "2300", "Java", "Abhishek");

//Passing arrays in function

function display(arr\_values:string[]) {

for(let i = 0;i<arr\_values.length;i++) {

console.log(arr[i]);

}

}

//Calling arrays in function

display(arr);

-------------------------------------------

TypeScript Spread operator

The spread operator is used to initialize arrays and objects from another array or object

EX:

let arr1 = [ 1, 2, 3];

let arr2 = [ 4, 5, 6];

//Create new array from existing array

let copyArray = [...arr1];

console.log("CopiedArray: " +copyArray);

//Create new array from existing array with more elements

let newArray = [...arr1, 7, 8];

console.log("NewArray: " +newArray);

//Create array by merging two arrays

let mergedArray = [...arr1, ...arr2];

console.log("MergedArray: " +mergedArray);

-----------------------------------------------------------------------------------------------

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.

Syntax

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

Example

let arrTuple = [101, "Vijay", 105.99, "Abhishek"];

console.log(arrTuple);

Accessing tuple Elements

Example

let empTuple = ["Rohit Sharma", 25, "Java"];

console.log("Name of the Employee is : "+empTuple [0]);

console.log("Age of the Employee is : "+empTuple [1]);

console.log(empTuple [0]+" is working in "+empTuple [2]);

----------------------------------------------------------------------

Operations on Tuple

A tuple has two operations:

Push()

Pop()

Example

let empTuple = ["Rohit Sharma", 25, "Java"];

console.log("Items: "+empTuple);

console.log("Length of Tuple Items before push: "+empTuple.length); // returns the tuple size

empTuple.push(10001); // append value to the tuple

console.log("Length of Tuple Items after push: "+empTuple.length);

console.log("Items: "+empTuple);

--------------------------------------------------------------------------------------------

TypeScript Union

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.

Syntax

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

Example

let value: number|string;

value = 120;

console.log("The Numeric value of a value is: "+value);

value = "Welcome to TP";

console.log("The String value of a value is: "+value);

------------------------------------------------------------------------------------------------

Passing Union Type in Function Parameter

Example

function display(value: (number | string))

{

if(typeof(value) === "number")

console.log('The given value is of type number.');

else if(typeof(value) === "string")

console.log('The given value is of type string.');

}

display(123);

display("ABC");

---------------------------------------------------------------------

Passing Union Type to Arrays

TypeScript allows passing a union type to an array.

Example

let arrType:number[]|string[];

let i:number;

arrType = [1,2,3,4];

console.log("Numeric type array:")

for(i = 0;i<arrType.length;i++){

console.log(arrType[i]);

}

arrType = ["India","America","England"];

console.log("String type array:")

for(i = 0;i<arrType.length;i++){

console.log(arrType[i]);

}

--------------------------------------------------------------------------------------------------------

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

array.forEach(callback[, thisObject]);

Parameter Details

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

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.

Example with string

let apps = ['WhatsApp', 'Instagram', 'Facebook'];

let playStore = [];

apps.forEach(function(item){

playStore.push(item)

});

console.log(playStore);

---------------------------

Example with number

var num = [5, 10, 15];

num.forEach(function (value) {

console.log(value);

});

Disadvantage of forEach()

It does not provide a way to stop or break the forEach() loop.

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.

Map Methods

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

Ex:

let map = new Map();

map.set('1', 'abhishek');

map.set(1, 'www.abc.com');

map.set(true, 'bool1');

map.set('2', 'ajay');

console.log( "Value1= " +map.get(1) );

console.log("Value2= " + map.get('1') );

console.log( "Key is Present= " +map.has(3) );

console.log( "Size= " +map.size );

console.log( "Delete value= " +map.delete(1) );

console.log( "New Size= " +map.size );

execution:

> tsc -target es6 a.ts

>node a.js

-----------------------------------------------------------------------

Iterating Map Data

We can iterate over map keys or values or entries by using 'for...of' loop.

Example

let ageMapping = new Map();

ageMapping.set("Rakesh", 40);

ageMapping.set("Abhishek", 25);

ageMapping.set("Amit", 30);

//Iterate over map keys

for (let key of ageMapping.keys()) {

console.log("Map Keys= " +key);

}

//Iterate over map values

for (let value of ageMapping.values()) {

console.log("Map Values= " +value);

}

console.log("The Map Enteries are: ");

//Iterate over map entries

for (let entry of ageMapping.entries()) {

console.log(entry[0], entry[1]);

}

------------------------------------------------------------------------------------

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.

Methods in set

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.

EX:

let studentEntries = new Set();

//Add Values

studentEntries.add("John");

studentEntries.add("Peter");

studentEntries.add("Gayle");

studentEntries.add("Kohli");

studentEntries.add("Dhawan");

//Returns Set data

console.log(studentEntries);

//Check value is present or not

console.log(studentEntries.has("Kohli"));

console.log(studentEntries.has(10));

//It returns size of Set

console.log(studentEntries.size);

//Delete a value from set

console.log(studentEntries.delete("Dhawan"));

//Clear whole Set

studentEntries.clear();

//Returns Set data after clear method.

console.log(studentEntries);

--------------------------------------------------------------------------------------

Chaining of Set Method

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

Example

let studentEntries = new Set();

//Chaining of add() method is allowed in TypeScript

studentEntries.add("John").add("Peter").add("Gayle").add("Kohli");

//Returns Set data

console.log("The List of Set values:");

console.log(studentEntries);

------------------------------------------------------------------------------------

Iterating Set Data

We can iterate over set values or entries by using 'for...of' loop.

