# YASH TRAINING & DEVELOPMENT

Topic : Training for TypeScript

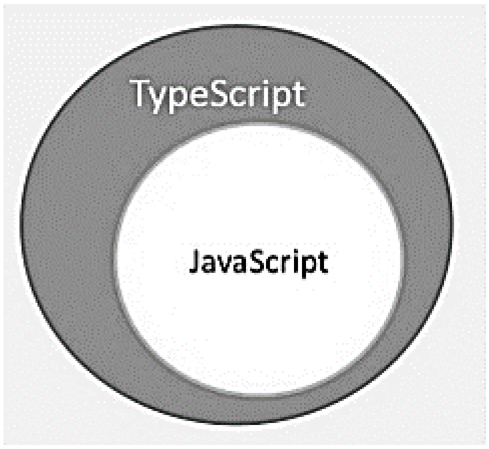
# INDEX

# TypeScript - Overview

## What is TypeScript?

By definition, “TypeScript is JavaScript for application-scale development.”

TypeScript is a strongly typed, object oriented, compiled language. It was designed by **Anders Hejlsberg** (designer of C#) at Microsoft. TypeScript is both a language and a set of tools. TypeScript is a typed superset of JavaScript compiled to JavaScript. In other words, TypeScript is JavaScript plus some additional features.



## Features of TypeScript

**TypeScript is just JavaScript**. TypeScript starts with JavaScript and ends with JavaScript. Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use TypeScript. All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.

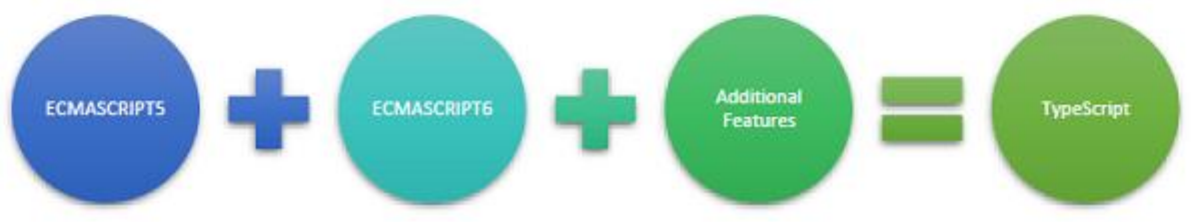
**TypeScript supports other JS libraries**. Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

**JavaScript is TypeScript**. This means that any valid **.js** file can be renamed to **.ts** and compiled with other TypeScript files.

**TypeScript is portable**. TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, TypeScript doesn’t need a dedicated VM or a specific runtime environment to execute.

## TypeScript and ECMAScript

The ECMAScript specification is a standardized specification of a scripting language. There are six editions of ECMA-262 published. Version 6 of the standard is codenamed "Harmony". TypeScript is aligned with the ECMAScript6 specification.



TypeScript adopts its basic language features from the ECMAScript5 specification, i.e., the official specification for JavaScript. TypeScript language features like Modules and class-based orientation are in line with the EcmaScript 6 specification. Additionally, TypeScript also embraces features like generics and type annotations that aren't a part of the EcmaScript6 specification.

## Why Use TypeScript?

TypeScript is superior to its other counterparts like CoffeeScript and Dart programming languages in a way that TypeScript is extended JavaScript. In contrast, languages like Dart, CoffeeScript are new languages in themselves and require language-specific execution environment.

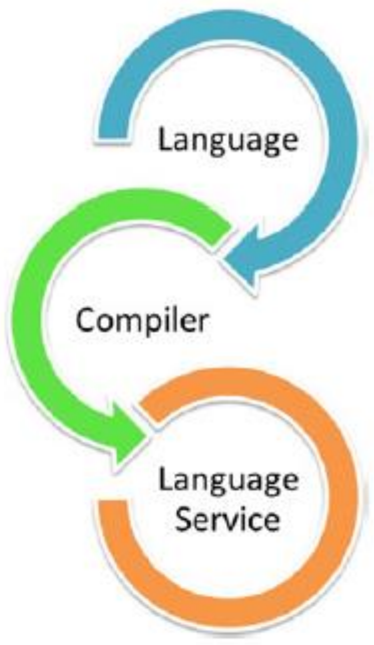
The benefits of TypeScript include −

- **Compilation** − JavaScript is an interpreted language. Hence, it needs to be run to test that it is valid. It means you write all the codes just to find no output, in case there is an error. Hence, you have to spend hours trying to find bugs in the code. The TypeScript transpiler provides the error-checking feature. TypeScript will compile the code and generate compilation errors, if it finds some sort of syntax errors. This helps to highlight errors before the script is run.
- **Strong Static Typing** − JavaScript is not strongly typed. TypeScript comes with an optional static typing and type inference system through the TLS (TypeScript Language Service). The type of a variable, declared with no type, may be inferred by the TLS based on its value.
- TypeScript **supports type definitions** for existing JavaScript libraries. TypeScript Definition file (with **.ts** extension) provides definition for external JavaScript libraries. Hence, TypeScript code can contain these libraries.
- TypeScript **supports Object Oriented Programming** concepts like classes, interfaces, inheritance, etc.

