ITC002A

# Typescript Fundamentals

Version 4.9

# Table of Contents

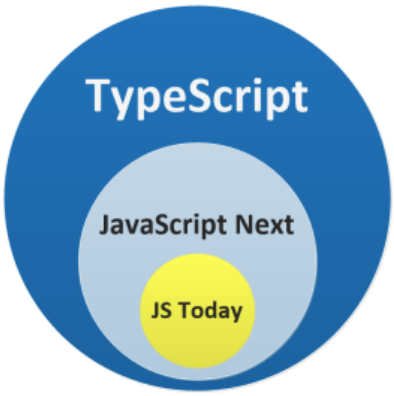
# Module 1: Getting Started with Typescript

## 1.1 OVERVIEW

JavaScript was introduced as a language for the client side. The development of Node.js has marked JavaScript as an emerging server-side technology too. However, as JavaScript code grows, it tends to get messier, making it difficult to maintain and reuse the code. Moreover, its failure to embrace the features of Object Orientation, strong type checking and compile-time error checks prevents JavaScript from succeeding at the enterprise level as a full-fledged server-side technology. TypeScript was presented to bridge this gap.

## 1.2 TYPESCRIPT

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. It's commonly used for application-scale development. TypeScript was first released in October 2012.




## 1.3 JAVASCRIPT, ECMASCRIPT AND TYPESCRIPT

JavaScript is a general-purpose scripting language that conforms to the ECMAScript specification. The ECMAScript specification is a blueprint for creating a scripting language. JavaScript is an implementation of that blueprint.

On the whole, JavaScript implements the ECMAScript specification as described in ECMA-262. ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6, since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020).


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.

**ECMAScript Features**

| Version | Official name | Features |
|---|---|---|
| **ES1** | ECMAScript 1 (1997) | First edition |
| **ES2** | ECMAScript 2 (1998) | Editorial changes |
| **ES3** | ECMAScript 3 (1999) | Added regular expressions<br>Added try/catch<br>Added switch<br>Added do-while |
| **ES4** | ECMAScript 4 | Never released |
| **ES5** | ECMAScript 5 (2009) | Added "strict mode"<br>Added JSON support<br>Added String.trim()<br>Added Array.isArray()<br>Added Array iteration methods<br>Allows trailing commas for object literals |
| **ES6** | ECMAScript 2015 | Added let and const<br>Added default parameter values<br>Added Array.find()<br>Added Array.findIndex() |
| **ES2016** | ECMAScript 2016 | Added exponential operator (**)<br>Added Array.includes() |
| **ES2017** | ECMAScript 2017 | Added string padding<br>Added Object.entries()<br>Added Object.values()<br>Added async functions<br>Added shared memory<br>Allows trailing commas for function parameters |
| **ES2018** | ECMAScript 2018 | Added rest / spread properties<br>Added asynchronous iteration |

| | | |
|---|---|---|
| | | Added Promise.finally()<br>Additions to RegExp |
| **ES2019** | ECMAScript 2019 | String.trimStart()<br>String.trimEnd()<br>Array.flat()<br>Object.fromEntries<br>Optional catch binding |
| **ES2020** | ECMAScript 2020 | The Nullish Coalescing Operator (??) |

## 1.4 PURPOSE OF TYPESCRIPT

JavaScript is a dynamic programming language with no type system. JavaScript doesn't check assigned values, variables are declared using the var keyword, and it can point to any value. JavaScript doesn't support classes and other object-oriented features (ECMA2015 supports it). So, without the type system, it is not easy to use JavaScript to build complex applications with large teams working on the same code.

The type system increases the code quality, readability and makes it easy to maintain and refactor codebase. More importantly, errors can be caught at compile time rather than at runtime.

Hence, the reason to use TypeScript is that it catches errors at compile-time, so that you can fix it before you run code. It supports object-oriented programming features like data types, classes, enums, etc., allowing JavaScript to be used at scale.

TypeScript compiles into simple JavaScript. The TypeScript compiler is also implemented in TypeScript and can be used with any browser or JavaScript engines like Node.js. TypeScript needs an ECMAScript 3 or higher compatible environment to compile. This is a condition met by all major browsers and JavaScript engines today.

Some of the most popular JavaScript frameworks like Angular are written in TypeScript.

## 1.5 ADVANTAGES OF 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:

### 1.5.1 Static type checking

Static type checking is defined as **type checking performed at compile time**. It checks the type variables at compile-time, which means the type of the variable is known at the compile time.

A static type-checker like TypeScript, ***a static types systems*** describe the shapes and behaviors of what our values will be when we run our programs. TypeScript uses that information and informs the developer where the type errors are.

**In dynamically-typed languages, the errors occur only once the program is executed** for example in JavaScript this code is valid:

```
let value = 5;
value = "hello";
```

Here, the type of `value` changes from a number to a string. In TypeScript, this is forbidden.

```
let value = 5;
value = "hello"; // error: Type '"hello"' is not assignable to type
'number'.
```

There's one famous example of JavaScript being weird. If a program inputs a number, and the user inputs a string, you might try to do some arithmetic on it.

```
// userInput = "2"
console.log(userInput + 2);
```

You may expect the console to print "4", but JavaScript doesn't do this. Instead, it gives you, "22". Because **`userInput`** is a string, adding a number to it will convert the number to a string. JavaScript concatenates the two strings together, and the result is "22".

If you use JavaScript, you're not likely to know what the type for some object is. It's possible you won't realize what methods a certain object has, or even what fields it has. It will run just fine, until you need to use the result of that property, causing semantic errors. The real benefit to using TypeScript is to prevent those types of errors.

### 1.5.2 Avoiding Non-Exception failures

We have been discussing certain things like runtime errors - cases where the JavaScript runtime tells us that it thinks something is nonsensical. Those cases come up because the ECMAScript specification has explicit instructions on how the language should behave when it runs into something unexpected.

For example, the specification says that trying to call something that isn't callable should throw an error. Maybe that sounds like "obvious behavior", but you could imagine that accessing a property that doesn't exist on an object should throw an error too. Instead, JavaScript gives us different behavior and returns the value **`undefined`**:

```
const user = {
    name: "Daniel",
    age: 26,
  };
  user.location; // returns undefined
```

Ultimately, Typescript has to make the call over what code should be flagged as an error in its system, even if it's "valid" JavaScript that won't immediately throw an error. In TypeScript, the following code produces an error about `location` not being defined:

```
const user = {
    name: "Daniel",
    age: 26,
  };

  user.location;

  Property 'location' does not exist on type '{ name: string; age:
number; }'.
```

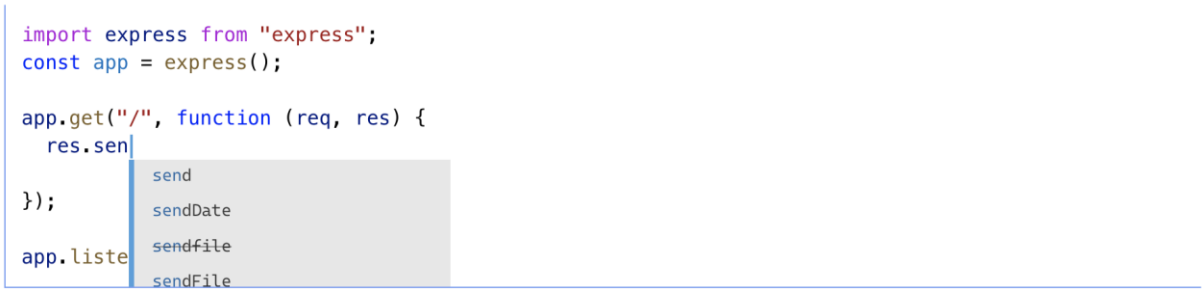
While sometimes that implies a trade-off in what you can express, the intent is to catch legitimate bugs in our programs.

