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).



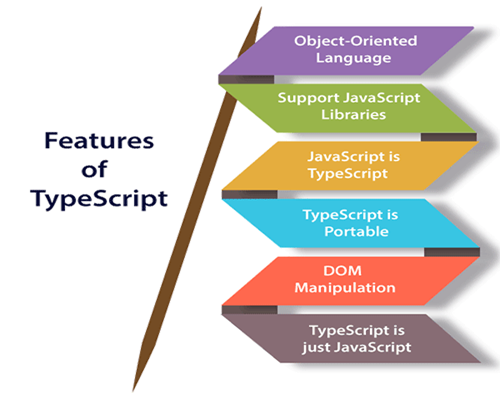
## Why use TypeScript?

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.

|  |  |
| --- | --- |
| **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.lo |

# Features of TypeScript



# 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)

19.7M

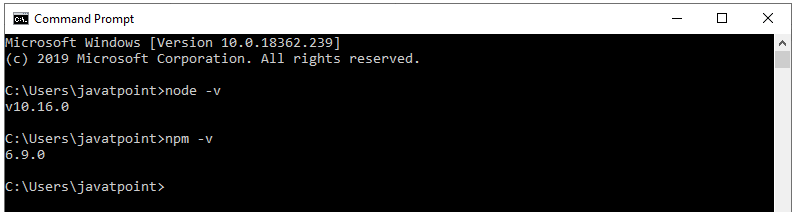
414

Java Try Catch

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.

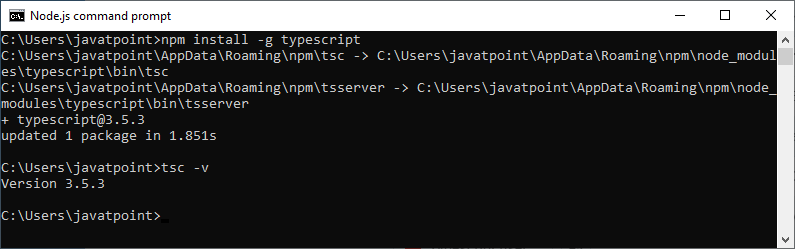
1. $ node -v
2. $ npm -v



**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:

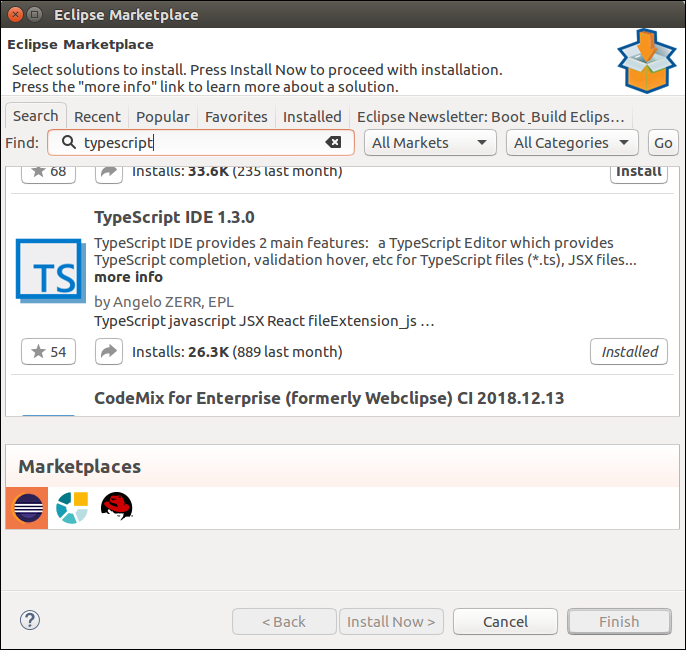
**In Windows:** [**https://www.javatpoint.com/javafx-how-to-install-eclipse**](https://www.javatpoint.com/javafx-how-to-install-eclipse)