## Components of TypeScript

At its heart, TypeScript has the following three components −

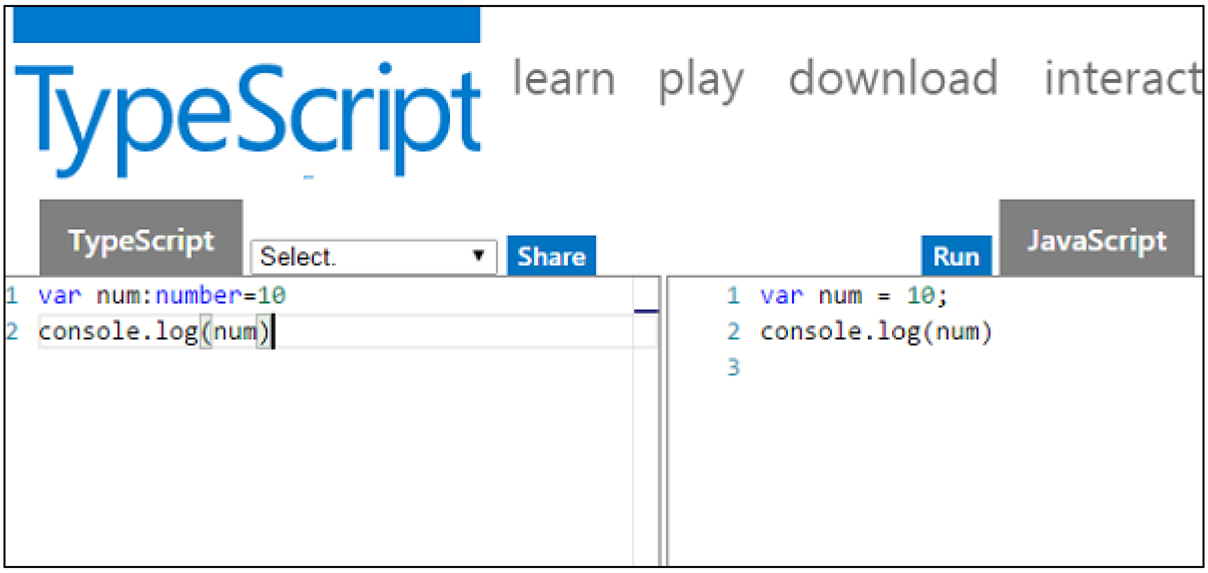
- **Language** − It comprises of the syntax, keywords, and type annotations.
- **The TypeScript Compiler** − The TypeScript compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.
- **The TypeScript Language Service** − The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications. The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.



# TypeScript - Environment Setup

## TypeScript – Try it Option Online

You may test your scripts online by using The TypeScript at www.typescriptlang.org/Playground. The online editor shows the corresponding JavaScript emitted by the compiler.



```
var num:number=10
console.log(num)
```

```
var num = 10;
console.log(num)
```

You may try the following example using **Playground**.

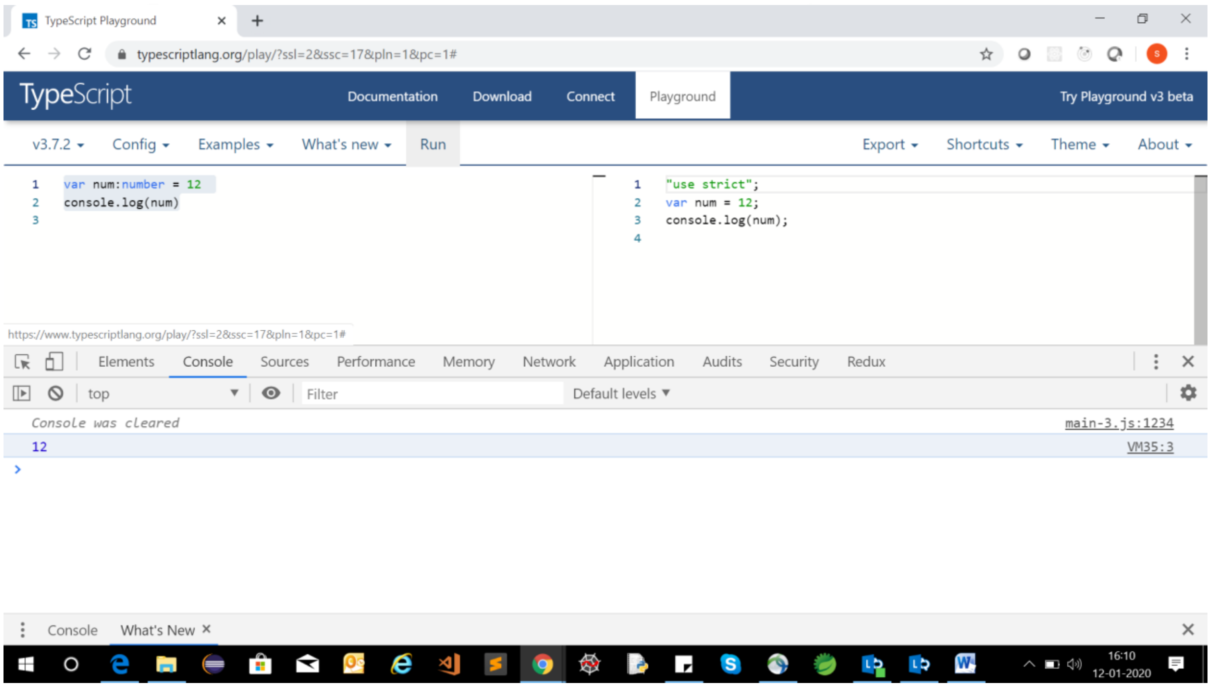
```
var num:number = 12
console.log(num)
```

On compiling , it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var num = 12;
console.log(num);
```

The output of the above program is given below −

