02

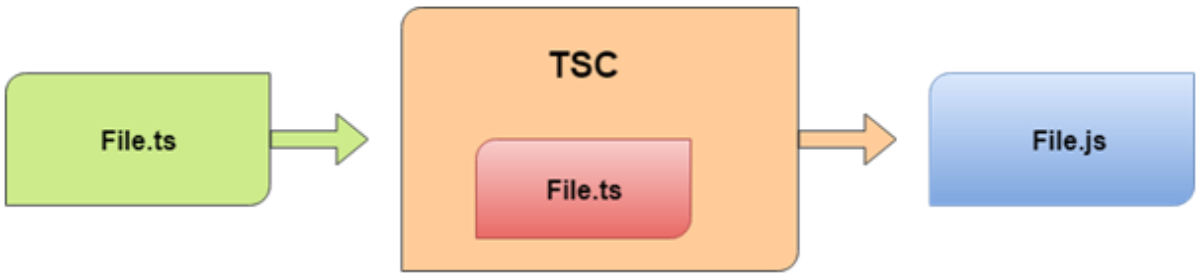
TypeScript

WHAT YOU WILL LEARN

Contents

TypeScript Introduction

- TypeScript developed by Microsoft company in 2012
- TypeScript is an open-source pure object-oriented programing language.
- TypeScript is a programming language which is developed by using Java Script.
- TypeScript is superset of java script which adds data types + classes + Interfaces etc...
- 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.



Advantages of TypeScript

- **highlights errors:** Typescript always highlights errors at compilation time during the time of development, whereas JavaScript points out errors at the runtime.
- **strongly typed**: TypeScript supports strongly typed or static typing, whereas this is not in JavaScript.
- **Namespace**: It has a namespace concept by defining a module.

Disadvantage of TypeScript over JavaScript

- TypeScript takes a long time to compile the code.
- The TypeScript application in the browser, a compilation step is required to transform TypeScript into JavaScript.

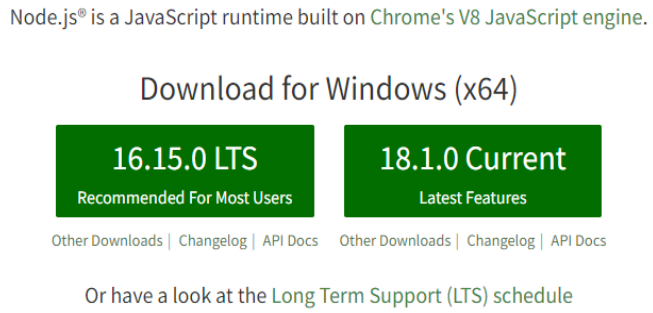
Java Script vs TypeScript

SN	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.log('Sum of the numbers is: ' + sum);`

Environment Setup to work TypeScript

1) Install Node JS (nodejs.org)
    a. Search in browser https://nodejs.org/



    b. Download LTS Version
    c. Install node and check version of node
    d. Node -v : command is used to check installed node version

2) Install TypeScript (in command prompt npm install -g typescript)
    a. npm install -g typescript: in command prompt
    b. tsc -v : will show current installed version
3) Install VS Code and Setup Folder

