Types in TypeScript

TypeScript is a strongly typed language, whereas javascript is not. A variable which has a value defined as a string can be changed to a number without any issues in Javascript. The same is not tolerated in TypeScript. In TypeScript, the type to a variable is defined at the start only and through the execution, it has to maintain the same type any changes to it will lead to a compile-time error during compilation to javascript.

In TypeScript, a data type defines the kind of values a variable can hold, ensuring type safety and enhancing code clarity. Below are the types available in TypeScript:

- **Primitive Types:** Basic types like number, string, boolean, null, undefined, and symbol.
- **Object Types:** Complex structures including arrays, classes, interfaces, and functions.

Primitive Types

Primitive types are the most basic data types in TypeScript. They represent simple, immutable values and are directly assigned.

Type	**Keyword**	**Description**
Number	number	Represents both integer and floating-point numbers.
String	string	Represents textual data.
Boolean	boolean	Represents logical values: true or false.
Null	null	Represents the intentional absence of any object value.
Undefined	undefined	Represents an uninitialized variable.
Symbol	symbol	Represents a unique, immutable value, often used as object keys.
BigInt	bigint	Represents integers with arbitrary precision.

Object Types

Object types are more complex structures that can contain multiple values and functions. They are mutable and can be manipulated after creation.

Type	**Description**
Object	Represents any non-primitive type; however, its use is discouraged in favor of more specific types.
Array	Represents a collection of elements of a specific type.
Tuple	Represents an array with a fixed number of elements of specific types.
Enum	Represents a set of named constants, allowing for a collection of related values.
Function	Represents a callable entity; can define parameter and return types.
Class	Defines a blueprint for creating objects with specific properties and methods.
Interface	Describes the shape of an object, specifying property names and types.

Advanced Types
TypeScript also offers advanced types that provide additional capabilities for complex type definitions:

Type	Description
Union Types	Allows a variable to hold one of several types, providing flexibility in type assignments.
Intersection Types	Combines multiple types into one, requiring a value to satisfy all included types.
Literal Types	Enables exact value types, allowing variables to be assigned specific values only.
Mapped Types	Creates new types by transforming properties of an existing type according to a specified rule.

Best Practices of Using Data types in TypeScript

- **Use let and const Instead of var:** Prefer let and const for block-scoped variables to avoid issues with hoisting and scope leakage.
- **Avoid the any Type:** Refrain from using any as it bypasses type checking; opt for specific types to maintain type safety.
- **Leverage Type Inference**: Allow TypeScript to infer types when possible, reducing redundancy and enhancing code readability.
- **Utilize Utility Types:** Employ built-in utility types like Partial<T> and Readonly<T> to create flexible and readable type definitions.

Let's go through the following important primitive types:

- Number
- String
- Boolean
- Any
- Void

Number

Takes only integers, floats, fractions, etc.

Syntax**:**

```
let a :number = 10;
let marks :number = 150;
let price :number = 10.2;
```

Here are some important methods which can be used on Number types:

toFixed() – it will convert the number to a string and will keep decimal places given to the method.

toString() – this method will convert number to a string.

valueOf() – this method will give back the primitive value of the number.

toPrecision() – this method will format the number to a specified length.

Example : with all String methods

```
let _num :number = 10.345;
_num.toFixed(2); // "10.35"
_num.valueOf(); // 10.345
_num.toString(); // "10.345"
_num.toPrecision(2); //"10"
```

String

String: only string values

Syntax:

```
let str :string = "hello world";
```

Here are some important methods which can be used on String types:

- **split**() – this method will split the string into an array.
- **charAt**() – this method will give the first character for the index given.

- **indexOf**() – this method will give the position of the first occurrence for the value given to it.
- **Replace** () – this method takes 2 strings, first the value to search in the string and if present will replace it will the 2nd one and will give a new string back.
- **Trim** () – this method will remove white spaces from both sides of the string.
- **substr**() – this method will give a part of the string which will depend on the position given as start and end.
- **substring**() – this method will give a part of the string which will depend on the position given as start and end. The character at the end position will be excluded.
- **toUpperCase**() -will convert the string to uppercase
- **toLowerCase**() – will convert the string to lowercase.