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

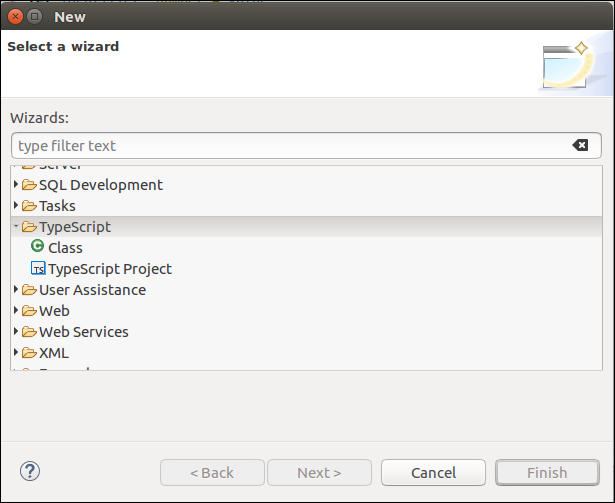
**In CentOS:** [**https://www.javatpoint.com/how-to-install-eclipse-on-centos**](https://www.javatpoint.com/how-to-install-eclipse-on-centos)

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

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



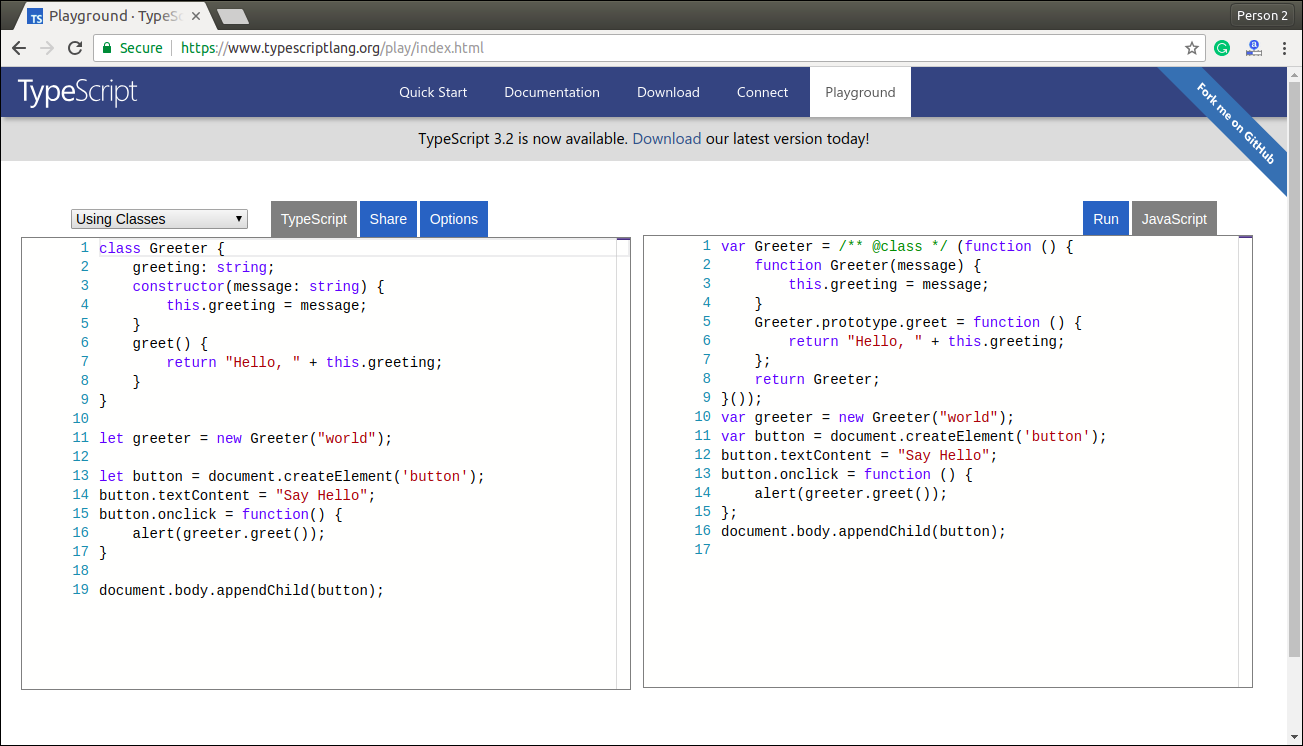
* In the next window, select **Features** which you want to install, and click **Confirm**.
* A new window will open, select **Accept Terms and Condition**, Click **Next**, and follow the on-screen instructions.
* Now **restart** Eclipse. To verify the TypeScript, go to **New->Other->TypeScript**. Once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