### 1.5.3 Types for tooling

TypeScript can catch bugs when we make mistakes in our code. That's great, but TypeScript can *also* prevent us from making those mistakes in the first place.

The type-checker has information to check things like whether we're accessing the right properties on variables and other properties. Once it has that information, it can also start suggesting which properties you might want to use.

That means TypeScript can be leveraged for editing code too, and the core type-checker can provide error messages and code completion as you type in the editor. That's part of what people often refer to when they talk about tooling in TypeScript.




TypeScript takes tooling seriously, and that goes beyond completions and errors as you type. An editor that supports TypeScript can deliver "quick fixes" to automatically fix errors, refactoring to easily re-organize code, and useful navigation features for jumping to definitions of a variable, or finding all references to a given variable. All of this is built on top of the type-checker.

## 1.6 TSC: TYPESCRIPT COMPILER

### 1.6.1 Setup

NPM (NodeJS Package Manager) is used to install the Typescript package on your local machine or a project. Make sure you have NodeJS runtime installed on your local machine.

To install Typescript, run the following command:

```
npm install -g typescript
```

The above command will install TypeScript globally so that you can use it in any project. Check the installed version of TypeScript using the following command:

```
tsc -v
```

### 1.6.2 Compiling TS into JS

TypeScript code is written in a file with **.ts** extension and then compiled into JavaScript using the TypeScript compiler. A TypeScript file can be written in any code editor. A TypeScript compiler needs to be installed on your platform. Once installed, the command **tsc <filename>.ts** compiles the TypeScript code into a plain JavaScript file. JavaScript files can then be included in the HTML and run on any browser or run-in server-side using NodeJS runtime.


Now we can write our first TypeScript program: **hello.ts**:

```
function greet(person, date) {
  console.log(`Hello ${person}, today is ${date}!`);
}

greet("Brendan");
```

Let's type-check it by running the command **tsc** which was installed for us by the typescript package.

```
tsc hello.ts
```

If we run this command, notice that we get an error on the command line!

```
Expected 2 arguments, but got 1.
```

TypeScript is telling us we forgot to pass an argument to the greet function, and rightfully so. So far, we've only written standard JavaScript, and yet type-checking was still able to find problems with our code.

### 1.6.3 Typescript configuration

The presence of a **tsconfig.json** file in a directory indicates that the directory is the root of a TypeScript project. The **tsconfig.json** file specifies the root files and the compiler options required to compile the project.

By invoking **`tsc`** with no input files, the compilation is executed on the root directory which contains the **`tsconfig.json`** file.

To initialize a folder as a typescript project, and add **`tsconfig.json`**, run the following command:

```
tsc --init
```

This will create the TS configuration file (**`tsconfig.json`**), the following is the default configuration:

```
{
    "compilerOptions": {
      // project options
      "lib": [
        "ESNext",
        "dom"
      ],
      "outDir": "lib",
      "removeComments": true,
      "target": "ES6",

      // Module resolution
      "baseUrl":
      "esModuleInterop": true
      "moduleResolution": "node",
      "paths": {},

      // Source Map
      "sourceMap": true,
      "sourceRoot": "/",

      // Strict Checks
      "alwaysStrict": true,
      "allowUnreachableCode": false,
      "noImplicitAny": true,
      "strictNullChecks": true

      // Linter Checks
      "noImplicitReturns": true,
      "noUncheckedIndexedAccess": true,
      "noUnusedLocals": true,
      "noUnusedParameters": true
    },
    "include": ["./**/*.ts"],
    "exclude": [
      "node_modules/**/*"
    ]
  }
```

# Module 2: Types, Variables and Functions

## 1.1 OVERVIEW

In this module, we'll cover some of the most common types of values you'll find in JavaScript code, and explain the corresponding ways to describe those types in TypeScript.

Types can also appear in many more places than just type annotations. As we learn about the types themselves, we'll also learn about the places where we can refer to these types to form new constructs.

We'll start by reviewing the most basic and common types you might encounter when writing JavaScript or TypeScript code. These will later form the core building blocks of more complex types.

## 1.2 TYPE ANNOTATION

### 1.2.1 Static VS Dynamic typing

JavaScript is a dynamically typed language, but **TypeScript is a statically typed language**. Longer answer: In dynamically typed languages all type checks are performed in a runtime, only when your program is executing.

- **Dynamic typing**: the type is associated with the value, and checked at **run-time**.

```
function g( a ) {
    return a / 10;
}
```

- **Static typing**: type is associated with variable or *textual expression, and checked at* ***compile-time*****.**

```
function g( a : number ) : number{
    return a / 10;
}
```

### 1.2.2 Annotations

As motioned before TypeScript is a static typed language, where we can specify the type of the variables, function parameters and object properties. We can specify the type using **:Type** after the name of the variable, parameter or property. There can be a space after the colon. TypeScript includes all the primitive types of JavaScript- **`number`**, **`string`** and **`boolean`** (More details in the next section)

```
var age: number = 32; // number variable
var name: string = "John";// string variable
```

In the above example, each variable is declared with their data type. These are type annotations. You cannot change the value using a different data type other than the declared data type of a variable. If you try to do so, TypeScript compiler will show an error. This helps in catching JavaScript errors. For example, if you assign a string to a variable **age** or a number to **name** in the above example, then it will give an error.

Type annotations are used to enforce type checking. It is not mandatory in TypeScript to use type annotations. However, type annotations help the compiler in checking types and helps avoid errors dealing with data types. It is also a good way of writing code for easier readability and maintenance by future developers working on your code.

### 1.2.3 Variable

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

- **var**: Variables in TypeScript can be declared using var keyword, same as in JavaScript. The scoping rules remains the same as in JavaScript.
- **let**: 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.
- **const**: 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.

```
var n: number = 10
const greeting = "Hello" //Never changes

function f() {
    let n: number = 0 //Scoped to f function
}
```

## 1.3 PRIMITIVE TYPES

Simple types are also called the primitive types and they belong to built-in predefined types found in TypeScript. The primitive data type is **string**, **number**, **boolean**, **null**, and **undefined** types. There are the following primitive or built-in types in TypeScript, which are described below:

### 1.3.1 number

The number primitive type is the same as the JavaScript primitive type and represents double-precision 64-bit IEEE (Institute of Electrical and Electronics Engineers) 754 format floating-point values. The number keyword is used to represent number type values (integer of float).

```
var n: number
n = 2
n = 10.5
```

### 1.3.2 string

The `string` primitive type is the same as the JavaScript primitive type and it represents a sequence of characters stored as Unicode UTF-16 code. String values are surrounded by single quotation marks or double quotation marks.

```
var text: string;
text = "hello world"
text = 'hello world'
```

### 1.3.3 boolean

The `boolean` primitive type is the same as the JavaScript primitive type and represents a logical value; either true or false.

```
var b: boolean;
b = true
b = false
```

### 1.3.4 null

The `null` primitive type is the same as the JavaScript primitive type and represents a null literal and it is possible to directly reference the null type value itself. In other words, null is both value and a type.

```
var n = null;
var n: null;
```

### 1.3.5 undefined

The undefined type is the same as the JavaScript primitive type and is the type of the undefined literal. A variable is `undefined` when it's not assigned any value after being declared. `null` refers to a value that is either empty or doesn't exist. null means no value. To make a variable null we must assign null value to it as by default in typescript unassigned values are termed undefined.

```
var n = undefined;
var n: undefined;
```

## 1.4 SPECIAL TYPES

### 1.4.1 Any

TypeScript also has a special type, **any**, that you can use whenever you don’t want a particular value to cause type-checking errors.