Example:

```
let _str:string = "Typescript";

_str.charAt(1); // y
_str.split(""); //["T", "y", "p", "e", "s", "c", "r", "i", "p", "t"]
_str.indexOf("s"); //4 , gives -1 is the value does not exist in the
string.
_str.replace("Type", "Coffee"); //"Coffeescript"
_str.trim(); //"Typescript"
_str.substr(4, _str.length); //"script"
_str.substring(4, 10); //"script"
_str.toUpperCase();//"TYPESCRIPT"
_str.toLowerCase();//"typescript"
```

Boolean

Will accept logical values like true, false, 0, and 1.

Syntax:

```
let bflag :boolean = 1;
let status :boolean = true;
```

Any

Syntax**:**

```
let a :any = 123
a = "hello world"; // changing type will not give any error.
```

Variables declared using **any** type can take the variable as a string, number, array, boolean or void. TypeScript will not throw any compile-time error; this is similar to the variables declared in JavaScript. Make use of any type variable only when you aren't sure about the type of value which will be associated with that variable.

Void

Void type is mostly used as a return type on a function which does not have anything to return.

Syntax:

```
function testfunc():void{
 //code here
}
```

Type Assignment

When creating a variable, there are two main ways TypeScript assigns a type:

- Explicit
- Implicit

In both examples below `firstName` is of type `string`

Explicit Type

Explicit - writing out the type:

```
let firstName: string = "Dylan";
```

Explicit type assignments are easier to read and more intentional. This is also called the Type Annotation

Implicit Type

Implicit - TypeScript will "guess" the type, based on the assigned value:

```
let firstName = "Dylan";
```

Note: Having TypeScript "guess" the type of a value is called **infer**.

Implicit assignment forces TypeScript to **infer** the value.

Implicit type assignments are shorter, faster to type, and often used when developing and testing.

Error In Type Assignment

TypeScript will throw an error if data types do not match.

Example

```
let firstName: string = "Dylan"; // type string
firstName = 33; // attempts to re-assign the value to a different type
```

Implicit type assignment would have made `firstName` less noticeable as a `string`, but both will throw an error:

Example

```
let firstName = "Dylan"; // inferred to type string
firstName = 33; // attempts to re-assign the value to a different type
```

JavaScript will **not** throw an error for mismatched types.

Unable to Infer

TypeScript may not always properly infer what the type of a variable may be. In such cases, it will set the type to `any` which disables type checking.

Example

```
// Implicit any as JSON.parse doesn't know what type of data it returns
so it can be "any" thing...
const json = JSON.parse("55");
// Most expect json to be an object, but it can be a string or a number
like this example
console.log(typeof json);
```

This behavior can be disabled by enabling `noImplicitAny` as an option in a TypeScript's project `tsconfig.json`. That is a JSON config file for customizing how some of TypeScript behaves.

TypeScript Array

An Array in TypeScript is a data type wherein you can store multiple values. Let's learn how to declare and assign values for Array operations in TypeScript. Since TypeScript is a strongly typed language, you have to tell what will be the data type of the values in an array. Otherwise, it will consider it as of type any.

Declare and Initialize an Array

Syntax:

```
let nameofthearray: Array<typehere>
```

Example:

```
let months: Array<string> = ["Jan", "Feb", "March", "April",
"May", "June", "July", "Aug", "Sept", "Oct", "Nov", "Dec"];
//array with all string values.

let years: Array<number> = [2015, 2016, 2017, 2018, 2019];
//array will all numbers
```

```
let month_year: Array<string | number> = ["Jan", 2015, "Feb",
2016]; //array with string and numbers mixed.

//array of all types boolean, string, number, object etc.
let alltypes: Array<any> = [true, false, "Harry", 2000, {"a":
"50", "b": "20"}];
```

Another way of declaring arrays

```
const names: string[] = [];
names.push("Dylan"); // no error
// names.push(3); // Error: Argument of type 'number' is not assignable
to parameter of type 'string'.
```

Readonly

The `readonly` keyword can prevent arrays from being changed.

Example

```
const names: readonly string[] = ["Dylan"];
names.push("Jack"); // Error: Property 'push' does not exist on type
'readonly string[]'.
// try removing the readonly modifier and see if it works?
```

Type Inference

TypeScript can infer the type of an array if it has values.