## Online Compiler for TypeScript

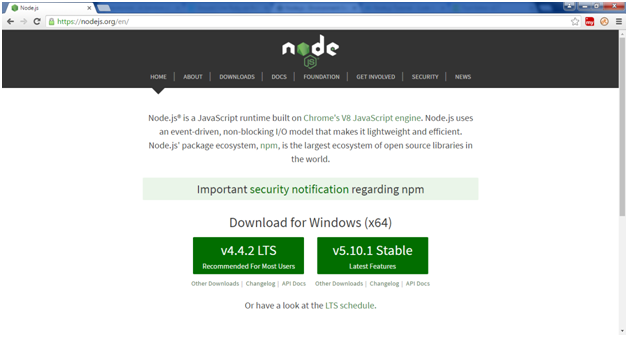
We can also run our script online with the official compiler. To do this, go to the below link. [**https://www.typescriptlang.org/play/index.html**](https://www.typescriptlang.org/play/index.html)

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

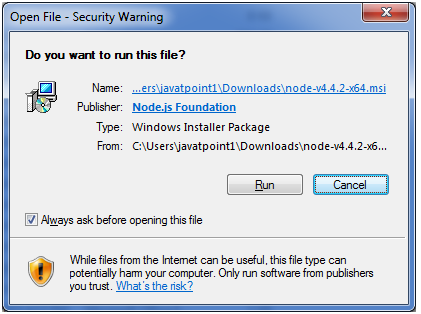
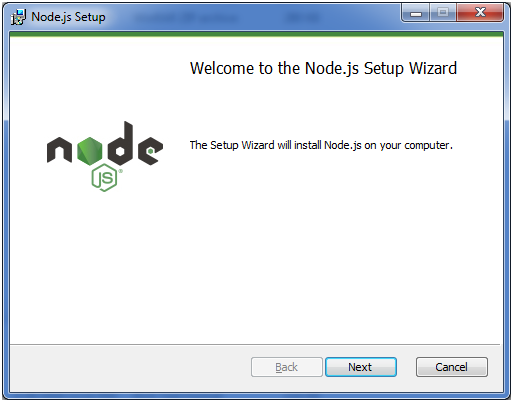


**How to download Node.js:**

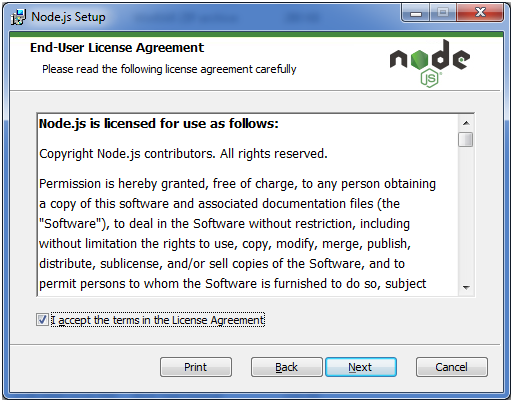
You can download the latest version of Node.js installable archive file from <https://nodejs.org/en/>



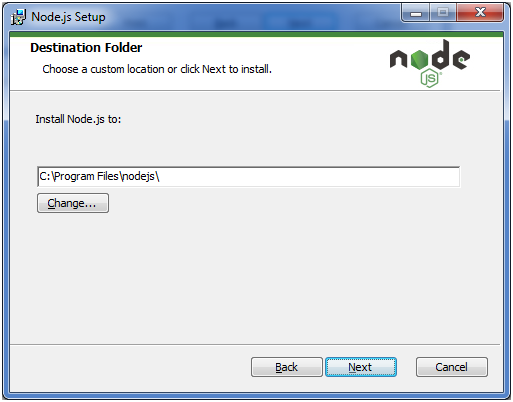
Here, you deploy the installation of node-v4.4.2 LTS recommended for most users.