In TypeScript, every type is assignable to **any**. This makes **any** a top-type (also known as a universal supertype) of the type system.

When a value is of type any, you can access any properties of it (which will in turn be of type any), call it like a function, assign it to (or from) a value of any type, or anything else that's syntactically legal.

```
let value: any;
value = true; // OK
value = 42; // OK
value = "Hello World"; // OK
value = null; // OK
value = undefined; // OK
```

### 1.4.2 Unknown

Just like all types are assignable to **any**, all types are assignable to **unknown**. The **unknown** type is only assignable to the any type and the unknown type itself. Intuitively, this makes sense: only a container that is capable of holding values of arbitrary types can hold a value of type **unknown**; after all, we don't know anything about what kind of value is stored in **unknown** value.

```
let value: unknown;
let some : any;

value = some; // OK
value = true; // OK
value = 42; // OK
value = "Hello World"; // OK
value = null; // OK
value = undefined; // OK
```

### 1.4.3 Never

TypeScript introduced a new type **never**, which indicates the values that will never occur. The never type is used when you are sure that something is never going to occur. For example, you write a function which will not return to its end point or always throws an exception.

```
function throwError(errorMsg: string): never {
    throw new Error(errorMsg);
}

function keepProcessing(): never {
    while (true) {
      console.log('I always does something and never ends.')
    }
}
```

### 1.4.4 Void

Similar to other languages like Java, **void** is used where there is no data. For example, if a function does not return any value, then you can specify **void** as return type.

Difference between never and void is that the void type can have undefined or null as a value whereas never cannot have any value.

```
function sayHi(): void {
    console.log('Hi!')
}
```

There is no meaning to assign void to a variable, as only null or undefined is assignable to void.

```
let empty: void = undefined; //OK
let num: void = 1; // Error
```

### 1.4.5 Array

An array is a special type of data type which can store multiple values of same data type sequentially using a special syntax.

TypeScript supports arrays, similar to JavaScript. There are two ways to declare an array:

- Using square brackets. This method is similar to how you would declare arrays in JavaScript.

  ```
  let fruits: string[] = ['Apple', 'Orange', 'Banana'];
  ```

- Using a generic array type, **Array<type>**.

  ```
  let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];
  ```

**Note:** **number[]** and **[number]** are two different things (refer to the section on Tuples).

### 1.4.6 Tuple

TypeScript introduced a new data type called Tuple. Tuple can contain two values of different data types. Consider the following example of number, string and tuple type variables.

```
var empId: number = 1;
var empName: string = "Steve";

// Tuple type variable
var employee: [number, string] = [empId, empName];
employee = [2, "another"]
```

## 1.5 TYPE INFERENCE

TypeScript is a typed language. However, it is not mandatory to specify the type of a variable. TypeScript infers types of variables when there is no explicit information available in the form of type annotations.

Types are inferred by TypeScript compiler when:

- Variables are initialized
- Default values are set for parameters
- Function return types are determined

For example,

```
var a = "some text"
```

Here, since we are not explicitly defining **`a: string`** with a type annotation, TypeScript infers the type of the variable based on the value assigned to the variable. The value of a is a string and hence the type of a is inferred as **`string`**.

```
var a = "some text";
var b = 123;
a = b; // Error: Type 'number' is not assignable to type 'string'
```

## 1.6 TYPE ASSERTION

Type assertion allows you to set the type of a value and tell the compiler not to infer it. This is when you, as a programmer, might have a better understanding of the type of a variable than what TypeScript can infer on its own. Such a situation can occur when you might be porting over code from JavaScript and you may know a more accurate type of the variable than what is currently assigned.

It is similar to type casting in other languages like C# and Java. However, unlike C# and Java, there is **no runtime effect** of type assertion in TypeScript. It is merely a way to let the TypeScript compiler know the type of a variable.

There are two ways to do type assertion in TypeScript:

- Using the angular bracket "**<>**" syntax. So far in this section, we have used angular brackets to show type assertion.

```
let code: any = 123;
let employeeCode = <number> code;
```

- Using "**as**" keyword

```
let code: any = 123;
let employeeCode = code as number;
```

In the above example, we have a variable code of type any. We assign the value of this variable to another variable called **`employeeCode`**. However, we know that code is of type number, even though it has been declared as 'any'. So, while assigning code to **`employeeCode`**, we have asserted that code is of type number in this case, and we are certain about it. Now, the type of **`employeeCode`** is number.

## 1.7 TYPE ALIASES

In Typescript, Type aliases give a type a new name. They are similar to interfaces in that they can be used to name primitives and any other kinds that you'd have to define by hand otherwise. Aliasing doesn't truly create a new type; instead, it gives that type a new name. Aliasing a primitive isn't very practical as it's easy using primitive types, however, it can be used for documentation purposes.

```
type Num = number
var n : Num = 10
```

Here, we created our custom type **Num** which is simply a number.

It can also be used to rename other types or interfaces like in the following example:

```
interface Shape {
    //properties here
}

type Paint = Shape
```

## 1.8 OBJECT TYPES

In JavaScript, the fundamental way that we group and pass around data is through objects. In TypeScript, we represent those through *object types*.

There are two different general types for objects:

- **`Object`** with an uppercase "O" is the type of all instances of class **`Object`**:

```
let obj1: Object;
```

- **`object`** with a lowercase "o" is the type of all non-primitive values:

```
let obj2: object;
```

If you reassign a primitive value to the **`obj2`** object, you'll get an error:

```
employee = "Jane";

//error TS2322: Type '"Jane"' is not assignable to type 'object'.
```

### 1.8.1 Object literal

An object literal is a comma-separated list of name-value pairs inside of curly braces. Those values can be properties and functions. in Typescript Objects can also be described via their properties:

```
// Object type literal
let obj3: {prop: boolean};
```

```
interface Person {  //Or type Person
    name: string;
    age: number;
  }
var someone : Person
```

### 1.8.2 Optional properties

Much of the time, we'll find ourselves dealing with objects that *might* have a property set. In those cases, we can mark those properties as *optional* by adding a question mark (**?**) to the end of their names.

```
interface PaintOptions {
    shape: Shape;
    xPos?: number; //Optional
    yPos?: number; //Optional
  }
var paint = { shape : square} //OK
```

In the above example, both **xPos** and **yPos** are considered optional. We can choose to provide either of them, so every call above to **paintShape** is valid. All optionality really says is that if the property *is* set, it better have a specific type.