```
12
```



```
var num:number = 12
console.log(num)
```

```
"use strict";
var num = 12;
console.log(num);
```

```
Console was cleared                                   main-3.js:1234
12                                                    VM35:3
```

## Local Environment Setup

Typescript is an Open Source technology. It can run on any browser, any host, and any OS. You will need the following tools to write and test a Typescript program −

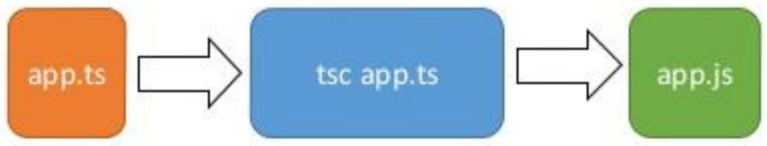
### A Text Editor

The text editor helps you to write your source code. Examples of a few editors include Windows Notepad, Notepad++, Emacs, vim or vi, etc. Editors used may vary with Operating Systems.

The source files are typically named with the extension **.ts**

### The TypeScript Compiler

The TypeScript compiler is itself a **.ts** file compiled down to JavaScript (.js) file. The TSC (TypeScript Compiler) is a source-to-source compiler (transcompiler / transpiler).



The TSC generates a JavaScript version of the **.ts** file passed to it. In other words, the TSC produces an equivalent JavaScript source code from the Typescript file given as an input to it. This process is termed as transpilation.

However, the compiler rejects any raw JavaScript file passed to it. The compiler deals with only **.ts** or **.d.ts** files.

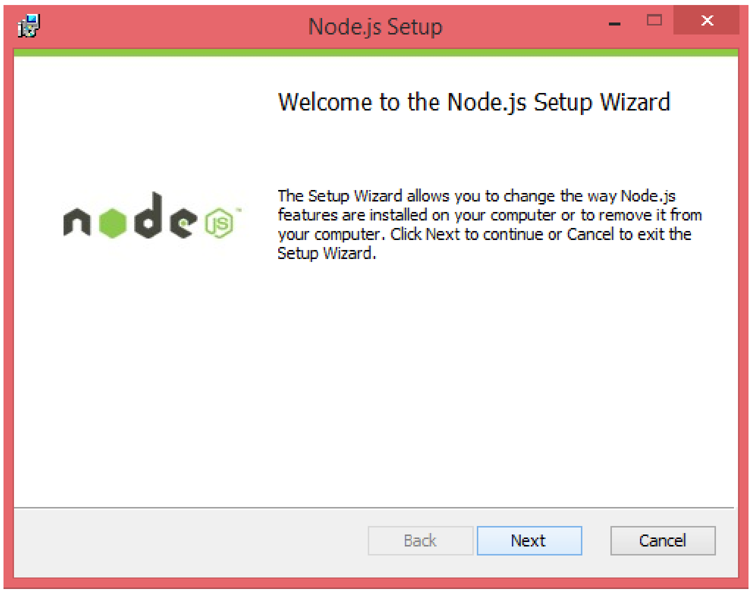
## Installing Node.js

Node.js is an open source, cross-platform runtime environment for server-side JavaScript. Node.js is required to run JavaScript without a browser support. It uses Google V8 JavaScript engine to execute code. You may download Node.js source code or a pre-built installer for your platform. Node is available here − *https://nodejs.org/en/download*

### Installation on Windows

Follow the steps given below to install Node.js in Windows environment.

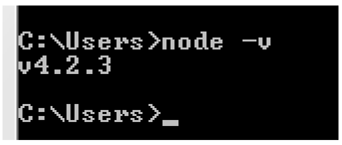
**Step 1** − Download and run the .msi installer for Node.



Welcome to the Node.js Setup Wizard

The Setup Wizard allows you to change the way Node.js features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

**Step 2** − To verify if the installation was successful, enter the command ***node –v*** in the terminal window.

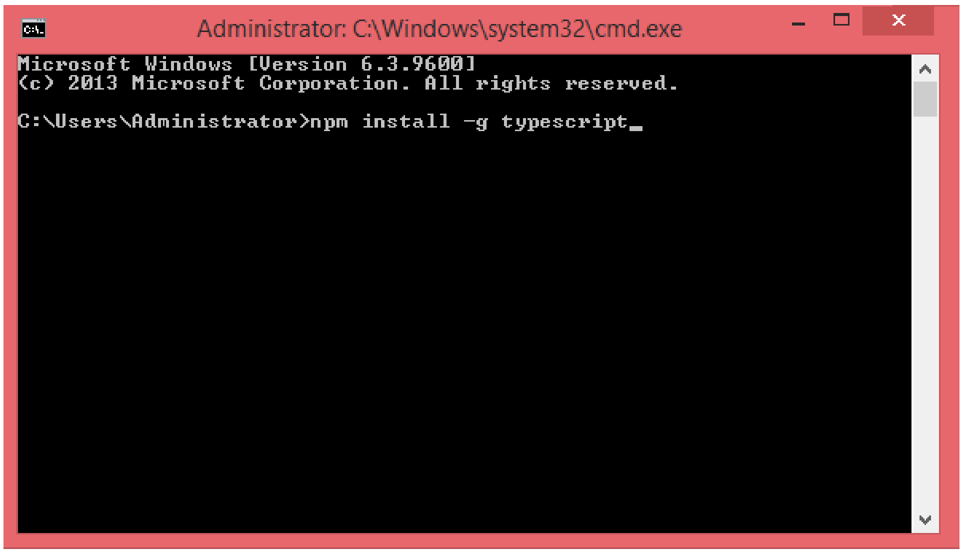


```
C:\Users>node -v
v4.2.3

C:\Users>_
```

**Step 3** − Type the following command in the terminal window to install TypeScript.

```
npm install -g typescript
```


Administrator: C:\Windows\system32\cmd.exe

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Administrator>npm install -g typescript_
```

## IDE Support

Typescript can be built on a plethora of development environments like Visual Studio, Sublime Text 2, WebStorm/PHPStorm, Eclipse, Brackets, etc. Visual Studio Code and Brackets IDEs are discussed here. The development environment used here is Visual Studio Code (Windows platform).

## Visual Studio Code

This is an open source IDE from Visual Studio. It is available for Mac OS X, Linux and Windows platforms. VScode is available at − https://code.visualstudio.com/

### Installation on Windows

Download Visual Studio Code for Windows.