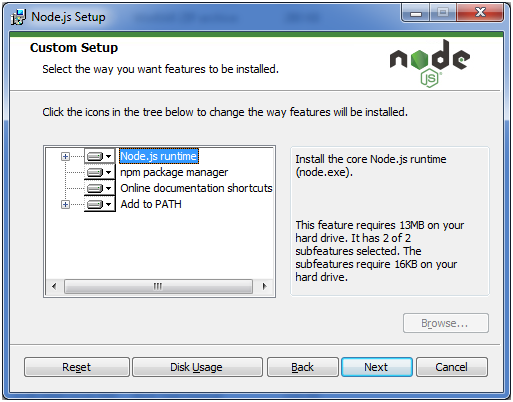
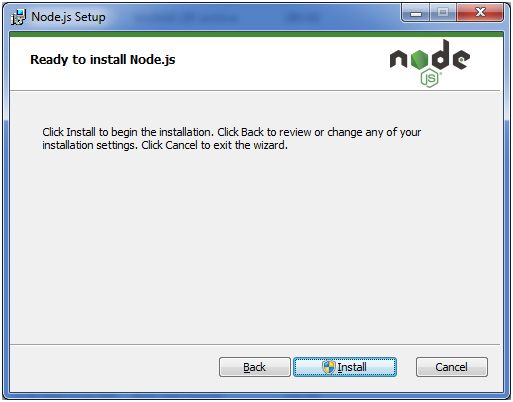
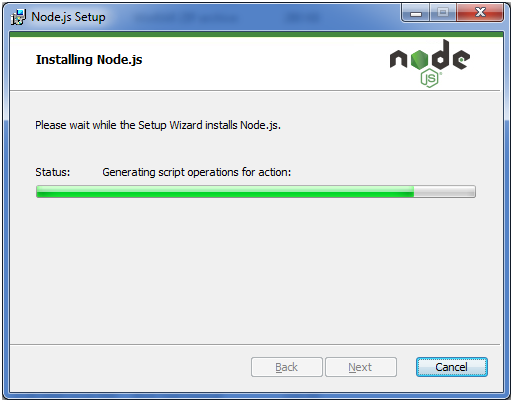
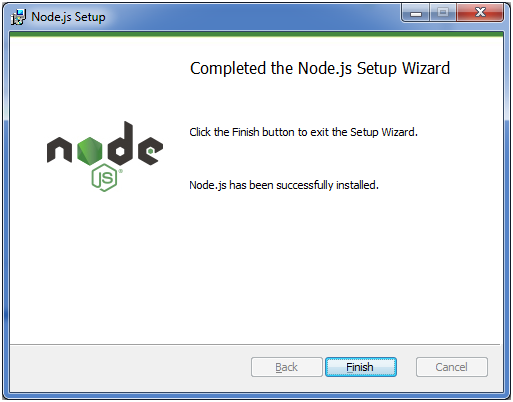
Accept the terms of license agreement.



Choose the location where you want to install.



Ready to install:

Typescript installation:

Open cmd and execute below commands:

npm install typescript --save-dev

 npm install typescript –g

npm install typescript@latest –g

to check thetypescript version below command:

tsc –v

First Example on TypeScript:

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.

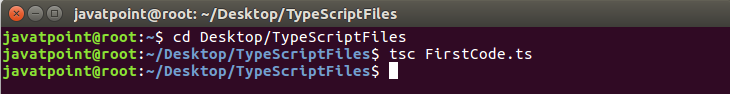
**Step-1** Open the Text Editor and write/copy the following code.

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

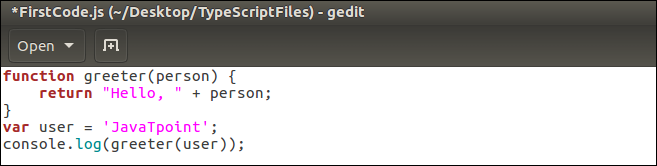
**Step-2** Save the above file as "**.ts**" extension.

Exception Handling in Java - Javatpoint

**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**.

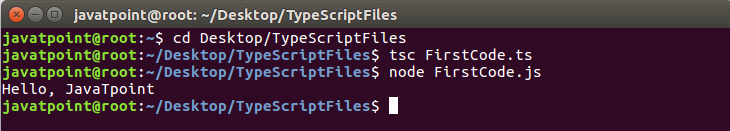


It will generate JavaScript file with ".js" extension at the same location where the TypeScript source file exists. The below ".js" file is the output of TypeScript (.ts) file.



