Syntax defines a set of rules for writing programs. Every language specification defines its own syntax. A TypeScript program is composed of −

* Modules
* Functions
* Variables
* Statements and Expressions
* Comments

Your First TypeScript Code

Let us start with the traditional “Hello World” example −

var message:string = "Hello World"

console.log(message)

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var message = "Hello World";

console.log(message);

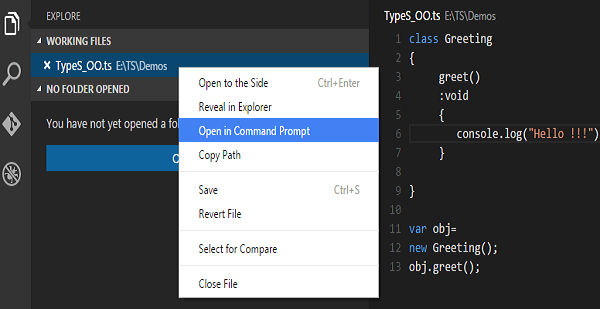
* Line 1 declares a variable by the name message. Variables are a mechanism to store values in a program.
* Line 2 prints the variable’s value to the prompt. Here, console refers to the terminal window. The function *log ()* is used to display text on the screen.

Compile and Execute a TypeScript Program

Let us see how to compile and execute a TypeScript program using Visual Studio Code. Follow the steps given below −

**Step 1** − Save the file with .ts extension. We shall save the file as Test.ts. The code editor marks errors in the code, if any, while you save it.

**Step 2** − Right-click the TypeScript file under the Working Files option in VS Code’s Explore Pane. Select Open in Command Prompt option.



**Step 3** − To compile the file use the following command on the terminal window.

tsc Test.ts

**Step 4** − The file is compiled to Test.js. To run the program written, type the following in the terminal.

node Test.js

Compiler Flags

Compiler flags enable you to change the behavior of the compiler during compilation. Each compiler flag exposes a setting that allows you to change how the compiler behaves.

The following table lists some common flags associated with the TSC compiler. A typical command-line usage uses some or all switches.

|  |  |
| --- | --- |
| **S.No.** | **Compiler flag & Description** |
| 1. | **--help**  Displays the help manual |
| 2. | **--module**  Load external modules |
| 3. | **--target**  Set the target ECMA version |
| 4. | **--declaration**  Generates an additional .d.ts file |
| 5. | **--removeComments**  Removes all comments from the output file |
| 6. | **--out**  Compile multiple files into a single output file |
| 7. | **--sourcemap**  Generate a sourcemap (.map) files |
| 8. | **--module noImplicitAny**  Disallows the compiler from inferring the any type |
| 9. | **--watch**  Watch for file changes and recompile them on the fly |

**Note** − Multiple files can be compiled at once.

tsc file1.ts, file2.ts, file3.ts

Identifiers in TypeScript

Identifiers are names given to elements in a program like variables, functions etc. The rules for identifiers are −

* Identifiers can include both, characters and digits. However, the identifier cannot begin with a digit.
* Identifiers cannot include special symbols except for underscore (\_) or a dollar sign ($).
* Identifiers cannot be keywords.
* They must be unique.
* Identifiers are case-sensitive.
* Identifiers cannot contain spaces.

The following tables lists a few examples of valid and invalid identifiers −

|  |  |
| --- | --- |
| **Valid identifiers** | **Invalid identifiers** |
| firstName | Var |
| first\_name | first name |
| num1 | first-name |
| $result | 1number |

TypeScript ─ Keywords

Keywords have a special meaning in the context of a language. The following table lists some keywords in TypeScript.

|  |  |  |  |
| --- | --- | --- | --- |
| break | as | any | switch |
| case | if | throw | else |
| var | number | string | get |
| module | type | instanceof | typeof |
| public | private | enum | export |
| finally | for | while | void |
| null | super | this | new |
| in | return | true | false |
| any | extends | static | let |
| package | implements | interface | function |
| new | try | yield | const |
| continue | do | catch |  |

Whitespace and Line Breaks

TypeScript ignores spaces, tabs, and newlines that appear in programs. You can use spaces, tabs, and newlines freely in your program and you are free to format and indent your programs in a neat and consistent way that makes the code easy to read and understand.

TypeScript is Case-sensitive

TypeScript is case-sensitive. This means that TypeScript differentiates between uppercase and lowercase characters.

Semicolons are optional

Each line of instruction is called a **statement**. Semicolons are optional in TypeScript.

**Example**

console.log("hello world")

console.log("We are learning TypeScript")

A single line can contain multiple statements. However, these statements must be separated by a semicolon.

Comments in TypeScript

Comments are a way to improve the readability of a program. Comments can be used to include additional information about a program like author of the code, hints about a function/ construct etc. Comments are ignored by the compiler.