# TypeScript - Basic Syntax

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);
```

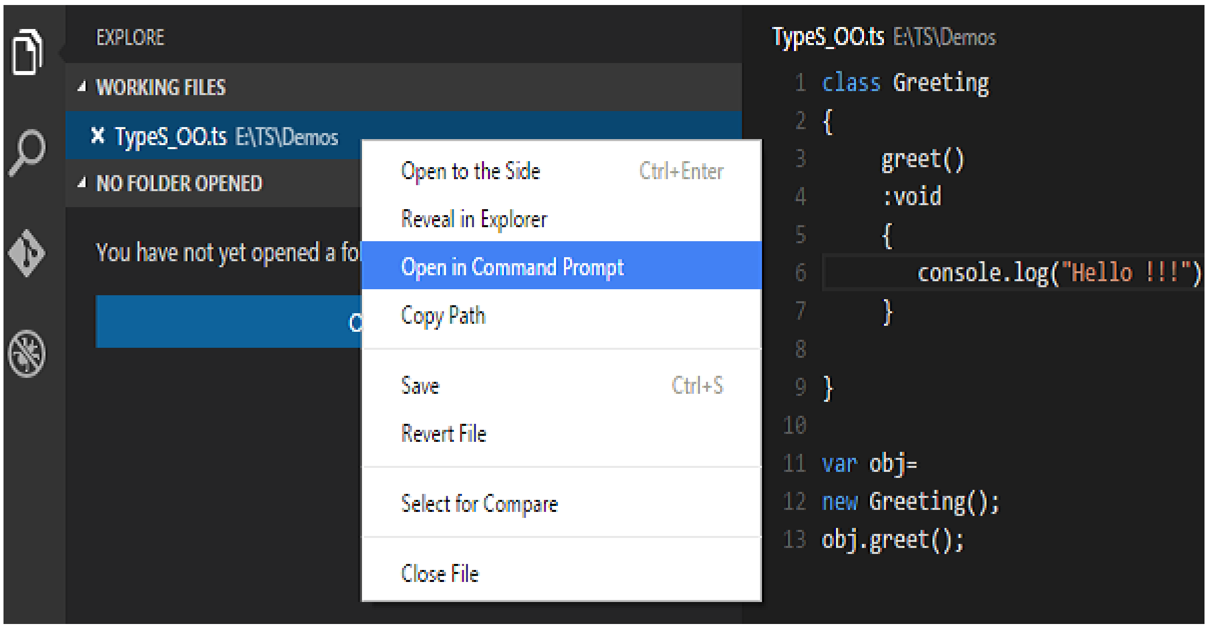
- Line 1 declares a variable by the name message. Variables are a mechanism to store values in a program.
- Line 2 prints the variable’s value to the prompt. Here, console refers to the terminal window. The function *log ()* is used to display text on the screen.

## 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.



```
class Greeting
{
     greet()
     :void
     {
          console.log("Hello !!!")
     }

}

var obj=
new Greeting();
obj.greet();
```

**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
```

## 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**

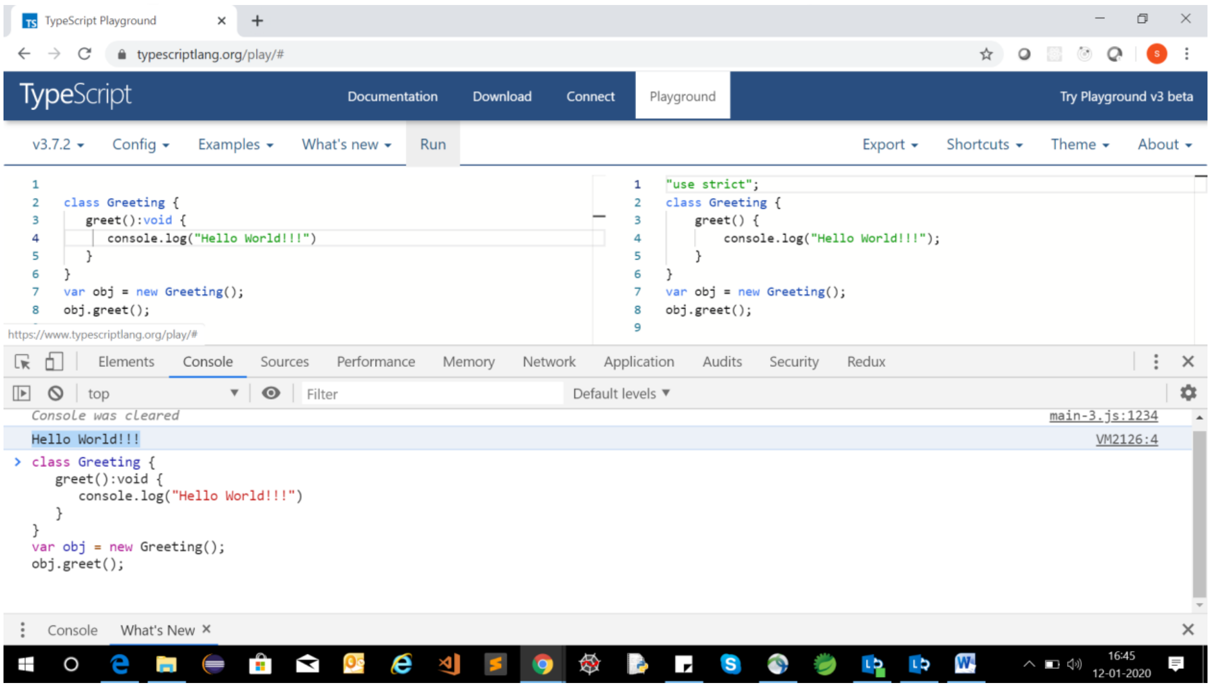
```
class Greeting {
   greet():void {
      console.log("Hello World!!!")
   }
}
var obj = new Greeting();
obj.greet();
```

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 ()*.

