TypeScript Data Type - Array

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

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

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

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

2. Using a generic array type, Array<elementType>.

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

Both methods produce the same output.

Of course, you can always initialize an array like shown below, but you will not get the advantage of TypeScript's type system.

let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false];

Arrays can contain elements of any data type, numbers, strings, or even objects.

Arrays can be declared and initialized separately.

Example: Array Declaration and Initialization

let fruits: Array<string>;

fruits = ['Apple', 'Orange', 'Banana'];

let ids: Array<number>;

ids = [23, 34, 100, 124, 44];

An array in TypeScript can contain elements of different data types using a generic array type syntax, as shown below.

Example: Multi Type Array

let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];

// or

let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];

Accessing Array Elements:

The array elements can be accessed using the index of an element e.g. ArrayName[index]. The array index starts from zero, so the index of the first element is zero, the index of the second element is one and so on.

Example: Access Array Elements

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

fruits[0]; // returns Apple

fruits[1]; // returns Orange

fruits[2]; // returns Banana

fruits[3]; // returns undefined

Use the for loop to access array elements as shown below.

Example: Access Array Elements using Loop

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

for(var index in fruits)

{

console.log(fruits[index]); // output: Apple Orange Banana

}

for(var i = 0; i < fruits.length; i++)

{

console.log(fruits[i]); // output: Apple Orange Banana

}

Array Methods

The following table lists all Array methods which can be used for different purposes.

| Method | Description |
| --- | --- |
| pop() | Removes the last element of the array and return that element |
| push() | Adds new elements to the array and returns the new array length |
| sort() | Sorts all the elements of the array |
| concat() | Joins two arrays and returns the combined result |
| indexOf() | Returns the index of the first match of a value in the array (-1 if not found) |
| copyWithin() | Copies a sequence of elements within the array |
| fill() | Fills the array with a static value from the provided start index to the end index |
| shift() | Removes and returns the first element of the array |
| splice() | Adds or removes elements from the array |
| unshift() | Adds one or more elements to the beginning of the array |
| includes() | Checks whether the array contains a certain element |
| join() | Joins all elements of the array into a string |
| lastIndexOf() | Returns the last index of an element in the array |
| slice() | Extracts a section of the array and returns the new array |
| toString() | Returns a string representation of the array |
| toLocaleString() | Returns a localized string representing the array |

The following example demonstrates some of the array methods.

Example: Array Methods

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

fruits.sort();

console.log(fruits); //output: [ 'Apple', 'Banana', 'Orange' ]

console.log(fruits.pop()); //output: Orange

fruits.push('Papaya');

console.log(fruits); //output: ['Apple', 'Banana', 'Papaya']

fruits = fruits.concat(['Fig', 'Mango']);

console.log(fruits); //output: ['Apple', 'Banana', 'Papaya', 'Fig', 'Mango']

console.log(fruits.indexOf('Papaya'));//output: 2

# 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 = {}));

TypeScript Data Type - Any

TypeScript has type-checking and compile-time checks. However, we do not always have prior knowledge about the type of some variables, especially when there are user-entered values from third party libraries. In such cases, we need a provision that can deal with dynamic content. The Any type comes in handy here.

Example: Any

let something: any = "Hello World!";

something = 23;

something = true;

The above code will compile into the following JavaScript.

var something = "Hello World!";

something = 23;

something = true;

Similarly, you can create an array of type any[] if you are not sure about the types of values that can contain this array.

Example: Any type Array

let arr: any[] = ["John", 212, true];

arr.push("Smith");

console.log(arr); //Output: [ 'John', 212, true, 'Smith' ]

TypeScript - for Loops

Here, you will learn about for loops in TypeScript. There are three types of for loops:

1. for loop
2. for..of loop
3. for..in loop

for Loop

The for loop is used to execute a block of code a given number of times, which is specified by a condition.

Syntax:

for (first expression; second expression; third expression ) {

// statements to be executed repeatedly

}

Here, the first expression is executed before the loop starts. The second expression is the condition for the loop to execute. And the third expression is executed after the execution of every code block.

Example: for Loop

for (let i = 0; i < 3; i++) {

console.log ("Block statement execution no." + i);

}

Result:

Block statement execution no.0

Block statement execution no.1

Block statement execution no.2

In the above example, the first statement let i = 0 declares and initializes a variable. The second conditional statement i < 3 checks whether the value of i is less than 3 or not, and if it is then it exits the loop. The third statement i++ increases the value of i by 1. Thus, the above loop will execute the block three times, until the value of i becomes 3.

for...of Loop

TypeScript includes the **for...of** loop to iterate and access elements of an array, list, or tuple collection. The for...of loop returns elements from a collection e.g. array, list or tuple, and so, there is no need to use the traditional for loop shown above.

Example: for..of Loop

let arr = [10, 20, 30, 40];

for (var val of arr) {

console.log(val); // prints values: 10, 20, 30, 40

}

The for...of loop can also return a character from a string value.

Example: for..of Loop

let str = "Hello World";

for (var char of str) {

console.log(char); // prints chars: H e l l o W o r l d

}

for...in Loop

Another form of the for loop is for...in. This can be used with an array, list, or tuple. The for...in loop iterates through a list or collection and returns an index on each iteration.

Example: for..in Loop

let arr = [10, 20, 30, 40];

for (var index in arr) {

console.log(index); // prints indexes: 0, 1, 2, 3

console.log(arr[index]); // prints elements: 10, 20, 30, 40

}

You can also use let instead of var. The difference is that the variable declared using let will not be accessible out of the for..in loop, as shown below.

Example: for..in Loop

let arr = [10, 20, 30, 40];

for (var index1 in arr) {

console.log(index1); // prints indexes: 0, 1, 2, 3

}

console.log(index1); //OK, prints 3

for (let index2 in arr) {

console.log(index2); // prints elements: 0, 1, 2, 3

}

console.log(index2); //Compiler Error: Cannot find index2

TypeScript - Rest Parameters

In the function chapter, you learned about functions and their parameters. TypeScript introduced rest parameters to accommodate n number of parameters easily.

When the number of parameters that a function will receive is not known or can vary, we can use rest parameters. In JavaScript, this is achieved with the "arguments" variable. However, with TypeScript, we can use the rest parameter denoted by ellipsis ....

We can pass zero or more arguments to the rest parameter. The compiler will create an array of arguments with the rest parameter name provided by us.

Example: Rest Parameters

function Greet(greeting: string, ...names: string[]) {

return greeting + " " + names.join(", ") + "!";

}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"

In the above example, we have a function with two parameters: greeting and names. Here, names is a rest parameter denoted by ellipses .... While calling the function, we first pass "Steve", "Bill" as the rest parameters. These are combined into a string array by joining the elements of the names array. Hence, it returns "Hello Steve, Bill!". During the second function call, we do not pass any arguments as the rest parameters. This is accepted by the compiler and hence the output is "Hello !"

The rest parameters can be used in functions, arrow functions or classes.

Example: Rest Parameters

let Greet = (greeting: string, ...names: string[]) => {

return greeting + " " + names.join(", ") + "!";

}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"

Remember, rest parameters must come last in the function defination, otherwise the TypeScript compiler will show an error. The following is not valid.

Example: Wrong Rest Parameters

function Greet(...names: string[], greeting: string) { // Compiler Error

return greeting + " " + names.join(", ") + "!";

}