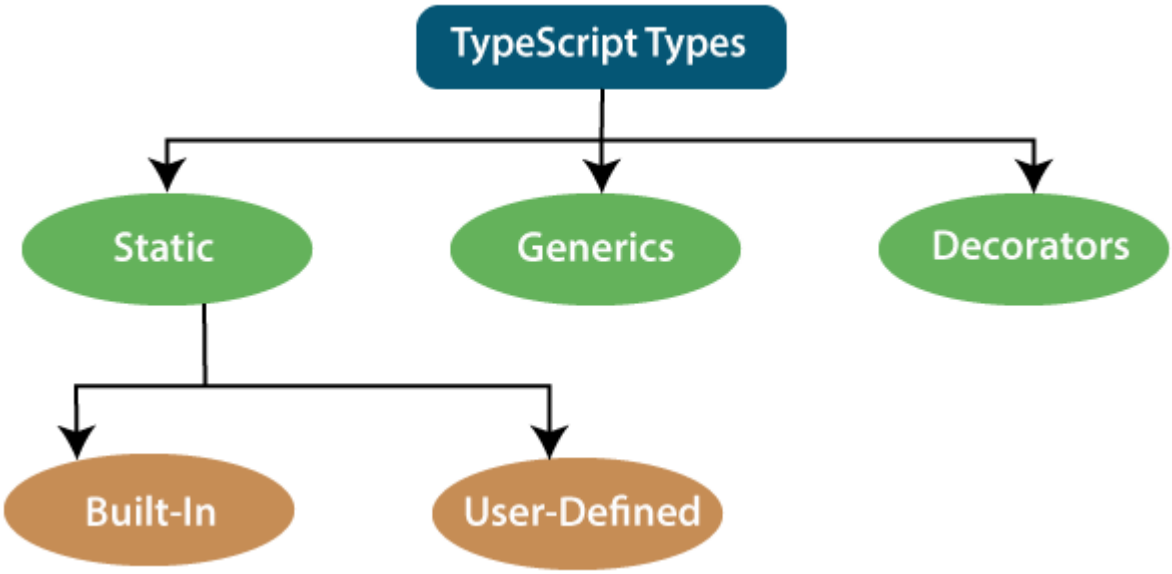
Variables

A variable is the storage location, which is used to store value/information.

Rules are:

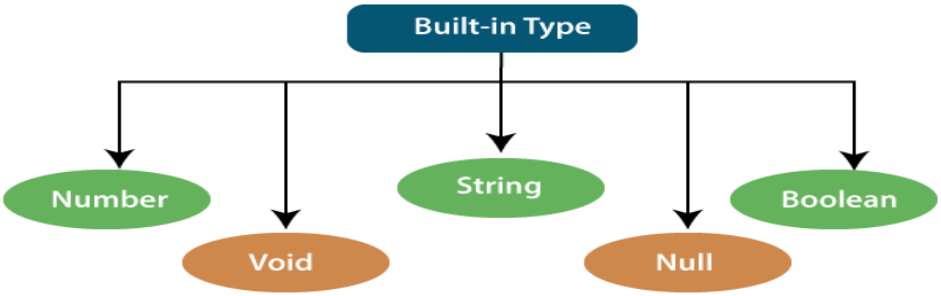
- The variable name must be an **alphabet** or **numeric digits**.
- The variable name cannot start with digits.
- The variable name cannot contain **spaces** and **special character**, except the u**nderscore(_)** and the **dollar($)** sign.
- In **ES6**, we can define variables using **let** and **const** keyword.

Datatypes

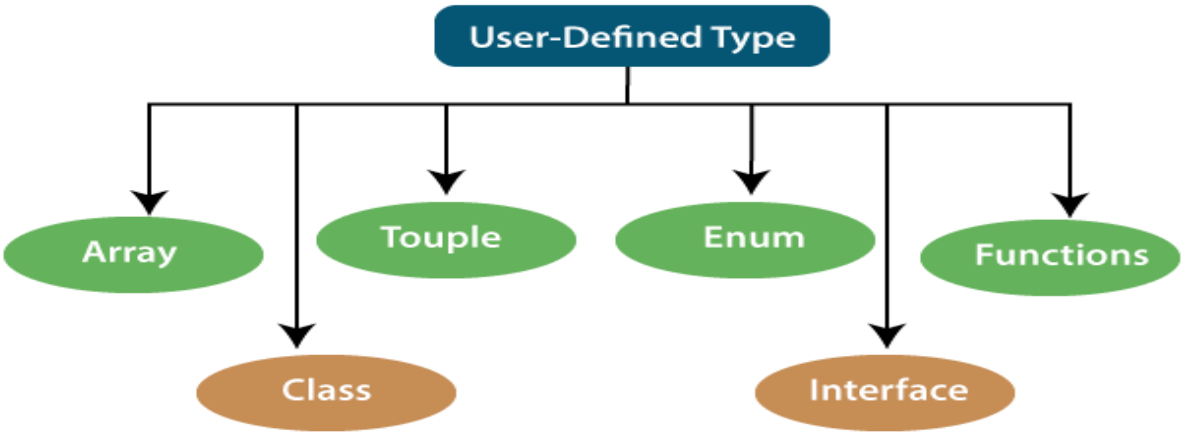


Built-in or Primitive Type

The TypeScript has five built-in data types, which are given below.



- **Number:**
- **String :** the string data type to represents the text in TypeScript.
- **Boolean :** A Boolean value is a truth value which specifies whether the condition is true or not.
- **Void :** A void is a return type of the functions which do not return any type of value.
- **Null :** Null represents a variable whose value is undefined.
- **Undefined** : The Undefined primitive type denotes all uninitialized variables in TypeScript
- **Any :** It is the "super type" of all data type in TypeScript. It allows us to opt-in and opt-out of type-checking during compilation. 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.