The output of the above program is given below −

```
Hello World!!!
```

Try it online https://www.typescriptlang.org/play/?ssl=2&ssc=22&pln=1&pc=1#



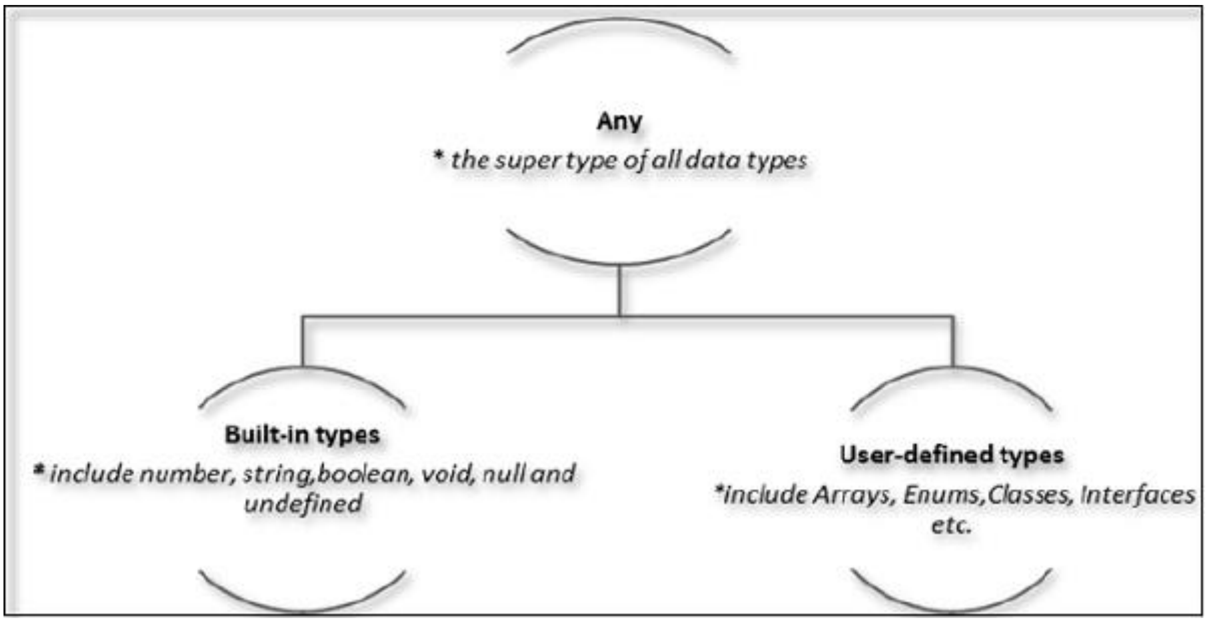
```
class Greeting {
    greet():void {
      console.log("Hello World!!!")
    }
}
var obj = new Greeting();
obj.greet();
```

```
"use strict";
class Greeting {
    greet() {
        console.log("Hello World!!!");
    }
}
var obj = new Greeting();
obj.greet();
```

```
Console was cleared                                   main-3.js:1234
Hello World!!!                                          VM2126:4
> class Greeting {
      greet():void {
        console.log("Hello World!!!")
      }
  }
  var obj = new Greeting();
  obj.greet();
```

# TypeScript - Types

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


Any
* the super type of all data types
Built-in types
* include number, string,boolean, void, null and undefined
User-defined types
*include Arrays, Enums,Classes, Interfaces etc.

## 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.

## 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.

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

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

```
var [identifier] : [type-annotation] ;
```

- Declare its value but no type. The variable type will be set to the data type of the assigned value.

| | | | | |
|---|---|---|---|---|
| var | [identifier] | = | value | ; |

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

| | | |
|---|---|---|
| var | [identifier] | ; |

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

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

## TypeScript - Variable

TypeScript follows the same rules as JavaScript for variable declarations. Variables can be declared using: `var`, `let`, and `const`.

### var Declaration

Variables in TypeScript can be declared using var keyword, same as in JavaScript. The scoping rules remains the same as in JavaScript.

### let Declaration

To solve problems with **var** declarations, ES6 introduced two new types of variable declarations in JavaScript, using the keywords **let** and **const**. TypeScript, being a superset of JavaScript, also supports these new types of variable declarations.

### Example: Variable Declaration using let

```
let employeeName = "John";
// or
let employeeName:string = "John";
```

The let declarations follow the same syntax as var declarations. Unlike variables declared with `var`, variables declared with `let` have a block-scope. This means that the scope of let variables is limited to their containing block, e.g. function, if else block or loop block. Consider the following example.

### Example: let Variables Scope

```
let num1:number = 1;

function letDeclaration() {
    let num2:number = 2;

    if (num2 > num1) {
        let num3: number = 3;
        num3++;
    }

    while(num1 < num2) {
        let num4: number = 4;
        num1++;
    }
    console.log(num1); //OK
    console.log(num2); //OK
    console.log(num3); //Compiler Error: Cannot find name 'num3'
    console.log(num4); //Compiler Error: Cannot find name 'num4'
}

letDeclaration();
```

In the above example, all the variables are declared using let. `num3` is declared in the if block so its scope is limited to the if block and cannot be accessed out of the if block. In the same way, `num4` is declared in the `while` block so it cannot be accessed out of while block. Thus, when accessing `num3` and `num4` else where will give a compiler error.

The same example with the var declaration is compiled without an error.

### Example: var Variables Scope

```
var num1:number = 1;

function varDeclaration() {
    var num2:number = 2;

    if (num2 > num1) {
        var num3: number = 3;
        num3++;
    }

    while(num1 < num2) {
        var num4: number = 4;
        num1++;
    }

    console.log(num1); //2
    console.log(num2); //2
    console.log(num3); //4
    console.log(num4); //4
}

varDeclaration();
```

## Advantages of using let over var

1) Block-scoped let variables cannot be read or written to before they are declared.

## Example: let vs var

```
console.log(num1); // Compiler Error: error TS2448: Block-scoped variable 'num'
used before its declaration
let num1:number = 10 ;

console.log(num2); // OK, Output: undefined
var num2:number = 10 ;
```

In the above example, the TypeScript compiler will give an error if we use variables before declaring them using let, whereas it won't give an error when using variables before declaring them using var.

2) Let variables cannot be re-declared

The TypeScript compiler will give an error when variables with the same name (case sensitive) are declared multiple times in the same block using let.

## Example: Multiple Variables with the Same Name

```
let num:number = 1; // OK
let Num:number = 2;// OK
let NUM:number = 3;// OK
let NuM:number = 4;// OK

let num:number = 5;// Compiler Error: Cannot redeclared block-scoped variable
'num'
let Num:number = 6;// Compiler Error: Cannot redeclared block-scoped variable
'Num'
let NUM:number = 7;// Compiler Error: Cannot redeclared block-scoped variable
'NUM'
let NuM:number = 8;// Compiler Error: Cannot redeclared block-scoped variable
'NuM'
```

In the above example, the TypeScript compiler treats variable names as case sensitive. `num` is different than `Num`, so it won't give any error. However, it will give an error for the variables with the same name and case.

Variables with the same name and case can be declared in different blocks, as shown below.