### 1.8.3 Optional chaining

The optional chaining provides a new (**?.**) operator which allows you to do **optional property access**. In other words, when you try access the property of an object, if TypeScript hits a **null** or **undefined** value then it will immediately return **undefined**. Example:

```
interface Customer {
    name: string;
    email?: string;
    address?: {
        type?: string;
        city?: string;
        state?: string;
    }
}

// Previous way
if (customer && customer.address && customer.address.city) {
    let city = customer.address.city;
}

//New way
let city = customer?.address?.city;
```

### 1.8.4 Non-null assertion

A new (**!**) post-fix expression operator may be used to assert that its operand is non-null and non-undefined in contexts where the type checker is unable to conclude that fact. Specifically, the operation `x!` produces a value of the type of **x** with **null** and **undefined** excluded. Similar to type assertions of the forms **<T>x** and **x as T**, the **(!)** non-null assertion operator is simply removed in the emitted JavaScript code.

```
var customer: Customer = {name: "John"}

var n = customer.name!  //OK
var email = customer.email!  //Error, email is undefined
```

## 1.9 DUCK TYPING

The duck-typing technique in TypeScript is used to compare two objects by determining if they have the same type matching properties and objects members or not.

In duck-typing, two objects are considered to be of the same type if both share the same set of properties. Duck-typing verifies the presence of certain properties in the objects, rather than their actual type, to check their suitability. The concept is generally explained by the following phrase "**When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."**

The TypeScript compiler implements the duck-typing system that allows object creation **on the fly** while keeping type safety. The following example shows how we can pass objects that don't explicitly implement an interface but contain all of the required members to a function.

```
interface IPoint {
    x: number
    y: number
}

var a: IPoint

var b = {x: 1, y: 2}

a = b   //Correct
b = a   // Also correct
```

## 1.10 FUNCTIONS

Functions are the basic building block of any application, whether they're local functions, imported from another module, or methods on a class. They're also values, and just like other values allows you to specify the types of both the input and output values of functions.

### 1.10.1 Parameter Type Annotations

When you declare a function, you can add type annotations after each parameter to declare what types of parameters the function accepts. Parameter type annotations go after the parameter name:

```
// Parameter type annotation
function greet(name: string) {
    console.log("Hello, " + name.toUpperCase() + "!!");
  }
```

When a parameter has a type annotation, arguments to that function will be checked:

```
// Would be a runtime error if executed!
greet(42);

Argument of type 'number' is not assignable to parameter of type
```

### 1.10.2 Optional parameters

We can model this in TypeScript by marking the parameter as *optional* with (**?**):

```
function f(x?: number) {
    // ...
  }
f(); // OK
f(10); // OK
```

You can also provide a parameter **default** value:

```
function f(x = 10) {
    // ...
  }
```

### 1.10.3 Rest parameters

In addition to using optional parameters or overloads to make functions that can accept a variety of fixed argument counts, we can also define functions that take an unbounded number of arguments using rest parameters.

A rest parameter appears after all other parameters, and uses the **spread** (**...**) syntax:

```
function multiply(n: number, ...m: number[]) {
    return m.map((x) => n * x);
  }
  // 'a' gets value [10, 20, 30, 40]
  const a = multiply(10, 1, 2, 3, 4);
```

In TypeScript, the type annotation on these parameters is implicitly **any[]** instead of **any**, and any type annotation given must be of the form **Array<T>**or **T[],** or a tuple **[T]** type

### 1.10.4 Return Type Annotations

You can also add return type annotations. Return type annotations appear after the parameter list:

```
function getFavoriteNumber(): number {
    return 26;
  }
```

Much like variable type annotations, you usually don't need a return type annotation because TypeScript will infer the function's return type based on its **`return`** statements. The type annotation in the above example doesn't change anything. Some codebases will explicitly specify a return type for documentation purposes, to prevent accidental changes, or just for personal preference.

### 1.10.5 Arrow functions

Fat arrow notations are used for anonymous function. They are also called lambda functions in other languages.

```
(param1, param2, ..., paramN) => expression
```

Using fat arrow **`=>`**, we dropped the need to use the **`function`** keyword. Parameters are passed in the parenthesis **`()`**, and the function expression is enclosed within the curly brackets **`{}`.**

```
let sum = (x: number, y: number): number => {
  return x + y;
}
```

In the above example, **`sum`** is an arrow function. **`(x:number, y:number)`** denotes the parameter types, **`:number`** specifies the return type. The fat arrow => separates the function parameters and the function body. The right side of **`=>`** can contain one or more code statements.

# Module 3: Modules and Namespaces

## 1.1 OVERVIEW

Starting with ECMAScript 2015, JavaScript has a concept of modules, Modules provide for better code reuse, stronger isolation and better tooling support for bundling. TypeScript shares this concept.

Namespaces are a TypeScript-specific way to organize code, in this module we'll go over how to use namespaces and modules

## 1.2 MODULES

Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported.

For example, consider the following TypeScript files: file1.ts and file2.ts

```
//file1.ts
var greeting : string = "Hello World!";
```

```
//file2.ts
console.log(greeting); //Prints Hello World!
greeting = "Hello TypeScript"; // allowed
```

The above variable greeting, declared in file1.ts is accessible in file2.ts as well. Not only it is accessible but also it is open to modifications. Anybody can easily override variables declared in the global scope without even knowing they are doing so! This is a dangerous space as it can lead to conflicts/errors in the code.

In JavaScript (or Typescript), Modules are a way to create a local scope in the file. So, all variables, classes, functions, etc. that are declared in a module are not accessible outside the module. A module can be created using the keyword **`export`** and a module can be used in another module using the keyword **`import`**.

Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level. Modules import one another using a module loader. At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it. Well-known module loaders used in JavaScript are Node.js's loader.

## 1.3 EXPORTS

### 1.3.1 Exporting a declaration

Any declaration (such as a variable, function, class, type alias, or interface) can be exported by adding the **`export`** keyword.

```
//ZipCodeValidator.ts
export const numberRegexp = /^[0-9]+$/;
export class ZipCodeValidator implements StringValidator {
  isAcceptable(s: string) {
    return s.length === 5 && numberRegexp.test(s);
  }
}
```

### 1.3.2 Export statement

Export statements are handy when exports need to be renamed for consumers, so the above example can be written as:

```
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
      return s.length === 5 && numberRegexp.test(s);
    }
  }
  export { ZipCodeValidator };
  export { ZipCodeValidator as mainValidator };
```

### 1.3.3 Default export

If we want to export a single value or to have a fallback value for your module, you could use a **`default`** export:

```
export default function cube(x) {
    return x * x * x;
  }
```

## 1.4 IMPORTS

Importing is just about as easy as exporting from a module. Importing an exported declaration is done through using one of the **`import`** forms below:

### 1.4.1 Importing a single export

To import a single item (class, interface, variable, function, ..., etc) use the following syntax:

```
import { ZipCodeValidator } from "./ZipCodeValidator";
let myValidator = new ZipCodeValidator();
```

imports can also be renamed:

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

### 1.4.2 Importing the entire module

To import all items within a module, you can reference it as a single variable use it to access the module exports:

```
import * as Validators from "./ZipCodeValidator";
let myValidator = new Validators.ZipCodeValidator();
```

### 1.4.3 Importing types (TS Specific)

Prior to TypeScript 3.8, you can import a type using **`import`**. With TypeScript 3.8, you can import a type using the **`import`** statement, or using **`import type`**.

```
// Explicitly use import type
import type { APIResponseType } from "./api";
// Explicitly pull out a value (getResponse) and a type
(APIResponseType)
```

Any explicitly marked **`type`** import is guaranteed to be removed from your JavaScript output, therefore having better code optimization.