- **Array:** An array is a collection of elements of the same data type.
    - TypeScript also allows us to work with arrays of values. An array can be written in two ways:
    - Use the type of the elements followed by [] to denote an array of that element type:
        - var list : number[] = [1, 3, 5];
    - The second way uses a generic array type:
        - 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.
        - let a: [string, number];
        - a = ["hi", 8, "how", 5]; // OK

Functions

- Functions are the building blocks of readable, maintainable, and reusable code.
- A function is a set of statements to perform a specific task.

Advantage of function

- **Code reusability:** We can call a function several times without writing the same block of code again. The code reusability saves time and reduces the program size.
- **Less coding:** Functions makes our program compact. So, we don't need to write many lines of code each time to perform a common task.
- **Easy to debug:** It makes the programmer easy to locate and isolate faulty information.

Function Aspects

There are three aspects of a function.

1. **Function declaration:** A function declaration tells the compiler about the function name, function parameters, and return type. The syntax of the function declaration is:

   function functionName([arg1, arg2, ...argN]);
2. **Function definition:** It contains the actual statements which are going to executes. It specifies what and how a specific task would be done. The syntax of the function definition is:

   function functionName([arg1, arg2, ...argN]){ }
3. **Function call:** We can call a function from anywhere in the program. The parameter/argument cannot differ in function calling and a function declaration. We must pass the same number of functions as it is declared in the function declaration. The syntax of the function call is:

   FunctionName();

Types of Function

1. **Named functions:** When we declare and call a function by its given name, then this type of function is known as a **named** function.

```
function welcome(){
   console.log("Welcome to Career Infotech");
}

welcome( );
```

2. **Anonymous functions :** A function without a name is known as an anonymous function. These type of functions are dynamically declared at runtime. It is defined as an expression. We can store it in a variable, so it does not need function names.

```
let result = function (x:number, y:number) : number {
    return x+y;
      }
    result (10,20);
```

we will use functions difference category of functions as below

1) Simple Function:

Simple function is function which does not have parameter or return

```
Example – function fun(){
Console.log(“hello”);
}
fun();
```

2) Function Parameters

Function parameter can be categories into the following:

- **Optional Parameter**
  - Optional Parameter should be the last argument
  - Optional Parameter will be represented using ?

```
function showDetails(id:number,name:string,e_mail_id?:string) {
  console.log("ID:", id, " Name:",name);
  if(e_mail_id!=undefined)
  console.log("Email-Id:",e_mail_id);
}
showDetails(101,"Virat Kohli");
showDetails(105,"Sachin","sachin@ci.com");
```

- **Default Parameter**
  - If the user does not pass a value to an argument, TypeScript initializes the default value for the parameter.

```
function displayName(name: string, greeting: string =
"Hello") : string {
   return greeting + ' ' + name + '!';
}
console.log(displayName('Career'));
console.log(displayName('career', 'Hi'));
```

- **Rest Parameter**
  - The rest parameter is used to pass **zero or more** values to a function.
  - **Rules to follow in rest parameter:**
    - Only one rest parameter is allowed in a function.
    - **It must be the last parameter in a parameter list.**

```
function sum(a: number, ...b: number[]): number
{
 let result = a;
 for (var i = 0; i < b.length; i++) {
 result += b[i];
 }
 return result;
}
let result1 = sum(3, 5);
let result2 = sum(3, 5, 7, 9);
console.log(result1 +"\n" + result2);
```

3) **Function return:** Functions may also return value along with control, back to the caller. Such functions are called as returning functions.
   i. The return_type can be any valid data type.
   ii. A returning function must end with a return statement.
   iii. A function can return at the most one value.
   iv. The data type of the value returned must match the return type of the function.

Example:

```
//function defined
function greet():string { //the function returns a string
   return "Hello World"
}

function caller() {
   var msg = greet() //function greet() invoked
   console.log(msg)
}

//invoke function
caller()
```

4) **Arrow function**
    a. ES6 version of TypeScript provides an arrow function which is the shorthand syntax for defining the anonymous function
    b. **Syntax**
        i. **Parameters:** A function may or may not have parameters.
        ii. **The arrow notation/lambda notation** (=>)
        iii. **Statements:** It represents the function's instruction set.
    c. **Example-**

There are two ways of writing a function in ES5 and ES6 style of coding.

```
// ES5: Without arrow function
var getResult = function(username, points) {
  return username + ' scored ' + points + ' points!';
};

// ES6: With arrow function
var getResult = (username: string, points: number): string => {
  return `${ username } scored ${ points } points!`;
}
```

5) **Recursive function**
6) **Callback function**

Class Object and Constructor OOPS

OOPS stands for object-oriented programming system
In OOPS programming languages "Classes" are fundamental entities which are used to create re-usable components.
"Class" is a plan or model which is used to create the object.