## Example: Same Variable Name in Different Blocks

```
let num:number = 1;

function demo() {
    let num:number = 2;

    if(true) {
        let num:number = 3;
        console.log(num); //Output: 3
    }

    console.log(num);//Output: 2
}
console.log(num); //Output: 1
demo();
```

Similarly, the compiler will give an error if we declare a variable that was already passed in as an argument to the function, as shown below.

```
function letDemo(a: number ) {
    let a:number = 10 ; //Compiler Error: TS2300: Duplicate identifier 'a'
    let b:number = 20 ;

    return a + b ;
}
```

Thus, variables declared using `let` minimize the possibilities of runtime errors, as the compiler give compile-time errors. This increases the code readability and maintainability.

## Const Declaration

Variables can be declared using const similar to var or let declarations. The const makes a variable a constant where its value cannot be changed. Const variables have the same scoping rules as let variables.

## Example: Const Variable

```
const num:number = 100;
num = 200; //Compiler Error: Cannot assign to 'num' because it is a constant or
read-only property
```

Const variables must be declared and initialized in a single statement. Separate declaration and initialization is not supported.

```
const num:number; //Compiler Error: const declaration must be initialized
num = 100;
```

Const variables allow an object sub-properties to be changed but not the object structure.

## Example: const Object

```
const playerCodes = {
    player1 : 9,
    player2 : 10,
    player3 : 13,
    player4 : 20
};
playerCodes.player2 = 11; // OK

playerCodes = {     //Compiler Error: Cannot assign to playerCodes because it is a
constant or read-only
    player1 : 50,   // Modified value
    player2 : 10,
    player3 : 13,
    player4 : 20
};
```

Even if you try to change the object structure, the compiler will point this error out.

```
const playerCodes = {
    player1: 9,
    player2: 10,
    player3: 13,
    player4: 20
};

playerCodes = { //Compiler Error: Cannot assign to playerCodes because it is a
constant or read-only
    player1: 9,
    player2: 10,
    player3: 13,
    player4: 20,
    player5: 22
};
```

# TypeScript - Functions

## Optional Parameters

Optional parameters can be used when arguments need not be compulsorily passed for a function's execution. A parameter can be marked optional by appending a question mark to its name. The optional parameter should be set as the last argument in a function. The syntax to declare a function with optional parameter is as given below −

```
function function_name (param1[:type], param2[:type],
param3[:type])
```

### Example: Optional Parameters

```
function disp_details(id:number,name:string,mail_id?:string) {
   console.log("ID:", id);
   console.log("Name",name);

   if(mail_id!=undefined)
   console.log("Email Id",mail_id);
}
disp_details(123,"John");
disp_details(111,"mary","mary@xyz.com");
```

- The above example declares a parameterized function. Here, the third parameter, i.e., mail_id is an optional parameter.
- If an optional parameter is not passed a value during the function call, the parameter's value is set to undefined.
- The function prints the value of mail_id only if the argument is passed a value.

The above code will produce the following output −

```
ID:123
Name John
ID: 111
Name  mary
Email Id mary@xyz.com
```

## Rest Parameters

Rest parameters are similar to variable arguments in Java. Rest parameters don't restrict the number of values that you can pass to a function. However, the values passed must all be of the same type. In other words, rest parameters act as placeholders for multiple arguments of the same type.

To declare a rest parameter, the parameter name is prefixed with three periods. Any nonrest parameter should come before the rest parameter.

### Example: Rest Parameters

```
function addNumbers(...nums:number[]) {
   var i;
   var sum:number = 0;

   for(i = 0;i<nums.length;i++) {
      sum = sum + nums[i];
   }
   console.log("sum of the numbers",sum)
}
addNumbers(1,2,3)
addNumbers(10,10,10,10,10)
```

- The function addNumbers() declaration, accepts a rest parameter *nums*. The rest parameter's data type must be set to an array. Moreover, a function can have at the most one rest parameter.
- The function is invoked twice, by passing three and six values, respectively.
- The for loop iterates through the argument list, passed to the function and calculates their sum.

The output of the above code is as follows −

```
sum of numbers 6
sum of numbers 50
```

## Default Parameters

Function parameters can also be assigned values by default. However, such parameters can also be explicitly passed values.

### Syntax

```
function function_name(param1[:type],param2[:type] =
default_value) {
}
```

**Note** − A parameter cannot be declared optional and default at the same time.

### Example: Default parameters

```
function calculate_discount(price:number,rate:number = 0.50) {
   var discount = price * rate;
   console.log("Discount Amount: ",discount);
}
calculate_discount(1000)
calculate_discount(1000,0.30)
```

Its output is as follows −

```
Discount amount : 500
Discount amount : 300
```

- The example declares the function, *calculate_discount*. The function has two parameters - price and rate.
- The value of the parameter *rate* is set to *0.50* by default.
- The program invokes the function, passing to it only the value of the parameter price. Here, the value of *rate* is *0.50* (default)
- The same function is invoked, but with two arguments. The default value of *rate* is overwritten and is set to the value explicitly passed.

## Anonymous Function

Functions that are not bound to an identifier (function name) are called as **anonymous functions**. These functions are dynamically declared at runtime. Anonymous functions can accept inputs and return outputs, just as standard functions do. An anonymous function is usually not accessible after its initial creation.

Variables can be assigned an anonymous function. Such an expression is called a function expression.

### Syntax

```
var res = function( [arguments] ) { ... }
```

### Example ─ A Simple Anonymous function

```
var msg = function() {
   return "hello world";
}
console.log(msg())
```

On compiling, it will generate the same code in JavaScript.

It will produce the following output −

```
hello world
```

### Example ─ Anonymous function with parameters

```
var res = function(a:number,b:number) {
   return a*b;
};
console.log(res(12,2))
```

The anonymous function returns the product of the values passed to it.

On compiling, it will generate following JavaScript code −

```
//Generated by typescript 1.8.10
var res = function (a, b) {
   return a * b;
};
```

```
console.log(res(12, 2));
```

The output of the above code is as follows −

```
24
```

## Lambda Functions

Lambda refers to anonymous functions in programming. Lambda functions are a concise mechanism to represent anonymous functions. These functions are also called as **Arrow functions**.

### Lambda Function - Anatomy

There are 3 parts to a Lambda function −

- **Parameters** − A function may optionally have parameters
- **The fat arrow notation/lambda notation (=>)** − It is also called as the goes to operator
- **Statements** − represent the function’s instruction set

**Tip** − By convention, the use of single letter parameter is encouraged for a compact and precise function declaration.

