TypeScript Data Type - Enum

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:

1. Numeric enum
2. String enum
3. Heterogeneous enum

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:

Example: Numeric Enum

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.

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

enum PrintMedia {

Newspaper = 1,

Newsletter = 5,

Magazine = 5,

Book = 10

}

The enum can be used as a function parameter or return type, as shown below:

Example: Enum as Return Type

enum PrintMedia {

Newspaper = 1,

Newsletter,

Magazine,

Book

}

function getMedia(mediaName: string): PrintMedia {

if ( mediaName === 'Forbes' || mediaName === 'Outlook') {

return PrintMedia.Magazine;

}

}

let mediaType: PrintMedia = getMedia('Forbes'); // returns Magazine

In the above example, we declared an enum PrintMedia. Next, we declare a function getMedia() that takes in an input parameter mediaName of the type string. This function returns an enum PrintMedia. In the function, we check for the type of media. If the media name matches 'Forbes' or 'Outlook', we return enum member PrintMedia.Magazine.

Computed Enums:

Numeric enums can include members with computed numeric value. The value of an enum member can be either a constant or computed. The following enum includes members with computed values.

Example: Computed Enum

enum PrintMedia {

Newspaper = 1,

Newsletter = getPrintMediaCode('newsletter'),

Magazine = Newsletter \* 3,

Book = 10

}

function getPrintMediaCode(mediaName: string): number {

if (mediaName === 'newsletter') {

return 5;

}

}

PrintMedia.Newsetter; // returns 5

PrintMedia.Magazine; // returns 15

When the enum includes computed and constant members, then uninitiated enum members either must come first or must come after other initialized members with numeric constants. The following will give an error.

enum PrintMedia {

Newsletter = getPrintMediaCode('newsletter'),

Newspaper, // Error: Enum member must have initializer

Book,

Magazine = Newsletter \* 3,

}

The above enum can be declared as below.

enum PrintMedia {

Newspaper,

Book,

Newsletter = getPrintMediaCode('newsletter'),

Magazine = Newsletter \* 3

}

// or

enum PrintMedia {

Newsletter = getPrintMediaCode('newsletter'),

Magazine = Newsletter \* 3,

Newspaper = 0,

Book,

}

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:

Example: 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.

Heterogeneous Enum

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

Example: Heterogeneous Enum

enum Status {

Active = 'ACTIVE',

Deactivate = 1,

Pending

}

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.

Example: Reverse Mapping

enum PrintMedia {

Newspaper = 1,

Newsletter,

Magazine,

Book

}

PrintMedia.Magazine; // returns 3

PrintMedia["Magazine"];// returns 3

PrintMedia[3]; // returns Magazine

As you can see in the above example, PrintMedia[3] returns its member name "Magazine". This is because of reverse mapping. 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.

TypeScript can compile the above enum into the following JavaScript function.

Example: Compiled JavaScript of Enum

var PrintMedia;

(function (PrintMedia) {

PrintMedia[PrintMedia["Newspaper"] = 1] = "Newspaper";

PrintMedia[PrintMedia["Newsletter"] = 2] = "Newsletter";

PrintMedia[PrintMedia["Magazine"] = 3] = "Magazine";

PrintMedia[PrintMedia["Book"] = 4] = "Book";

})(PrintMedia || (PrintMedia = {}));

PrintMedia is an object in JavaScript which includes both value and name as properties and that's why enum in TypeScript supports reverse mapping.

So, both the following mappings are true to enums: name -> value, and value -> name.

TypeScript - Arrow Function

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

Syntax:

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

Using fat arrow (=>) we drop the need to use the 'function' keyword. Parameters are passed in the angular brackets <>, and the function expression is enclosed within the curly brackets {}.

Example: Fat Arrow Function

let sum = (x: number, y: number): number => {

return x + y;

}

sum(10, 20); //returns 30

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.