TypeScript supports the following types of comments −

* **Single-line comments ( // )** − Any text between a // and the end of a line is treated as a comment
* **Multi-line comments (/\* \*/)** − These comments may span multiple lines.

**Example**

//this is single line comment

/\* This is a

Multi-line comment

\*/

TypeScript and Object Orientation

TypeScript is Object-Oriented JavaScript. Object Orientation is a software development paradigm that follows real-world modelling. Object Orientation considers a program as a collection of objects that communicate with each other via mechanism called methods. TypeScript supports these object oriented components too.

* **Object** − An object is a real time representation of any entity. According to Grady Brooch, every object must have three features −
  + **State** − described by the attributes of an object
  + **Behavior** − describes how the object will act
  + **Identity** − a unique value that distinguishes an object from a set of similar such objects.
* **Class** − A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object.
* **Method** − Methods facilitate communication between objects.

**Example: TypeScript and Object Orientation**

class Greeting {

greet():void {

console.log("Hello World!!!")

}

}

var obj = new Greeting();

obj.greet();

The above example defines a class *Greeting*. The class has a method *greet ()*. The method prints the string “Hello World” on the terminal. The **new** keyword creates an object of the class (obj). The object invokes the method *greet ()*.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var Greeting = (function () {

function Greeting() {

}

Greeting.prototype.greet = function () {

console.log("Hello World!!!");

};

return Greeting;

}());

var obj = new Greeting();

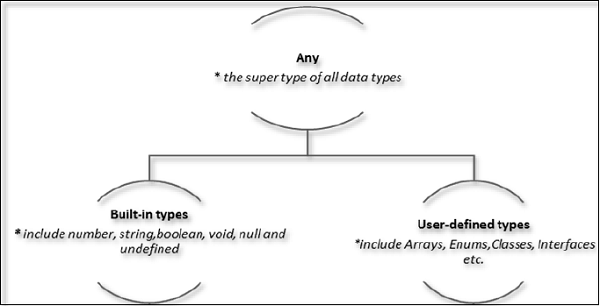
obj.greet()

The output of the above program is given below −

Hello World!!!

The Type System represents the different types of values supported by the language. The Type System checks the validity of the supplied values, before they are stored or manipulated by the program. This ensures that the code behaves as expected. The Type System further allows for richer code hinting and automated documentation too.

TypeScript provides data types as a part of its optional Type System. The data type classification is as given below −



The Any type

The **any** data type is the super type of all types in TypeScript. It denotes a dynamic type. Using the **any** type is equivalent to opting out of type checking for a variable.

Built-in types

The following table illustrates all the built-in types in TypeScript −

|  |  |  |
| --- | --- | --- |
| **Data type** | **Keyword** | **Description** |
| Number | number | Double precision 64-bit floating point values. It can be used to represent both, integers and fractions. |
| String | string | Represents a sequence of Unicode characters |
| Boolean | boolean | Represents logical values, true and false |
| Void | void | Used on function return types to represent non-returning functions |
| Null | null | Represents an intentional absence of an object value. |
| Undefined | undefined | Denotes value given to all uninitialized variables |

**Note** − There is no integer type in TypeScript and JavaScript.

Null and undefined ─ Are they the same?

The **null** and the **undefined** datatypes are often a source of confusion. The null and undefined cannot be used to reference the data type of a variable. They can only be assigned as values to a variable.

However, *null and undefined are not the same*. A variable initialized with undefined means that the variable has no value or object assigned to it while null means that the variable has been set to an object whose value is undefined.

User-defined Types

User-defined types include Enumerations (enums), classes, interfaces, arrays, and tuple. These are discussed in detail in the later chapters.

# TypeScript - Variables

Advertisements

[Previous Page](https://www.tutorialspoint.com/typescript/typescript_types.htm)

[Next Page](https://www.tutorialspoint.com/typescript/typescript_operators.htm)

A variable, by definition, is “a named space in the memory” that stores values. In other words, it acts as a container for values in a program. TypeScript variables must follow the JavaScript naming rules −

* Variable names can contain alphabets and numeric digits.
* They cannot contain spaces and special characters, except the underscore (\_) and the dollar ($) sign.
* Variable names cannot begin with a digit.

A variable must be declared before it is used. Use the **var** keyword to declare variables.

## Variable Declaration in TypeScript

The type syntax for declaring a variable in TypeScript is to include a colon (:) after the variable name, followed by its type. Just as in JavaScript, we use the **var** keyword to declare a variable.

When you declare a variable, you have four options −

* Declare its type and value in one statement.

Declare Type

* Declare its type but no value. In this case, the variable will be set to undefined.

Undefined

* Declare its value but no type. The variable type will be set to any.

Any

* Declare neither value not type. In this case, the data type of the variable will be any and will be initialized to undefined.