### Lambda Expression

It is an anonymous function expression that points to a single line of code. Its syntax is as follows −

```
( [param1, parma2,…param n] )=>statement;
```

### Example: Lambda Expression

```
var foo = (x:number)=>10 + x
console.log(foo(100))      //outputs 110
```

The program declares a lambda expression function. The function returns the sum of 10 and the argument passed.

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var foo = function (x) { return 10 + x; };
console.log(foo(100));      //outputs 110
```

Here is the output of the above code −

```
110
```

### Lambda Statement

Lambda statement is an anonymous function declaration that points to a block of code. This syntax is used when the function body spans multiple lines. Its syntax is as follows −

```
( [param1, parma2,…param n] )=> {
   
   //code block
}
```

### Example: Lambda statement

```
var foo = (x:number)=> {
   x = 10 + x
   console.log(x)
}
foo(100)
```

The function’s reference is returned and stored in the variable **foo**.

On compiling, it will generate following JavaScript code −

```
//Generated by typescript 1.8.10
var foo = function (x) {
   x = 10 + x;
   console.log(x);
};
foo(100);
```

The output of the above program is as follows −

```
110
```

## Syntactic Variations

### Parameter type Inference

It is not mandatory to specify the data type of a parameter. In such a case the data type of the parameter is any. Let us take a look at the following code snippet −

```
var func = (x)=> {
   if(typeof x=="number") {
      console.log(x+" is numeric")
   } else if(typeof x=="string") {
      console.log(x+" is a string")
   }
}
func(12)
func("Tom")
```

On compiling, it will generate the following JavaScript code −

```
//Generated by typescript 1.8.10
var func = function (x) {
   if (typeof x == "number") {
      console.log(x + " is numeric");
   } else if (typeof x == "string") {
      console.log(x + " is a string");
   }
};
func(12);
func("Tom");
```

Its output is as follows −

```
12 is numeric
Tom is a string
```

### Optional parentheses for a single parameter

```
var display = x=> {
   console.log("The function got "+x)
}
display(12)
```

On compiling, it will generate following JavaScript code −

```
//Generated by typescript 1.8.10
var display = function (x) {
   console.log("The function got " + x);
};
display(12);
```

Its output is as follows −

```
The function got 12
```

### Optional braces for a single statement, Empty parentheses for no parameter

The following example shows these two Syntactic variations.

```
var disp =()=> {
   console.log("Function invoked");
}
disp();
```

Its output is as follows −

```
Function invoked
```

# TypeScript - Interfaces

An interface is a syntactical contract that an entity should conform to. In other words, an interface defines the syntax that any entity must adhere to.

Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

Let's consider an object −

```
var person = {
   FirstName:"Tom",
   LastName:"Hanks",
   sayHi: ()=>{ return "Hi"}
};
```

If we consider the signature of the object, it could be −

```
{
   FirstName:string,
   LastName:string,
   sayHi()=>string
}
```

To reuse the signature across objects we can define it as an interface.

## Declaring Interfaces

The interface keyword is used to declare an interface. Here is the syntax to declare an interface −

### Syntax

```
interface interface_name {
}
```

### Example: Interface and Objects

```
interface IPerson {
   firstName:string,
   lastName:string,
   sayHi: ()=>string
}

var customer:IPerson = {
   firstName:"Tom",
   lastName:"Hanks",
   sayHi: ():string =>{return "Hi there"}
}
```

```
console.log("Customer Object ")
console.log(customer.firstName)
console.log(customer.lastName)
console.log(customer.sayHi())

var employee:IPerson = {
   firstName:"Jim",
   lastName:"Blakes",
   sayHi: ():string =>{return "Hello!!!"}
}

console.log("Employee  Object ")
console.log(employee.firstName);
console.log(employee.lastName);
```

The example defines an interface. The customer object is of the type IPerson. Hence, it will now be binding on the object to define all properties as specified by the interface.

Another object with following signature, is still considered as IPerson because that object is treated by its size or signature.

The output of the above example code is as follows −

```
Customer object
Tom
Hanks
Hi there
Employee  object
Jim
Blakes
Hello!!!
```

# TypeScript - Classes

TypeScript is object oriented JavaScript. TypeScript supports object-oriented programming features like classes, interfaces, etc. A class in terms of OOP is a blueprint for creating objects. A class encapsulates data for the object. Typescript gives built in support for this concept called class. JavaScript ES5 or earlier didn't support classes. Typescript gets this feature from ES6.

## Creating classes

Use the class keyword to declare a class in TypeScript. The syntax for the same is given below −

### Syntax

```
class class_name {
   //class scope
}
```

The class keyword is followed by the class name. The rules for identifiers must be considered while naming a class.

A class definition can include the following −

- **Fields** − A field is any variable declared in a class. Fields represent data pertaining to objects
- **Constructors** − Responsible for allocating memory for the objects of the class
- **Functions** − Functions represent actions an object can take. They are also at times referred to as methods

These components put together are termed as the data members of the class.

Consider a class Person in typescript.

```
class Person {
}
```

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var Person = (function () {
   function Person() {
   }
   return Person;
}());
```

### Example: Declaring a class

```
class Car {
   //field
   engine:string;
   //constructor
   constructor(engine:string) {
      this.engine = engine
   }

   //function
   disp():void {
      console.log("Engine is  :   "+this.engine)
   }
}
```

The example declares a class Car. The class has a field named engine. The **var** keyword is not used while declaring a field. The example above declares a constructor for the class.

A constructor is a special function of the class that is responsible for initializing the variables of the class. TypeScript defines a constructor using the constructor keyword. A constructor is a function and hence can be parameterized.

The **this** keyword refers to the current instance of the class. Here, the parameter name and the name of the class's field are the same. Hence to avoid ambiguity, the class's field is prefixed with the **this** keyword.

*disp()* is a simple function definition. Note that the function keyword is not used here.