The above arrow function sum will be converted into the following JavaScript code.

var sum = function (x, y) {

return x + y;

}

The following is an arrow function without parameters.

Example: Parameterless Arrow Function

let Print = () => console.log("Hello TypeScript");

Print(); //Output: Hello TypeScript

Furthermore, if the function body consists of only one statement then no need for the curly brackets and the return keyword, as shown below.

let sum = (x: number, y: number) => x + y;

sum(3,4); //returns 30

A class can include an arrow function as a property, as shown below.

Example: Arrow Function in Class

class Employee {

empCode: number;

empName: string;

constructor(code: number, name: string) {

this.empName = name;

this.empCode = code;

}

display = () => console.log(this.empCode +' ' + this.empName)

}

let emp = new Employee(1, 'Ram');

emp.display();

TypeScript Arrow function

ES6 version of TypeScript provides an arrow function which is the **shorthand** syntax for defining the anonymous function, i.e., for function expressions. It omits the function keyword. We can call it fat arrow (because -> is a thin arrow and => is a "**fat**" arrow). It is also called a **Lambda function**. The arrow function has lexical scoping of "**this**" keyword.

The motivation for arrow function is:

* When we don't need to keep typing function.
* It lexically captures the meaning of this keyword.
* It lexically captures the meaning of arguments.

Syntax

We can split the syntax of an Arrow function into three parts:

* **Parameters:** A function may or may not have parameters.
* **The arrow notation/lambda notation** (=>)
* **Statements:** It represents the function's instruction set.

1. (parameter1, parameter2, ..., parameterN) =**>** expression;

If we use the **fat arrow (=>)** notation, there is no need to use the **function** keyword. Parameters are passed in the brackets (), and the function expression is enclosed within the curly brackets {}.

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

1. // ES5: Without arrow function
2. var getResult = function(username, points) {
3. return username + ' scored ' + points + ' points!';
4. };
6. // ES6: With arrow function
7. var getResult = (username: string, points: number): string =**>** {
8. return `${ username } scored ${ points } points!`;
9. }

Arrow function with parameter

The following program is an example of arrow function with parameters.

1. let sum = (a: number, b: number): number =**>** {
2. return a + b;
3. }
4. console.log(sum(20, 30)); //returns 50

In the above example, the **sum** is an arrow function, "**a: number, b: number**" is a parameter type, "**: number**" is the return type, the arrow notation => separates the function parameter and the function body.

After compiling the above TypeScript program, the corresponding JavaScript code is:

1. let sum = (a, b) =**>** {
2. return a + b;
3. };
4. console.log(sum(20, 30)); //returns 50

# TypeScript forEach

The forEach() method is an array method which is used to execute a function on **each item in an array**. We can use it with the JavaScript data types like Arrays, Maps, Sets, etc. It is a useful method for displaying elements in an array.

### Syntax

We can declare the forEach() method as below.

1. array.forEach(callback[, thisObject]);

The forEach() method executes the provided **callback** once for each element present in the array in **ascending order**.

### Parameter Details

**1. callback:** It is a function used to test for each element. The callback function accepts **three arguments**, which are given below.

* **Element value:** It is the current value of the item.
* **Element index:** It is the index of the current element processed in the array.
* **Array:** It is an array which is being iterated in the forEach() method.

#### Note: These three arguments are optional.

**2. thisObject:** It is an object to use as this when executing the callback.

### Return Value

It will return the created array.

### Example with string

1. let apps = ['WhatsApp', 'Instagram', 'Facebook'];
2. let playStore = [];
4. apps.forEach(function(item){
5. playStore.push(item)
6. });
8. console.log(playStore);

The corresponding JavaScript code is:

1. var apps = ['WhatsApp', 'Instagram', 'Facebook'];
2. var playStore = [];
3. apps.forEach(function (item) {
4. playStore.push(item);
5. });
6. console.log(playStore);

**Output:**

### Example with number

1. var num = [5, 10, 15];
2. num.forEach(function (value) {
3. console.log(value);
4. });

**Output:**

### Disadvantage of forEach()

The following are the disadvantages of the use of the forEach() method:

1. It does not provide a way to stop or **break** the forEach() loop.
2. It only **works** with arrays.

TypeScript Access Modifiers

Like other programming languages, Typescript allows us to use access modifiers at the class level. It gives direct access control to the class member. These class members are functions and properties. We can use class members inside its own class, anywhere outside the class, or within its child or derived class.