Any and Undefined

The following table illustrates the valid syntax for variable declaration as discussed above −

|  |  |
| --- | --- |
| **S.No.** | **Variable Declaration Syntax & Description** |
| 1. | **var name:string = ”mary”**  The variable stores a value of type string |
| 2. | **var name:string;**  The variable is a string variable. The variable’s value is set to undefined by default |
| 3. | **var name = ”mary”**  The variable’s type is inferred from the data type of the value. Here, the variable is of the type string |
| 4. | **var name;**  The variable’s data type is any. Its value is set to undefined by default. |

### Example: Variables in TypeScript

var name:string = "John";

var score1:number = 50;

var score2:number = 42.50

var sum = score1 + score2

console.log("name"+name)

console.log("first score: "+score1)

console.log("second score: "+score2)

console.log("sum of the scores: "+sum)

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var name = "John";

var score1 = 50;

var score2 = 42.50;

var sum = score1 + score2;

console.log("name" + name);

console.log("first score: " + score1);

console.log("second score : " + score2);

console.log("sum of the scores: " + sum);

The output of the above program is given below −

name:John

first score:50

second score:42.50

sum of the scores:92.50

The TypeScript compiler will generate errors, if we attempt to assign a value to a variable that is not of the same type. Hence, TypeScript follows Strong Typing. The Strong typing syntax ensures that the types specified on either side of the assignment operator (=) are the same. This is why the following code will result in a compilation error −

var num:number = "hello" // will result in a compilation error

## Type Assertion in TypeScript

TypeScript allows changing a variable from one type to another. TypeScript refers to this process as *Type Assertion*. The syntax is to put the target type between < > symbols and place it in front of the variable or expression. The following example explains this concept −

### Example

var str = '1'

var str2:number = <number> <any> str //str is now of type number

console.log(str2)

If you hover the mouse pointer over the type assertion statement in Visual Studio Code, it displays the change in the variable’s data type. Basically it allows the assertion from type S to T succeed if either S is a subtype of T or T is a subtype of S.

The reason why it's not called "type casting" is that casting generally implies some sort of runtime support while, “type assertions” are purely a compile time construct and a way for you to provide hints to the compiler on how you want your code to be analyzed.

On compiling, it will generate following JavaScript code.

//Generated by typescript 1.8.10

var str = '1';

var str2 = str; //str is now of type number

console.log(str2);

It will produce the following output −

1

## Inferred Typing in TypeScript

Given the fact that, Typescript is strongly typed, this feature is optional. TypeScript also encourages dynamic typing of variables. This means that, TypeScript encourages declaring a variable without a type. In such cases, the compiler will determine the type of the variable on the basis of the value assigned to it. TypeScript will find the first usage of the variable within the code, determine the type to which it has been initially set and then assume the same type for this variable in the rest of your code block.

The same is explained in the following code snippet −

### Example: Inferred Typing

var num = 2; // data type inferred as number

console.log("value of num "+num);

num = "12";

console.log(num);

In the above code snippet −

* The code declares a variable and sets its value to 2. Note that the variable declaration doesn’t specify the data type. Hence, the program uses inferred typing to determine the data type of the variable, i.e., it assigns the type of the first value that the variable is set to. In this case, **num** is set to the type number.
* When the code tries to set the variable’s value to string. The compiler throws an error as the variable’s type is already set to number.

It will produce the following output −

error TS2011: Cannot convert 'string' to 'number'.

## TypeScript Variable Scope

The scope of a variable specifies where the variable is defined. The availability of a variable within a program is determined by its scope. TypeScript variables can be of the following scopes −

* **Global Scope** − Global variables are declared outside the programming constructs. These variables can be accessed from anywhere within your code.
* **Class Scope** − These variables are also called **fields**. Fields or class variables are declared within the class but outside the methods. These variables can be accessed using the object of the class. Fields can also be static. Static fields can be accessed using the class name.
* **Local Scope** − Local variables, as the name suggests, are declared within the constructs like methods, loops etc. Local variables are accessible only within the construct where they are declared.

The following example illustrates variable scopes in TypeScript.

### Example: Variable Scope

var global\_num = 12 //global variable

class Numbers {

num\_val = 13; //class variable

static sval = 10; //static field

storeNum():void {

var local\_num = 14; //local variable

}

}

console.log("Global num: "+global\_num)

console.log(Numbers.sval) //static variable

var obj = new Numbers();

console.log("Global num: "+obj.num\_val)

On transpiling, the following JavaScript code is generated −

var global\_num = 12; //global variable

var Numbers = (function () {

function Numbers() {

this.num\_val = 13; //class variable

}

Numbers.prototype.storeNum = function () {

var local\_num = 14; //local variable

};

Numbers.sval = 10; //static field

return Numbers;

}());

console.log("Global num: " + global\_num);

console.log(Numbers.sval); //static variable

var obj = new Numbers();

console.log("Global num: " + obj.num\_val);

It will produce the following output −

Global num: 12

10

Global num: 13

If you try accessing the local variable outside the method, it results in a compilation error.

error TS2095: Could not find symbol 'local\_num'.