## 1.5 MODULE RESOLUTION

*Module resolution* is the process the compiler uses to figure out what an import refers to. Consider an import statement like the following:

```
import { a } from "moduleA";
```

in order to check any use of **a**, the compiler needs to know exactly what it represents, and will need to check its definition **`moduleA`**.

First, the compiler will try to locate a file that represents the imported module. To do so the compiler follows one of two different strategies: **Relative** or **Non-relative**. These strategies tell the compiler where to look for **`moduleA`**.

Finally, if the compiler could not resolve the module, it will log an error. In this case, the error would be something like .

```
error TS2307: Cannot find module 'moduleA'
```

### 1.5.1 Relative module imports

A *relative import* is one that starts with **`/`**, **`./`** or **`../`**. Some examples include:

```
import Entry from "./components/Entry";
import { DefaultHeaders } from "../constants/http";
import "/mod";
```

your own modules that are guaranteed to maintain their relative location at runtime.

### 1.5.2 Non-Relative imports

Any other import is considered **non-relative**. Some examples include:

```
import * as $ from "jquery";
import { Component } from "@angular/core";
```

A non-relative import can be resolved relative to **baseUrl** (option in the **tsconfig.json**) which by default corresponds to the project root directory, through path mapping (which we'll cover below). They can also resolve external declarations. Use non-relative paths when importing any of your external dependencies (like **node_modules**).

## 1.6 PATH MAPPING

The hardest part about restructuring a Typescript project is updating all the import paths. While many of our favourite editors can help us, there is a powerful option hiding away in the typescript compiler options that removes this issue entirely.

A typical Typescript file will have imports like this

```
// Imports from node_modules
import { Axios } from 'axios';
// Import from relative paths
import { UserComponent } from './user.component';
import { env } from '../../../env;
```

For any shared code that lives in your app, depending where your current file is located, you may have `'../env'`, `'../../../env` or even `'../../../../env'`

These inconsistent relative paths are ugly and make restructuring our code painful. Move a file that contains these paths and the imports will have to change accordingly.

Typescript enables us to avoid this issue though an option is the **tsconfig.json** (TS configuration file) called **paths** which enables us to setup path mappings that we can use in our imports.

In our **tsconfig.json** file we first need to set a **baseUrl** to tell the compiler where the paths are starting from. In our case this is the **src** folder.

```
"compilerOptions": {
    "baseUrl": "./src",
    "paths": {
      "@app/*": ["app/*"],
      "@environments": ["env/*"],
    }
  }
```

This is telling the Typescript compiler that whenever it sees an import starting with `@app/` that it should look for the code under the `src/app/` folder.

This enables us to update the relative import paths and all modules now share exactly the same import path, this makes restructuring a lot easier.

```
// Imports from node_modules
import { Axios } from 'axios';
// Import from relative paths
import { UserComponent } from '@app/components/user.component';
import { env } from '@environments';
```

## 1.7 NAMESPACES

A namespace is a way that is used for logical grouping of functionalities. It allows us to organize our code in a much cleaner way. A namespace can include interfaces, classes, functions, and variables to support a group of related functionalities.

Unlike JavaScript, namespaces are inbuilt into TypeScript. In JavaScript, the variables declarations go into the global scope. If multiple JavaScript files are used in the same project, then there will be a possibility of confusing new users by overwriting them with a similar name. Hence, the use of TypeScript namespace removes the naming collisions.

### 1.7.1 Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace SomeNameSpaceName {
   export interface ISomeInterfaceName {      }
   export class SomeClassName {      }
}
```

The classes or interfaces which should be accessed outside the namespace should be marked with keyword export.

To access the class or interface in another namespace, the syntax will be

```
namespaceName.functionName;
```

In order to use namespace components at other places, first we need to include the namespace using the triple slash reference syntax.

```
/// <reference path="path to namespace file" />.
```

After including the namespace file using the reference tag, we can access all the functionalities using the namespace.

### 1.7.2 Compiling namespaces

A namespace can span in multiple files and allow to concatenate each file using "**--outFile**" as they were all defined in one place.

```
tsc --outFile File.js File.ts
```

In the above `--outFile` options, File.js is a name and path of the JavaScript file and File.ts is a namespace file name and path.

### 1.7.3 Namespace vs Module

The table below show the differences between namespaces and modules:

| **Namespace** | **Module** |
|---|---|
| Must use the namespace keyword and the export keyword to expose namespace components. | Uses the export keyword to expose module functionalities. |
| Used for logical grouping of functionalities with local scoping. | Used to organize the code in separate files and not pollute the global scope. |
| To use it, it must be included using triple slash reference syntax. | Must import it first in order to use it elsewhere. |
| Must export functions and classes to be able to access it outside the namespace. | All the exports in a module are accessible outside the module. |
| Namespaces cannot declare their dependencies. | Modules can declare their dependencies. |
| No need of module loader. Include the .js file of a namespace using the <script> tag in the HTML page. | Must include the module loader API which was specified at the time of compilation. |

# Module 4: Classes, Interfaces and Enums

## 1.1 OVERVIEW

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.

## 1.2 CLASS DEFINTION

### 1.2.1 Declaring Classes

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

```
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.
- **Constructor**: 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.

Example:

```
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.

### 1.2.2 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:

```
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.

### 1.2.3 Accessing fields 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()
```

## 1.3 GETTERS AND SETTERS

Getters and setters are another way for you to provide access to the properties of an object.

### 1.3.1 Getters

We use the **get** keyword followed by a function expression. The name of the function becomes the name of the property.

```
class Shape {

   _color : string

   get color() {
      return this._color
   }
```

The getter function executes when we read the value of the Property. Note that **we cannot pass an argument to the getter method**. The return value of the getter method becomes the value of the property access expression.

In the example above, `color` is a getter property, while `_color` is a regular Property. The `color` function returns the value of the `_color` property.

We access the `color` getter property just like any other Javascript Property. i.e. using the dot notation

```
console.log(shape.color)
```

Note that although the `color` is a function, we do not invoke it like a function i.e `car.color().` But access it just like a property i.e. `car.color`

### 1.3.2 Setters

To create a setter method, we use the **set** keyword followed by a function expression. The name of the function becomes the name of the property. **A 'set' accessor must have exactly one parameter**.

The following is the syntax of the setter method.

```
class Shape {

   _color : string

   set color(value:string) {
      this._color= value
   }
}
```

In the example above, `color` is a setter property, while `_color` is a regular Property. The `color` function accepts a value and updates the `_color` property.

The **`setter`** method executes when we assign a value to the property. JavaScript invokes the setter method with the value of the right-hand side of the assignment as the argument. If the setter method returns a value, then it is ignored.