On compiling, it will generate following JavaScript code.

```
//Generated by typescript 1.8.10
var Car = (function () {
   //constructor
   function Car(engine) {
      this.engine = engine;
   }

   //function
   Car.prototype.disp = function () {
      console.log("Engine is  :   " + this.engine);
   };
   return Car;
}());
```

## Creating Instance objects

To create an instance of the class, use the **new** keyword followed by the class name. The syntax for the same is given below −

### Syntax

```
var object_name = new class_name([ arguments ])
```

- The **new** keyword is responsible for instantiation.
- The right-hand side of the expression invokes the constructor. The constructor should be passed values if it is parameterized.

Example: Instantiating a class

```
var obj = new Car("Engine 1")
```

## Accessing Attributes and Functions

A class's attributes and functions can be accessed through the object. Use the ' . ' dot notation (called as the period) to access the data members of a class.

```
//accessing an attribute
obj.field_name

//accessing a function
obj.function_name()
```

Example: Putting them together

```
class Car {
   //field
   engine:string;

   //constructor
   constructor(engine:string) {
      this.engine = engine
   }

   //function
   disp():void {
      console.log("Function displays Engine is  :
"+this.engine)
   }
}

//create an object
var obj = new Car("XXSY1")

//access the field
console.log("Reading attribute value Engine as :  "+obj.engine)

//access the function
obj.disp()
```

The output of the above code is as follows −

```
Reading attribute value Engine as :  XXSY1
Function displays Engine is  :   XXSY1
```

## Class Inheritance

TypeScript supports the concept of Inheritance. Inheritance is the ability of a program to create new classes from an existing class. The class that is extended to create newer classes is called the parent class/super class. The newly created classes are called the child/sub classes.

A class inherits from another class using the ‘extends’ keyword. Child classes inherit all properties and methods except private members and constructors from the parent class.

### Syntax

```
class child_class_name extends parent_class_name
```

However, TypeScript doesn’t support multiple inheritance.

### Example: Class Inheritance

```
class Shape {
   Area:number

   constructor(a:number) {
      this.Area = a
   }
}

class Circle extends Shape {
   disp():void {
      console.log("Area of the circle:  "+this.Area)
   }
}

var obj = new Circle(223);
obj.disp()
```

The output of the above code is as follows −

```
Area of the Circle: 223
```

The above example declares a class Shape. The class is extended by the Circle class. Since there is an inheritance relationship between the classes, the child class i.e. the class Car gets an implicit access to its parent class attribute i.e. area.

Inheritance can be classified as −

- **Single** − Every class can at the most extend from one parent class
- **Multiple** − A class can inherit from multiple classes. TypeScript doesn’t support multiple inheritance.
- **Multi-level** − The following example shows how multi-level inheritance works.

### Example

```
class Root {
   str:string;
}

class Child extends Root {}
class Leaf extends Child {} //indirectly inherits from Root by
virtue of inheritance

var obj = new Leaf();
obj.str ="hello"
console.log(obj.str)
```

The class Leaf derives the attributes from Root and Child classes by virtue of multi-level inheritance.

## TypeScript ─ Class inheritance and Method Overriding

Method Overriding is a mechanism by which the child class redefines the superclass's method. The following example illustrates the same −

```
class PrinterClass {
   doPrint():void {
      console.log("doPrint() from Parent called…")
   }
}

class StringPrinter extends PrinterClass {
   doPrint():void {
      super.doPrint()
      console.log("doPrint() is printing a string…")
   }
}

var obj = new StringPrinter()
obj.doPrint()
```

The super keyword is used to refer to the immediate parent of a class. The keyword can be used to refer to the super class version of a variable, property or method. Line 13 invokes the super class version of the doWork() function.

The output of the above code is as follows −

```
doPrint() from Parent called…
doPrint() is printing a string…
```

## The static Keyword

The static keyword can be applied to the data members of a class. A static variable retains its values till the program finishes execution. Static members are referenced by the class name.

### Example

```
class StaticMem {
   static num:number;

   static disp():void {
      console.log("The value of num is"+ StaticMem.num)
   }
}

StaticMem.num = 12      // initialize the static variable
StaticMem.disp()       // invoke the static method
```

The output of the above code is as follows −

```
The value of num is 12
```

## The instanceof operator

The **instanceof** operator returns true if the object belongs to the specified type.

### Example

```
class Person{ }
var obj = new Person()
var isPerson = obj instanceof Person;
console.log(" obj is an instance of Person " + isPerson);
```

The output of the above code is as follows −

```
obj is an instance of Person True
```

## Data Hiding

A class can control the visibility of its data members to members of other classes. This capability is termed as Data Hiding or Encapsulation.

Object Orientation uses the concept of access modifiers or access specifiers to implement the concept of Encapsulation. The access specifiers/modifiers define the visibility of a class's data members outside its defining class.

The access modifiers supported by TypeScript are −

| S.No. | Access Specifier & Description |
|---|---|
| 1. | **Public**<br>A public data member has universal accessibility. Data members in a class are public by default. |
| 2. | **Private**<br>Private data members are accessible only within the class that defines these members. If an external class member tries to access a private member, the compiler throws an error. |
| 3. | **Protected**<br>A protected data member is accessible by the members within the same class as that of the former and also by the members of the child classes. |

### Example

Let us now take an example to see how data hiding works −

```
class Encapsulate {
   str:string = "hello"
   private str2:string = "world"
}

var obj = new Encapsulate()
console.log(obj.str)      //accessible
console.log(obj.str2)    //compilation Error as str2 is private
```

The class has two string attributes, str1 and str2, which are public and private members respectively. The class is instantiated. The example returns a compile time error, as the private attribute str2 is accessed outside the class that declares it.

## Classes and Interfaces

Classes can also implement interfaces.

```
interface ILoan {
   interest:number
}

class AgriLoan implements ILoan {
   interest:number
   rebate:number

   constructor(interest:number,rebate:number) {
      this.interest = interest
      this.rebate = rebate
   }
}

var obj = new AgriLoan(10,1)
console.log("Interest is : "+obj.interest+" Rebate is :
"+obj.rebate )
```

The class AgriLoan implements the interface Loan. Hence, it is now binding on the class to include the property **interest** as its member.

The output of the above code is as follows −

```
Interest is : 10 Rebate is : 1
```

Thank You