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.

Example

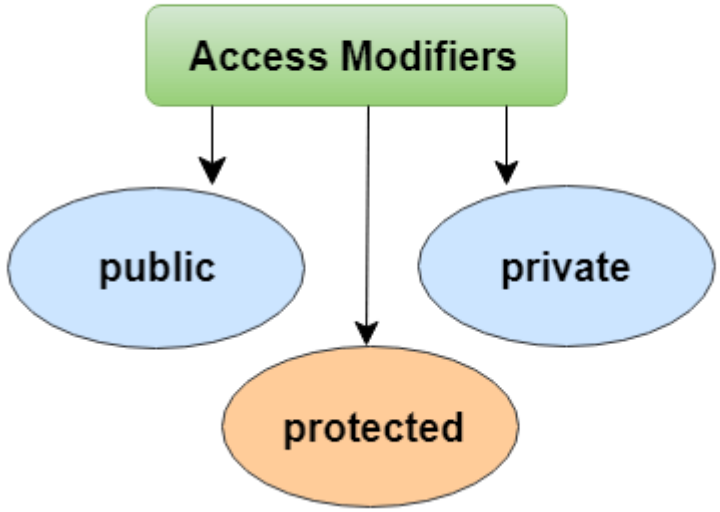
```
class <class_name>{
   field;
   method;
}
```

Access Modifiers

Data Hiding or Encapsulation

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.

TypeScript supports the three types of access modifier. These are:



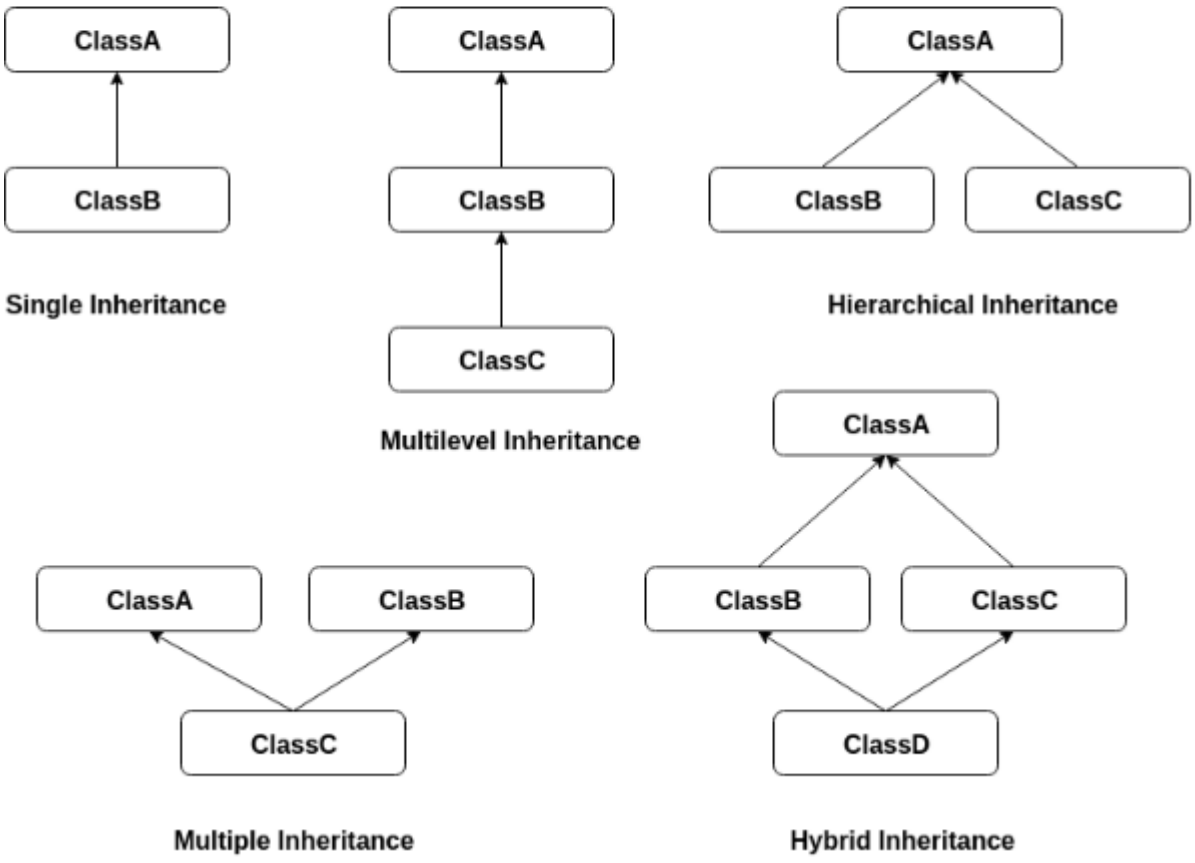
Inheritance

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.

Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance



Interfaces (Contracts)

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 interface contains only the **declaration** of the **methods** and **fields**, but not the **implementation**.

Syntax –

```
interface interface_name {
        // variables' declaration
        // methods' declaration
}
```

- An **interface** is a keyword which is used to declare a TypeScript Interface.
- An **interface_name** is the name of the interface.
- An interface body contains variables and methods declarations.

Example

```
interface OS {
    name: String;
    language: String;
}
let OperatingSystem = (type: OS): void => {
  console.log('Android ' + type.name + ' has ' + type.language + ' language.');
};
let Oreo = {name: 'O', language: 'Java'}
OperatingSystem(Oreo);
```

Enumerations

Enums stands for **Enumerations**. Enums are a new data type supported in TypeScript. It is used to define the set of **named constants**, i.e., a collection of related values. TypeScript supports both **numeric** and **string-based** enums. We can define the enums by using the **enum** keyword.

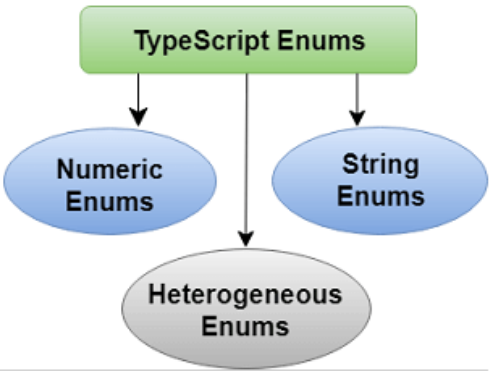
Why Enums?

Enums are useful in TypeScript because of the following:

- It makes it easy to change values in the future.
- It reduces errors which are caused by transporting or mistyping a number.
- It exists only during compilation time, so it does not allocate memory.
- It saves runtime and compile-time with inline code in JavaScript.
- It allows us to create constants that we can easily relate to the program.
- It will enable developers to develop memory-efficient custom constants in JavaScript, which does not support enums, but TypeScript helps us to access them.

There are **three** types of Enums in TypeScript. These are:

- Numeric Enums
- String Enums
- Heterogeneous Enums



Modules

JavaScript has a concept of modules from ECMAScript 2015. TypeScript shares this concept of a module.

A module is a way to create a group of related variables, functions, classes, and interfaces, etc. It executes in the **local scope**, not in the **global scope**.

the variables, functions, classes, and interfaces declared in a module cannot be accessible outside the module directly.

We can create a module by using the **export** keyword and can use in other modules by using the **import** keyword.

Modules import another module by using a **module loader**. At runtime, the module loader is responsible for locating and executing all dependencies of a module before executing it.

The most common modules loaders which are used in JavaScript are the **CommonJS** module loader for **Node.js** and **require.js** for Web applications.

We can divide the module into **two** categories:

1. Internal Module

Internal modules were in the **earlier version** of Typescript.

It was used for **logical grouping** of the classes, interfaces, functions, variables into a single unit and can be exported in another module.

The modules are named as a **namespace** in the latest version of TypeScript

Internal Module Syntax in Earlier Version:

```
module Sum {
   export function add(a, b) {
      console.log("Sum: " +(a+b));
   }
}
```

Internal Module Syntax from ECMAScript 2015:

```
namespace Sum {
   export function add(a, b) {
      console.log("Sum: " +(a+b));
   }
}
```

2. External Module

Module declaration

We can declare a module by using the **export** keyword. The syntax for the module declaration is given below.

```
//FileName : EmployeeInterface.ts
export interface Employee {
   //code declarations
}
```

We can use the declare module in other files by using an **import** keyword, which looks like below. The **file/module** name is specified without an **extension**.

```
import { class/interface name } from 'module_name';
```

Module Creation: **addition.ts**

```
export class Addition{
    constructor(private x?: number, private y?: number){
    }
    Sum(){
        console.log("SUM: " +(this.x + this.y));
    }
}
```

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

```
import {Addition} from './addition';

let addObject = new Addition(10, 20);

addObject.Sum();
```

Compiling and Executing Modules

Open the **terminal** and go to the location where you stored your **project**. command in the terminal window.

```
$ tsc --module commonjs app.ts
$ node ./app.js
```