```
//Setting color. Runs the setter method
shape.color="red";
```

We assign `the car.color="red"` the setter method executes. Inside the setter method, we assign the value to the property `_color`.

Notice that the `color` accessor property behind the scene uses the `_color` property to store the value. `_color` property is the **backing property of the `color` accessor property**. As a convention, we prepend the backing property with an underscore to indicate that `_color` should not be accessed directly.

## 1.4 ACCESS MODIFIERS

Access modifiers change the visibility of the properties and methods of a class. TypeScript provides three access modifiers:

- private
- protected
- public

Note that TypeScript controls the access logically during compilation time, not at runtime.

### 1.4.1 Private

The `private` modifier limits the visibility to the same-class only. When you add the `private` modifier to a property or method, you can access that property or method within the same class. Any attempt to access private properties or methods outside the class will result in an error at compile time.

The following example shows how to use the `private` modifier to the `firstName`, and `lastName` properties of the `person` class:

```
class Person {
   private firstName: string;
   private lastName: string;
   // ...
}
```

The following attempts to access the `firstName` property outside the class:

```
let person = new Person('John', 'Doe');
console.log(person.firstName); // compile error
```

### 1.4.2 Protected

The `protected` modifier allows properties and methods of a class to be accessible within same class and within subclasses.

When a class (child class) inherits from another class (parent class), it is a subclass of the parent class.

The TypeScript compiler will issue an error if you attempt to access the protected properties or methods from anywhere else.

To add the protected modifier to a property or a method, you use the `protected` keyword. For example:

```
class Person {
   protected id: number;
   private firstName: string;
   private lastName: string;
   // ...
}
```

The `id` property now is protected. It will be accessible within the `Person` class and in any class that inherits from the `Person` class (You'll learn more about inheritance in the next section)

### 1.4.3 Public

The public modifier allows class properties and methods to be accessible from all locations. If you don't specify any access modifier for properties and methods, they will take the public modifier by default.

For example, the `getFullName()` method of the `Person` class has the `public` modifier. The following explicitly adds the public modifier to the `getFullName()` method:

```
class Person {
   // ...
   public getFullName(): string {
       return `${this.firstName} ${this.lastName}`;
   }
```

It has the same effect as if the `public` keyword were omitted.

## 1.5 CLASS INHERITANCE

### 1.5.1 Extending classes

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.

```
class Child extends Parent {

}
```

Example:

```
class Shape {
   Area:number

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

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

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 **Circle** gets an implicit access to its parent class attribute i.e. area.

### 1.5.2 Constructor

To call the constructor of the parent class in the constructor of the child class, you use the `super()` syntax. For example:

```
class Employee extends Person {
   constructor(
       firstName: string,
       lastName: string,
       private jobTitle: string) {

       // call the constructor of the Person class:
       super(firstName, lastName);
   }
}
```

### 1.5.3 Method overriding

Method Overriding is a mechanism by which the child class redefines the superclass's method. A child (derived) class method may or may not use the logic defined in the parent (base) class method. In order to invoke the methods or properties of the base class, we may use the **super** keyword which would help us to invoke that particular method or property of the base class into the child class.

Following is a simple example of method overriding where `eat()` method of Student class overrides the `eat()` method of Person class.

```
class Person{
       name:string

       eat():void{
           console.log(this.name+" eats when hungry.")
       }
   }
```

```
class Student extends Person{
       // variables
       rollnumber:number;

       // constructors
       constructor(rollnumber:number, name1:string){
           super(name1); // calling Parent's constructor
       }
       // overriding super class method
       eat():void{
           console.log(this.name+" eats during break.")
       }
   }
```

Immediate super class methods could be called from sub class using **super** keyword.

```
eat():void{
            super.eat()
            console.log(this.name+" eats during break.")
        }
```

## 1.6 GENERICS

A major part of software engineering is building components that not only have well-defined and consistent APIs, but are also reusable. Components that are capable of working on the data of today as well as the data of tomorrow will give you the most flexible capabilities for building up large software systems.

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

### 1.6.1 Generic functions

The following example is the identity function, the identity function is a function that will return back whatever is passed in. Without using generics, we would have given function parameters specific type or use the top-type any

```
function identity(arg: any): any {
   return arg;
 }
```

While using **any** is certainly generic in that it will cause the function to accept any and all types for the type of **arg**, we actually are losing the information about what that type was when the function returns. If we passed in a number, the only information we have is that any type could be returned.

Instead, we need a way of capturing the type of the argument in such a way that we can also use it to denote what is being returned. Here, we will use a *type variable*, a special kind of variable that works on types rather than values.

```
function identity<Type>(arg: Type): Type {
   return arg;
 }
```

We've now added a type variable **Type** to the identity function. This **Type** allows us to capture the type the user provides (e.g. **number**), so that we can use that information later. Here, we use **Type** again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

We say that this version of the **`identity`** function is generic, as it works over a range of types. Unlike using **`any`**, it's also just as precise (i.e., it doesn't lose any information) as the first **`identity`** function that used numbers for the argument and return type.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
let output = identity<string>("myString");
```

Here we explicitly set `Type` to be `string` as one of the arguments to the function call, denoted using the **`<>`** around the arguments rather than **`()`**.

The second way is also perhaps the most common. Here we use *type argument inference* — that is, we want the compiler to set the value of `Type` for us automatically based on the type of the argument we pass in:

```
let output = identity("myString");
```

Notice that we didn't have to explicitly pass the type in the angle brackets (**`<>`**); the compiler just looked at the value `"myString"`, and set `Type` to its type. While type argument inference can be a helpful tool to keep code shorter and more readable, you may need to explicitly pass in the type arguments in some cases.

### 1.6.2 Generic Types

Generics in type aliases or interfaces allow creating types that are more reusable.

```
interface Wrapped<T> { value: T };
```

The above **`Wrapped`** is a generic interface because we used type variable **`<T>`**. The **`Wrapped`** interface includes the generic field **`value`** of type **`T`**

As mentioned in previous module, you can use interfaces as types. In the same way, generic interface can be used as type, as shown below.

```
interface KeyPair<T, U> {
   key: T;
   value: U;
}

let kv1: KeyPair<number, string> = { key:1, value:"Steve" }; // OK
let kv2: KeyPair<number, number> = { key:1, value:12345 }; // OK
```

### 1.6.3 Generic Classes

TypeScript supports generic classes. The generic type parameter is specified in angle brackets after the name of the class. A generic class can have generic fields (member variables) or methods.

```
class KeyValuePair<T,U>
{
    private key: T;
    private val: U;

    setKeyValue(key: T, val: U): void {
        this.key = key;
        this.val = val;
    }

    display():void {
        console.log(`Key = ${this.key}, val = ${this.val}`);
    }
}
```

In the above example, we created a generic class named **KeyValuePair** with a type variable in the angle brackets **<T, U>** . The **KeyValuePair** class includes two private generic member variables and a generic function **setKeyValue** that takes two input arguments of type **T** and **U**. This allows us to create an object of **KeyValuePair** with any type of key and value.

### 1.6.4 Generic constraints

A generic constraint is simply a way to put some constraints to a type. Suppose we have a generic function like this:

```
function log<T>(arg: T) {}

log("Hitchhiker's Guide to the Galaxy"); //OK
log(42); //OK
log([]); //OK
log({}); //OK
log(null); //OK
log(undefined); //OK
```

You'll notice that `null` and `undefined` are allowed here, but I'm sure most of the time these are invalid inputs. To solve this, we can put a constraint on our generic type to disallow empty value.