Example

let diceEntries = new Set();

diceEntries.add(1).add(2).add(3).add(4).add(5).add(6);

//Iterate over set entries

console.log("Dice Entries are:");

for (let diceNumber of diceEntries) {

console.log(diceNumber);

}

// Iterate set entries with forEach

console.log("Dice Entries with forEach are:");

diceEntries.forEach(function(value) {

console.log(value);

});

----------------------------------------------------------------------------------

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.

Syntax:

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

Arrow function with parameter

-----------------------------

Ex:

let sum = (a: number, b: number): number => {

return a + b;

}

console.log(sum(20, 30)); //returns 50

Note:

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.

Ex:

let sum = (a, b) => {

return a + b;

};

console.log(sum(20, 20)); //returns 50

Arrow function without parameters.

----------------------------------

let Print = () => console.log("Hello ");

Print();

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.

Ex:

let sum = (a: number, b: number) => a + b;

console.log("SUM: " +sum(5, 15));

Arrow function in a class

-------------------------

We can include the arrow function as a property in a class.

class Student {

studCode: number;

studName: string;

constructor(code: number, name: string) {

this.studName = name;

this.studCode = code;

}

showDetail = () => console.log("Student Code: " + this.studCode + '\nStudent Name: ' + this.studName)

}

let stud = new Student(101, 'Abhishek Mishra');

stud.showDetail();

TypeScript Classes

==================

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 has built-in support for using classes because it is based on ES6 version of JavaSript

Ex:

class Student {

studCode: number;

studName: string;

constructor(code: number, name: string) {

this.studName = name;

this.studCode = code;

}

getGrade() : string {

return "A+" ;

}

}

Ex:

class Student {

//defining fields

id: number;

name:string;

//creating method or function

display():void {

console.log("Student ID is: "+this.id)

console.log("Student ID is: "+this.name)

}

}

//Creating an object or instance

let obj = new Student();

obj.id = 101;

obj.name = "Virat Kohli";

obj.display();

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.

Ex:

class Student {

//defining fields

id: number;

name:string;

//defining constructor

constructor(id: number, name:string) {

this.id = id;

this.name = name;

}

//creating method or function

display():void {

console.log("Function displays Student ID is: "+this.id)

console.log("Function displays Student ID is: "+this.name)

}

}

//Creating an object or instance

let obj = new Student(101, "Virat Kohli")

//access the field

console.log("Reading attribute value of Student as: " +obj.id,)

console.log("Reading attribute value of Student as: " +obj.name)

//access the method or function

obj.display()

Example without constructor

----------------------------

//Defining a Student class.

class Student {

//defining fields

id: number;

name:string;

}

//Creating an object or instance

let obj = new Student();

// Initializing an object

obj.id = 101;

obj.name = "Virat Kohli";

//access the field

console.log("Student ID: " +obj.id,);

console.log("Student Name: " +obj.name);

Interface Declaration

---------------------

We can declare an interface as below.

interface interface\_name {

// variables' declaration

// methods' declaration

}

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.

Ex:

interface OS {

name: String;

language: String;

}

let OperatingSystem = (type: OS): void => {

console.log('Android ' + type.name + ' has ' + type.language + ' language.');

};

let Oreo = {name: 'O', language: 'Java'}

OperatingSystem(Oreo);

TypeScript Module

-----------------

JavaScript has a concept of modules from ECMAScript 2015.

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 we can use in other modules by using the import keyword.

Module declaration

------------------

We can declare a module by using the export keyword.

Syntax:

//FileName : Employee.ts

export class/interface Employee {

//code declarations

}

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.

Syntax:

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

Example

example:

Module Creation: addition.ts

export class Addition{

constructor(private x?: number, private y?: number){

}

Sum(){

console.log("SUM: " +(this.x + this.y));

}

}

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

import {Addition} from './addition';

let addObject = new Addition(10, 20);

addObject.Sum();

Importing multiple modules in single file

-----------------------------------------

We can define multiple modules in a single file.

Syntax:

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

Example

Module Creation: Modules.ts

export class Addition{

constructor(private x?: number, private y?: number){

}

Sum(){

console.log("SUM: " +(this.x + this.y));

}

}

export class Substraction{

constructor(private a?: number, private b?: number){

}

Substract(){

console.log("SUBSTRACTION: " +(this.a - this.b));

}

}

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

import {Addition, Substraction} from './Modules';

let addObject = new Addition(10, 20);

let subObject = new Substraction(20, 10);

addObject.Sum();

subObject.Substract();