#### 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.

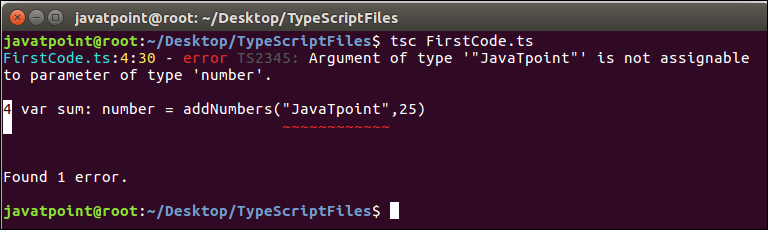
**Step 1** Open the Text Editor and write/copy the following code.

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

let:

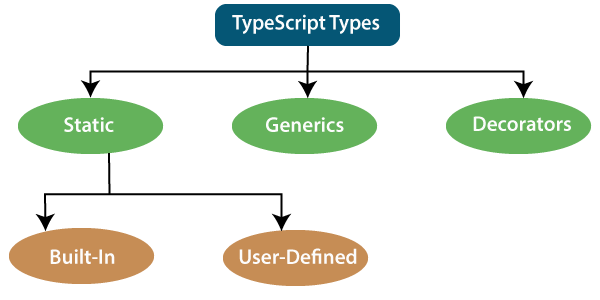
Variables defined with let cannot be Redeclared.

Variables defined with let must be Declared before use.

Variables defined with let have Block Scope.

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.



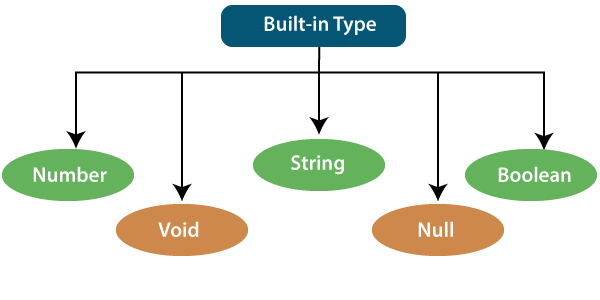
## 1. Static Types

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**

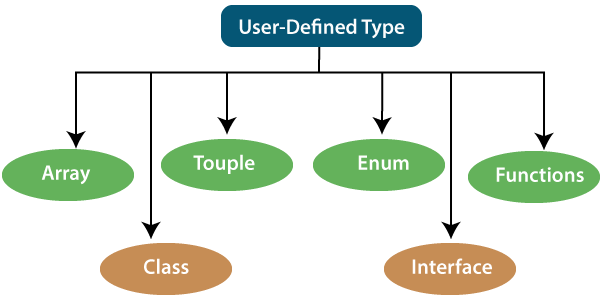
1. let identifier: any = value;

**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 );

# Difference between Null and Undefined

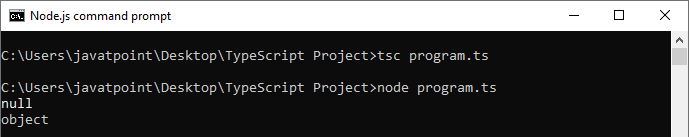
## Null

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:**



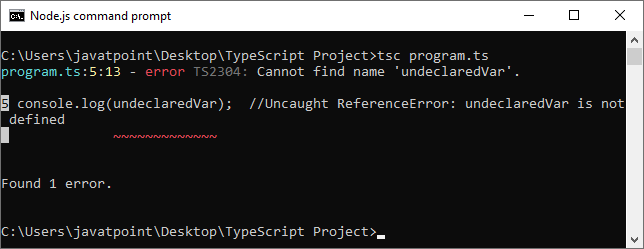
## Undefined

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 Javapoint !! ";
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);

# 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(/).

Competitive questions on Structures

|  |  |  |  |
| --- | --- | --- | --- |
| **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.