```
function log<T extends {}>(arg: T) {}

log("Hitchhiker's Guide to the Galaxy"); //OK
log(42); //OK
log([]); //OK
log({}); //OK
log(null); //Error
log(undefined); //Error
```

In the example above, `T extends {}` means that `T` can be any type that is a subclass of `{}` (an object), in Javascript `string`, `number`, `array` and `object` are all subclasses of `object`, while `undefined` and `null` are not, therefore they are disallowed. This is what generic constraint syntax look like, by specifying the base class of the T type.

**Generics with specific behavior**

In some generic functions or methods, we might invoke a specific method of the argument, but with generic, we can't be sure such property exists. Therefore, we need to further constraint our function to only accept an argument with a specific signature.

```
type Lengthwise = {
   length: number;
 };
 function getLength<T extends Lengthwise>(arg: T): number {
   return arg.length;
 }

 getLength({length:10}) //OK
 getLength({items: []}) //Error
```

## 1.7 INTERFACES

Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface. The TypeScript compiler does not convert interface to JavaScript. It uses interface only for type checking.

An interface is defined with the keyword `interface` and it can include properties and method declarations using a function or an arrow function.

```
interface IEmployee {
   empCode: number;
   empName: string;
   getSalary: (number) => number; // arrow function
   getManagerName(number): string;
}
```

In the above example, the `IEmployee` interface includes two properties `empCode` and `empName`. It also includes a method declaration `getSalaray` using an arrow function which includes one number parameter and a number return type. The `getManagerName` method is declared using a normal function. This means that any object of type `IEmployee` must define the two properties and two methods.

### 1.7.1 Interface as Type

Interface in TypeScript can be used to define a type

```
interface KeyPair {
   key: number;
   value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK
let kv2: KeyPair = { key:1, val:"Steve" };
// Compiler Error: 'val' doesn't exist in type 'KeyPair'
```

In the above example, an interface `KeyPair` includes two properties `key` and `value`. A variable `kv1` is declared as `KeyPair` type. So, it must follow the same structure as `KeyPair`. It means only an object with properties `key` of number type and `value` of string type can be assigned to a variable `kv1`. The TypeScript compiler will show an error if there is any change in the name of the properties or the data type is different than `KeyPair`.

Another variable `kv2` is also declared as `KeyPair` type but the assigned value is `val` instead of `value`, so this will cause an error. In the same way, kv3 assigns a number to the `value` property, so the compiler will show an error. Thus, TypeScript uses an interface to ensure the proper structure of an object.

### 1.7.2 Extending Interfaces

Interfaces can extend one or more interfaces. This makes writing interfaces flexible and reusable.

```
interface IPerson {
   name: string;
   gender: string;
}

interface IEmployee extends IPerson {
   empCode: number;
}

let empObj:IEmployee = {
   empCode:1,
   name:"Bill",
   gender:"Male"
}
```

In the above example, the `IEmployee` interface extends the `IPerson` interface. So, objects of `IEmployee` must include all the properties and methods of the `IPerson` interface otherwise, the compiler will show an error.

### 1.7.3 Implementing Interfaces

Similar to languages like Java and C#, interfaces in TypeScript can be implemented with a Class. The Class implementing the interface needs to strictly conform to the structure of the interface.

```
interface IEmployee {
   empCode: number;
   name: string;
   getSalary:(empCode: number) => number;
}
```

We use **`implements`** keyword to implement interfaces, the implementing class can define extra properties and methods, but at least it must define all the members of an interface.

```
class Employee implements IEmployee {
   empCode: number;
   name: string;

   constructor(code: number, name: string) {
       this.empCode = code;
       this.name = name;
   }

   getSalary(empCode:number):number {
       return 20000;
   }
}

let emp = new Employee(1, "Steve");
```

In the above example, the `IEmployee` interface is implemented in the Employee class using the the implement keyword. The implementing class should strictly define the properties and the function with the same name and data type. If the implementing class does not follow the structure, then the compiler will show an error.

## 1.8 ENUMS

Enums or enumerations are a new data type supported in TypeScript. Most object-oriented languages like Java and C# use enums. This is now available in TypeScript too.

In simple words, enums allow us to declare a set of named constants i.e. a collection of related values that can be numeric or string values.

There are three types of enums:

- Numeric enum
- String enum
- Heterogeneous enum

### 1.8.1 Numeric enum

Numeric enums are number-based enums i.e., they store string values as numbers. Enums can be defined using the keyword enum. Let's say we want to store a set of print media types. The corresponding enum in TypeScript would be:

```
enum PrintMedia {
   Newspaper,
   Newsletter,
   Magazine,
   Book
}
```

In the above example, we have an enum named `PrintMedia`. The enum has four values: `Newspaper`, `Newsletter`, `Magazine`, and `Book`. Here, enum values start from **zero** and increment by 1 for each member. It would be represented as:

```
Newspaper = 0
Newsletter = 1
Magazine = 2
Book = 3
```

Enums are always assigned numeric values when they are stored. The first value always takes the numeric value of 0, while the other values in the enum are incremented by 1.

We also have the option to initialize the first numeric value ourselves. For example, we can write the same enum as:

```
enum PrintMedia {
   Newspaper = 1,
   Newsletter,
   Magazine,
   Book
 }
```

The first member, Newspaper, is initialized with the numeric value 1. The remaining members will be incremented by 1 from the numeric value of the first value. Thus, in the above example, Newsletter would be 2, Magazine would be 3 and Book would be 4.

```
PrintMedia.Newsetter; // returns 2
PrintMedia.Magazine; // returns 3
```

It is not necessary to assign sequential values to Enum members. They can have any values.

### 1.8.2 String enum

String enums are similar to numeric enums, except that the enum values are initialized with string values rather than numeric values.

The benefits of using string enums is that string enums offer better readability. If we were to debug a program, it is easier to read string values rather than numeric values.

Consider the same example of a numeric enum, but represented as a string enum:

```
enum PrintMedia {
   Newspaper = "newspaper",
   Newsletter = "newsletter",
   Magazine = "magazine",
   Book = "book"
}
// Access String Enum
PrintMedia.Newspaper; //returns NEWSPAPER
PrintMedia['Magazine'];//returns MAGAZINE
```

In the above example, we have defined a string enum, `PrintMedia`, with the same values as the numeric enum above, with the difference that these enum values are initialized with string literals. The difference between numeric and string enums is that numeric enum values are auto-incremented, while string enum values need to be individually initialized.

### 1.8.3 Heterogenous enum

Heterogeneous enums are enums that contain both string and numeric values.

```
enum Status {
   Active = 'ACTIVE',
   Deactivate = 1,
   Pending
}
```

### 1.8.4 Reverse Mapping

Enum in TypeScript supports reverse mapping. It means we can access the value of a member and also a member name from its value. Consider the following example.

```
enum PrintMedia {
   Newspaper = 1,
   Newsletter,
   Magazine,
   Book
 }
 PrintMedia.Magazine;    // returns  3
 PrintMedia["Magazine"];// returns  3
 PrintMedia[3];          // returns  Magazine
```

Let's see how TypeScript implements reverse mapping using the following example.

