**TypeScript**

Install typescript on cmd.

C:\Users\Sravya>npm install -g typescript

tsc -v

Typescript is extended Javascript.

Javascript is dynamically type. Can know the type in runtime.

Typescript has type support and can identify the type during compilation.

The types are optional is typescript.

Enhanced IDE support.

Typescript is the main programming language of angular.

Download and install NodeJS.

Check node version using node -v on cmd.

Then install typescript using the following command.

npm install -g typescript

tsc -v

Open Visual Studio and Point to the folder . Create a new file called Main.ts

let message='Hello World';

console.log(message);

Goto the terminal and type tsc main.ts

The command creates a new javascript file called Main.js

The typescript file gets transpiled to javascript.

Now in the terminal type node main.js and you should be able to see the output.

You can see there will be an error below message as the file is treated as a script rather than a module.

To get rid of it include an export statement on the top.

export {}

let message:string='Hello World';

console.log(message);

To automatically recompile a typescript file in the terminal run the command tsc Main –watch

Variable Declarations:

Let and const keywords are used to declare variables.

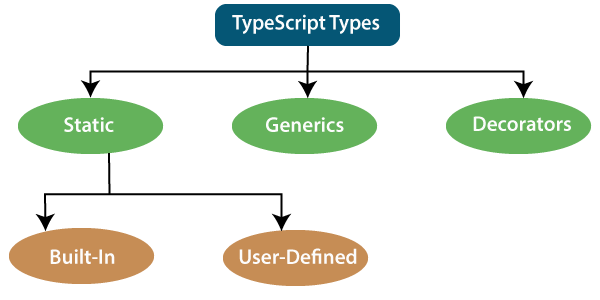
Cant redeclare the variables again.

Variable Types:

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.

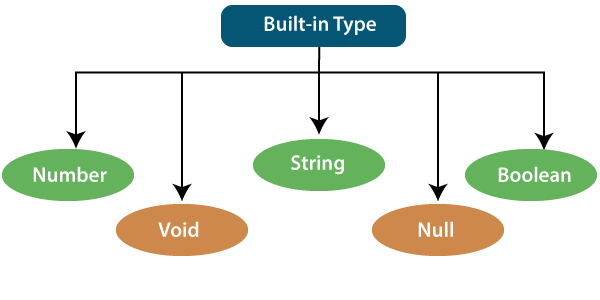


1. Static Types

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.

### **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:-**

let first: number = 12.0;             // number

let second: number = 0x37CF;          // hexadecimal

let third: number = 0o377 ;           // octal

let fourth: number = 0b111001;        // binary

console.log(first);           // 123

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**

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**

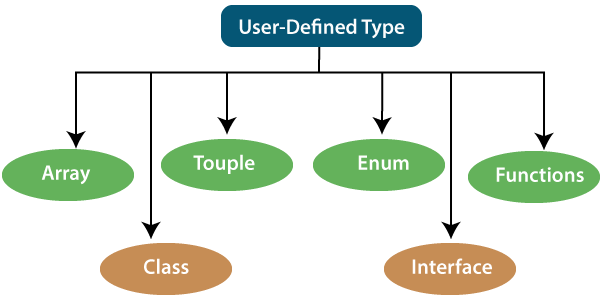
1. let identifier: any = value;

**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**

interface Calc {

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

}

let Calculator: Calc = {

    subtract(first: number, second: number) {

        return first - second;

    }

}

### **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);

    }

}

### **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;

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**

//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 + b;

};

## 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() {}

# TypeScript - Functions

Functions are the primary blocks of any program. In JavaScript, functions are the most important part since the JavaScript is a functional programming language. With functions, you can implement/mimic the concepts of object-oriented programming like classes, objects, polymorphism, and, abstraction.

Functions ensure that the program is maintainable and reusable, and organized into readable blocks. While TypeScript provides the concept of classes and modules, functions still are an integral part of the language.

In TypeScript, functions can be of two types: named and anonymous.

## Named Functions

A named function is one where you declare and call a function by its given name.

Example: Named Function

 Copy

function display() {

console.log("Hello TypeScript!");

}

display(); //Output: Hello TypeScript

Functions can also include parameter types and return type.

Example: Function with Parameter and Return Types

 Copy

function Sum(x: number, y: number) : number {

return x + y;

}

Sum(2,3); // returns 5

## Anonymous Function

An anonymous function is one which is defined as an expression. This expression is stored in a variable. So, the function itself does not have a name. These functions are invoked using the variable name that the function is stored in.