16.7M

335

varvavarcvaHistory of Java

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.

00:22/04:58

10 Sec

19.9M

426

HTML Tutorial

**Let us take another example.**

1. var x = "Javapoint";
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 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:

10 Sec

How to find Nth Highest Salary in SQL

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.

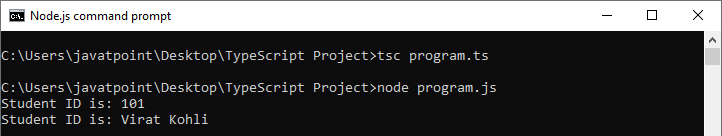
**Advantage of Method**

* 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.

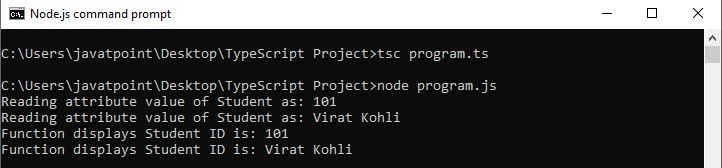
**Example**

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()

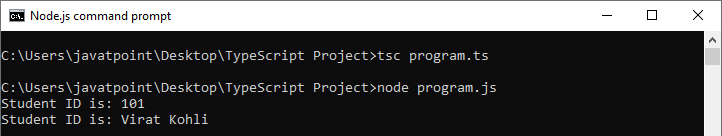
**Output:**



**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);

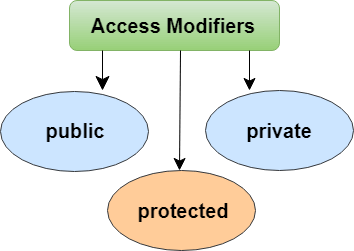
**Output:**



## 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:



To read more information about the access modifier,

# 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.

19.8M

416

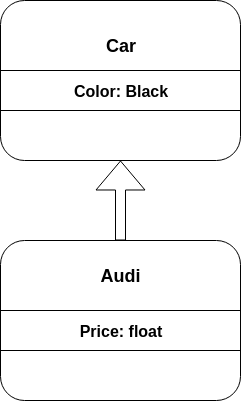
Java Try Catch

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.

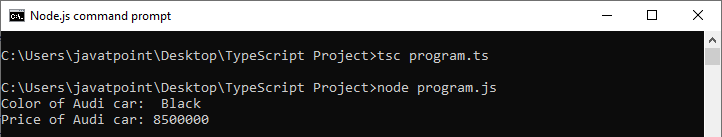
**Inheritance Example**



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:**

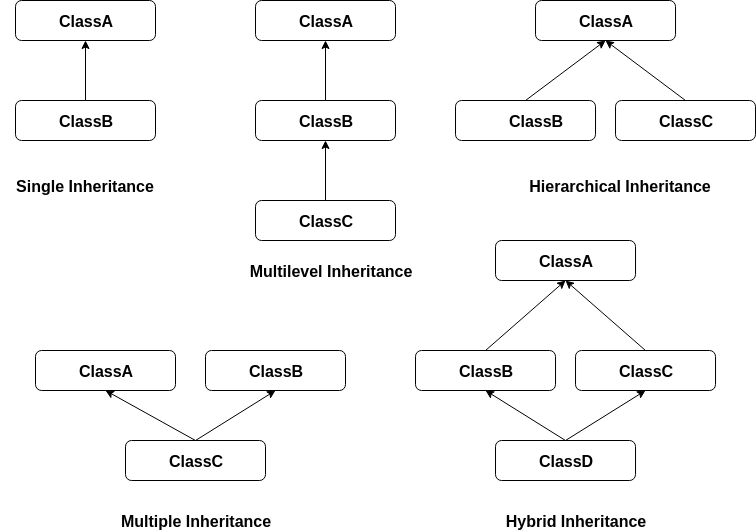


In the above example, the **Audi class** extends the **Car class** by using the **extends** keyword. It means the Audi class can include all the members of the Car class. The constructor of the Audi class initializes its own members as well as the parent class's properties by using a special keyword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



#### 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()

# 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.

20.7M

460

How to find Nth Highest Salary in SQL

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);