```
enum PrintMedia {
   Newspaper = 1,
   Newsletter,
   Magazine,
   Book
 }
 console.log(PrintMedia)
```

The above example gives the following output in the browser console.

```
{
   '1': 'Newspaper',
   '2': 'Newsletter',
   '3': 'Magazine',
   '4': 'Book',
   Newspaper: 1,
   Newsletter: 2,
   Magazine: 3,
   Book: 4        }
```

You will see that each value of the enum appears twice in the internally stored enum object. We know that num values can be retrieved using the corresponding enum member value. But it is also true that enum members can be retrieved using their values. This is called reverse mapping.

# Module 5: Type manipulation

## 1.1 OVERVIEW

TypeScript's type system is very powerful because it allows expressing types *in terms of other types*.

The simplest form of this idea is generics, we actually have a wide variety of *type operators* available to use. It's also possible to express types in terms of *values* that we already have.

By combining various type operators, we can express complex operations and values in a succinct, maintainable way. In this section we'll cover ways to express a new type in terms of an existing type or value.

## 1.2 TYPEOF OPERATOR

The **`typeof`** operator returns a string indicating the type of the operand's value, JavaScript already has a **`typeof`** operator you can use in an *expression* context:

```
// Prints "string"
console.log(typeof "Hello world");
```

TypeScript adds a `typeof` operator you can use in a *type* context to refer to the *type* of a variable or property:

```
let s = "hello";
let str: typeof s; //type of str is string
```

It's only legal to use **`typeof`** on identifiers (i.e. variable names) or their properties.

## 1.3 UNION TYPES

TypeScript's type system allows you to build new types out of existing ones using a large variety of operators. Now that we know how to write a few types, it's time to start *combining* them in interesting ways.

The first way to combine types you might see is a *union* type. A union type is a type formed from two or more other types, representing values that may be *any one* of those types. We refer to each of these types as the union's *members*.

We define union types using the (**|**) operator

```
type1 | type2 | type3 | .. | typeN
```

Let's write a function that can operate on strings or numbers:

```
function printId(id: number | string) {
   console.log("Your ID is: " + id);
 }
 // OK
 printId(101);
 // OK
 printId("202");
 // Error
 printId({ myID: 22342 });

//Argument of type '{ myID: number; }' is not assignable to parameter
of type 'string | number'.
```

It's common to want to use the same type more than once and refer to it by a single name. you can use a type alias to give a name to any type at all

```
type ID = number | string;

var id: ID = 1
var ids: ID = "my id"
```

An intersection type creates a new type by combining multiple existing types. The new type has all properties of the existing types.

To combine types, you use the (**&**) operator as follows:

```
type typeABC = typeA & typeB & typeC & …;
```

Suppose that you have three interfaces: `BusinessPartner`, `Identity`, and `Contact`.

```
interface Identity {
   id: number;
   name: string;
}

interface Contact {
   email: string;
   phone: string;
}
```

The following defines two intersection types:

```
type Employee = Identity & Contact;
```

The `Employee` type contains all properties of the `Identity` and `Contact` type:

```
let e: Employee = {
   id: 100, //Identity
   name: 'John Doe', //Identity
   email: 'john.doe@example.com',  //Contact
   phone: '(408)-897-5684' //Contact
};
```

## 1.5 TEMPLATE LITERAL TYPES

They have the same syntax as template literal strings in JavaScript, but are used in type positions. When used with concrete literal types, a template literal produces a new string literal type by concatenating the contents.

```
type World = "world";

type Greeting = `hello ${World}`;

var text: Greeting = "hello world" //OK
text = "hi world" //Error
```

When a union is used in the interpolated position, the type is the set of every possible string literal that could be represented by each union member:

```
type Lang = "en" | "fr" | "ar";
```

At the heart of most useful programs, we have to make decisions based on input. JavaScript programs are no different, but given the fact that values can be easily introspected, those decisions are also based on the types of the inputs. *Conditional types* help describe the relation between the types of inputs and outputs.

Conditional types take a form that looks a little like conditional expressions (`condition ? trueExpression : falseExpression`) in JavaScript:

```
type Type = SomeType extends OtherType ? TrueType : FalseType;
```

Example

```
interface Animal {
 }
 interface Dog extends Animal {
 }

 type Example1 = Dog extends Animal ? number : string;

 //Example1 = number
```

From the examples above, conditional types might not immediately seem useful - we can tell ourselves whether or not `Dog extends Animal` and pick `number` or `string`.

The power of conditional types comes from using them with generics, for example conditional return type for a given function

```
function some<T extends string|number>(arg: T) : T extends string ?
string : number {
   //impelement
}

var a = some(4) //a is number
var b = some("hello") //a is string
```

## 1.7 UTILITY TYPES

TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally.

### 1.7.1 Partial<Type>

Constructs a type with all properties of Type set to optional. This utility will return a type that represents all subsets of a given type.

```
interface Person {
  firstName: string;
  lastName: string;
}

var a : Partial<Person> = {} //OK
//firstName and lastName are optional
```

### 1.7.2 Required<Type>

Constructs a type consisting of all properties of **Type** set to required. The opposite of **Partial**.

```
interface Person {
  firstName?: string;
  lastName?: string;
}

var a : Required<Person> = {} //Error
//firstName and lastName are required
```

### 1.7.3 Readonly<Type>

Constructs a type with all properties of Type set to `readonly`, meaning the properties of the constructed type cannot be reassigned.

```
interface Person {
  firstName: string;
  lastName: string;
}
var a : Readonly<Person> = {
  firstName: "Me",
  lastName: "Me"
}
a.firstName = "something" //Error: readonly
```

### 1.7.4 Pick<Type, keys>

Constructs a type by picking the set of properties `Keys` (string literal or union of string literals) from Type.

```
var a : Pick<Person, "firstName"> = {
  firstName: "Me"
} //OK
```

### 1.7.5 Omit<Type, keys>

Constructs a type by picking all properties from Type and then **removing** Keys (opposite of Pick<Type, keys>).

```
var a : Omit<Person, "firstName"> = {
  lastName: "Me"
} //OK
```

### 1.7.6 Parameters<Type>

Constructs a tuple type from the types used in the parameters of a function type Type.

```
function f1(a: number, b: string ): void {
}

type Params = Parameters<typeof f1>
//Params : [a:number, b:number]

var params : Params = [1,"str"]
f1(...params) //OK
```

### 1.7.7 ReturnType<Type>

Constructs a type consisting of the return type of function Type.

```
function f1(a: number, b: string ): void {
}

type Return = ReturnType<typeof f1>
//void
```

### 1.7.8 UpperCase<StringType>

Converts each character in the string to the uppercase version.

```
type Greeting = "Hello, world"
type ShoutyGreeting = Uppercase<Greeting>

var hello: ShoutyGreeting = "HELLO, WORLD" //OK
hello: ShoutyGreeting = "HELLO, world" //Error
```

### 1.7.9 LowerCase<StringType>

Converts each character in the string to the lowercase equivalent.

### 1.7.10 Capitalize<StringType>

Converts the first character in the string to an uppercase equivalent.