Example

```
const numbers = [1, 2, 3]; // inferred to type number[]
numbers.push(4); // no error
// comment line below out to see the successful assignment
numbers.push("2"); // Error: Argument of type 'string' is not
assignable to parameter of type 'number'.
let head: number = numbers[0]; // no error
```

Different Ways to access elements from an Array

To get the elements from an array, the values starts from index 0 to the length of the array.

Example:

```
let years: Array<number> = [ 2016, 2017, 2018, 2019];
//array will all numbers
years[0]; // output will be 2016
years[1]; // output will be 2017
```

```
      years[2]; // output will be 2018
      years[3]; // output will be 2019
```

You can also get the elements from an array using TypeScript for loop as shown below:

Using TypeScript for loop

```
      let years: Array<number> = [ 2016, 2017, 2018, 2019];
      for (let i=0;i<=years.length; i++) {
           console.log(years[i]);
      }
Output:
      2016
      2017
      2018
      2019
```

Using for-in loop

```
      let years: Array<number> = [ 2016, 2017, 2018, 2019];
      for (let i in years) {
           console.log(years[i])
      }

Output:
      2016
      2017
      2018
      2019
```

Using for-of loop

```
      let years: Array<number> = [ 2016, 2017, 2018, 2019];
      for (let  i of years) {
           console.log(i)
      }
Output:
      2016
      2017
      2018
      2019
```

Using foreach loop

```
      let years: Array<number> = [ 2016, 2017, 2018, 2019];
      years.forEach(function(yrs, i) {
        console.log(yrs);
      });
Output:
      2016
      2017
```

```
2018
2019
```

TypeScript Array Methods

TypeScript Array object has many properties and methods which help developers to handle arrays easily and efficiently. You can get the value of a property by specifying `arrayname.property` and the output of a method by specifying `arrayname.method()`.

length property

If you want to know the number of elements in an array, you can use the `length` property.

```
let months: Array<string> = ["Jan", "Feb", "March", "April", "May",
"June", "July", "Aug", "Sept", "Oct", "Nov", "Dec"]; //array with
all string values.

console.log(months.length);  // 12
```

Reverse method

You can reverse the order of items in an array using the `reverse()` method.

```
let months: Array<string> = ["Jan", "Feb", "March", "April", "May",
"June", "July", "Aug", "Sept", "Oct", "Nov", "Dec"]; //array with
all string values.

console.log(months.reverse());  //  ["Dec", "Nov", "Oct", "Sept",
"Aug", "July", "June", "May", "April", "March", "Feb", "Jan"]
```

Sort method

You can sort the items in an array using the `sort()` method.

```
let months: Array<string> = ["Jan", "Feb", "March", "April", "May",
"June", "July", "Aug", "Sept", "Oct", "Nov", "Dec"]; //array with
all string values.

console.log(months.sort()); // ["April", "Aug", "Dec", "Feb", "Jan",
"July", "June", "March", "May", "Nov", "Oct", "Sept"]
```

Pop method

You can remove the last item of an array using the `pop()` method.

```
let months: Array<string> = ["Jan", "Feb", "March", "April", "May",
"June", "July", "Aug", "Sept", "Oct", "Nov", "Dec"]; //array with
all string values.

console.log(months.pop()); // Dec
```

Shift method

You can remove the first item of an array using the `shift()` method.

```
let months: Array<string> = ["Jan", "Feb", "March", "April", "May",
"June", "July", "Aug", "Sept", "Oct", "Nov", "Dec"]; //array with
all string values.

console.log(months.shift()); // Jan
```

Push method

You can add value as the last item of the array using the `push()` method.

```
let years: Array<number> = [2015, 2016, 2017, 2018, 2019]; //array
with all numbers

console.log(years.push(2020));

years.forEach(function(yrs, i) {
  console.log(yrs); // 2015 , 2016, 2017, 2018, 2019, 2020
});
```

concat method

You can join two arrays into one array using the `concat()` method.

```
let array1: Array<number> = [10, 20, 30];
let array2: Array<number> = [100, 200, 300];

console.log(array1.concat(array2)); // [10, 20, 30, 100, 200, 300]
```