Example: Anonymous Function

 Copy

let greeting = function() {

console.log("Hello TypeScript!");

};

greeting(); //Output: Hello TypeScript!

An anonymous function can also include parameter types and return type.

Example: Function with Paramter and Return Types

 Copy

let Sum = function(x: number, y: number) : number

{

return x + y;

}

Sum(2,3); // returns 5

## Function Parameters

Parameters are values or arguments passed to a function. In TypeScript, the compiler expects a function to receive the exact number and type of arguments as defined in the function signature. If the function expects three parameters, the compiler checks that the user has passed values for all three parameters i.e. it checks for exact matches.

Example: Function Parameters

 Copy

function Greet(greeting: string, name: string ) : string {

return greeting + ' ' + name + '!';

}

Greet('Hello','Steve');//OK, returns "Hello Steve!"

Greet('Hi'); // Compiler Error: Expected 2 arguments, but got 1.

Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.

This is unlike JavaScript, where it is acceptable to pass less arguments than what the function expects. The parameters that don't receive a value from the user are considered as undefined.

### **Optional Parameters**

TypeScript has an optional parameter functionality. The parameters that may or may not receive a value can be appended with a '?' to mark them as optional.

 Note:

All optional parameters must follow required parameters and should be at the end.

Example: Optional Parameter

 Copy

function Greet(greeting: string, name?: string ) : string {

return greeting + ' ' + name + '!';

}

Greet('Hello','Steve');//OK, returns "Hello Steve!"

Greet('Hi'); // OK, returns "Hi undefined!".

Greet('Hi','Bill','Gates'); //Compiler Error: Expected 2 arguments, but got 3.

In the above example, the second parameter name is marked as optional with a question mark appended at the end. Hence, the function Greet() accepts either 1 or 2 parameters and returns a greeting string. If we do not specify the second parameter then its value will be undefined.

### **Default Parameters**

TypeScript provides the option to add default values to parameters. So, if the user does not provide a value to an argument, TypeScript will initialize the parameter with the default value. Default parameters have the same behaviour as optional parameters. If a value is not passed for the default parameter in a function call, the default parameter must follow the required parameters in the function signature. Hence, default parameters can be omitted while calling a function. However, if a function signature has a default parameter before a required parameter, the function can still be called, provided the default parameter is passed a value of undefined.

Example: Default Parameter

 Copy

function Greet(name: string, greeting: string = "Hello") : string {

return greeting + ' ' + name + '!';

}

Greet('Steve');//OK, returns "Hello Steve!"

Greet('Steve', 'Hi'); // OK, returns "Hi Steve!".

Greet('Bill'); //OK, returns "Hello Bill!"

When the default parameters precede required parameters in a function, they can be called by passing undefined.

Example: Function Call

 Copy

function Greet(greeting: string = "Hello", name: string) : string {

return greeting + ' ' + name + '!';

}

Greet(undefined, 'Steve');//returns "Hello Steve!"

Greet("Hi", 'Steve'); //returns "Hi Steve!".

Greet(undefined, 'Bill'); //returns "Hello Bill!"

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:

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.

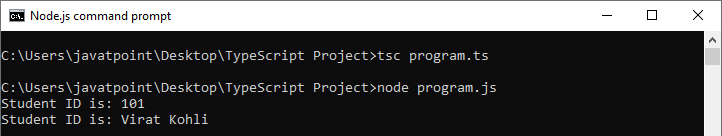
**Advantage of Method**

* 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.**

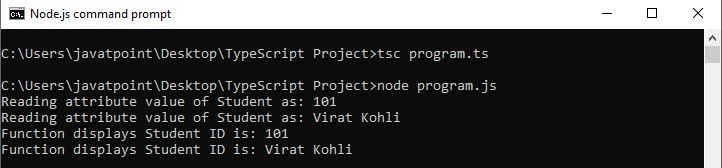
**Example**

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()

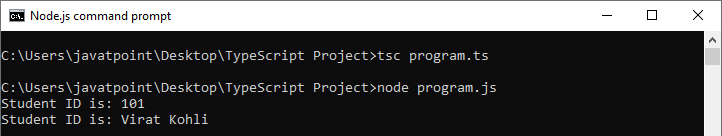
**Output:**



**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);

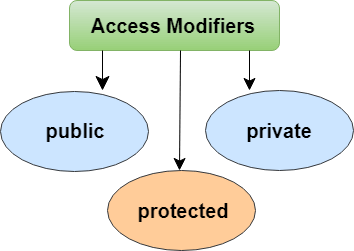
**Output:**



## 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, [click here](https://www.javatpoint.com/typescript-access-modifiers).