The access modifier increases the security of the class members and prevents them from invalid use. We can also use it to control the visibility of data members of a class. If the class does not have to be set any access modifier, TypeScript automatically sets public access modifier to all class members.

The TypeScript access modifiers are of three types. These are:

1. Public
2. Private
3. Protected.

Understanding all TypeScript access modifiers

Let us understand the access modifiers with a given table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Access Modifier** | **Accessible within class** | **Accessible in subclass** | **Accessible externally via class instance** |
| Public | Yes | Yes | Yes |
| Protected | Yes | Yes | No |
| Private | Yes | No | No |

Public

In TypeScript by default, all the members (properties and methods) of a class are public. So, there is no need to prefix members with this keyword. We can access this data member anywhere without any restriction.

**Example**

1. class Student {
2. public studCode: number;
3. studName: string;
4. }
6. let stud = new Student();
7. stud.studCode = 101;
8. stud.studName = "Joe Root";
10. console.log(stud.studCode+ " "+stud.studName);

In the above example, **studCode** is public, and **studName** is declared without a modifier, so TypeScript treats them as **public** by default. Since data members are public, they can be accessed outside of the class using an object of the class.

**Output:**

Private

The private access modifier cannot be accessible outside of its containing class. It ensures that the class members are visible only to that class in which it is containing.

**Example**

1. class Student {
2. public studCode: number;
3. private studName: string;
4. constructor(code: number, name: string){
5. this.studCode = code;
6. this.studName = name;
7. }
8. public display() {
9. return (`My unique code: ${this.studCode}, my name: ${this.studName}.`);
10. }
11. }
13. let student: Student = new Student(1, "JoeRoot");
14. console.log(student.display());

In the above example, **studCode** is private, and **studName** is declared without a modifier, so TypeScript treats it as public by default. If we access the private member outside of the class, it will give a compile error.

**Output:**

Protected

A Protected access modifier can be accessed only within the class and its subclass. We cannot access it from the outside of a class in which it is containing.

**Example**

1. class Student {
2. public studCode: number;
3. protected studName: string;
4. constructor(code: number, name: string){
5. this.studCode = code;
6. this.studName = name;
7. }
8. }
9. class Person extends Student {
10. private department: string;
12. constructor(code: number, name: string, department: string) {
13. super(code, name);
14. this.department = department;
15. }
16. public getElevatorPitch() {
17. return (`My unique code: ${this.studCode}, my name: ${this.studName} and I am in ${this.department} Branch.`);
18. }
19. }
20. let joeRoot: Person = new Person(1, "JoeRoot", "CS");
21. console.log(joeRoot.getElevatorPitch());

In the above example, we can't use the name from outside of **Student** class. We can still use it from within an instance method of Person class because **Person** class derives from Student class.

**Output:**

Readonly Modifier

* We can make the properties of the class, type, or interface readonly by using the readonly modifier.
* This modifier needs to be initialized at their declaration time or in the constructor.
* We can also access readonly member from the outside of a class, but its value cannot be changed.

**Example**

1. class Company {
2. readonly country: string = "India";
3. readonly name: string;
5. constructor(contName: string) {
6. this.name = contName;
7. }
8. showDetails() {
9. console.log(this.name + " : " + this.country);
10. }
11. }
13. let comp = new Company("JavaTpoint");
14. comp.showDetails(); // JavaTpoint : India
16. comp.name = "TCS"; //Error, name can be initialized only within constructor