Skip to main content

TypeScript 3.2 is now available. Download (/#download-links) our latest version today!

Documentation ▾

Basic Types

Introduction

For programs to be useful, we need to be able to work with some of the simplest units of data: numbers, strings, structures, boolean values, and the like. In TypeScript, we support much the same types as you would expect in JavaScript, with a convenient enumeration type thrown in to help things along.

Boolean

The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a `boolean` value.

```
let isDone: boolean = false;
```

Number

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type `number`. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

```
let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;
```

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type `string` to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (`"`) or single quotes (`'`) to surround string data.

```
let color: string = "blue";
color = 'red';
```

You can also use *template strings*, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`` ` ``) character, and embedded expressions are of the form `${ expr }`.

```
let fullName: string = `Bob Bobbington`;
let age: number = 37;
let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;
```

This is equivalent to declaring `sentence` like so:

```
let sentence: string = "Hello, my name is " + fullName + ".\n\n" +
    "I'll be " + (age + 1) + " years old next month.";
```

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by `[]` to denote an array of that element type:

```
let list: number[] = [1, 2, 3];
```

The second way uses a generic array type, `Array<elemType>` :

```
let list: Array<number> = [1, 2, 3];
```

Tuple

Tuple types allow you to express an array where the type of a fixed number of elements is known, but need not be the same. For example, you may want to represent a value as a pair of a `string` and a `number` :

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

When accessing an element with a known index, the correct type is retrieved:

```
console.log(x[0].substr(1)); // OK
console.log(x[1].substr(1)); // Error, 'number' does not have 'substr'
```

When accessing an element outside the set of known indices, a union type is used instead:

```
x[3] = "world"; // OK, 'string' can be assigned to 'string | number'

console.log(x[5].toString()); // OK, 'string' and 'number' both have 'toString'

x[6] = true; // Error, 'boolean' isn't 'string | number'
```

Union types are an advanced topic that we'll cover in a later chapter.

Enum

A helpful addition to the standard set of datatypes from JavaScript is the `enum` . As in languages like C#, an enum is a way of giving more friendly names to sets of numeric values.

```
enum Color {Red, Green, Blue}
let c: Color = Color.Green;
```

By default, enums begin numbering their members starting at `0`. You can change this by manually setting the value of one of its members. For example, we can start the previous example at `1` instead of `0`:

```
enum Color {Red = 1, Green, Blue}
let c: Color = Color.Green;
```

Or, even manually set all the values in the enum:

```
enum Color {Red = 1, Green = 2, Blue = 4}
let c: Color = Color.Green;
```

A handy feature of enums is that you can also go from a numeric value to the name of that value in the enum. For example, if we had the value `2` but weren't sure what that mapped to in the `Color` enum above, we could look up the corresponding name:

```
enum Color {Red = 1, Green, Blue}
let colorName: string = Color[2];

console.log(colorName); // Displays 'Green' as its value is 2 above
```

Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the `any` type:

```
let notSure: any = 4;
notSure = "maybe a string instead";
notSure = false; // okay, definitely a boolean
```

The `any` type is a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. You might expect `Object` to play a similar role, as it does in other languages. But variables of type `Object` only allow you to assign any value to them - you can't call arbitrary methods on them, even ones that actually exist:

```
let notSure: any = 4;
notSure.ifItExists(); // okay, ifItExists might exist at runtime
notSure.toFixed(); // okay, toFixed exists (but the compiler doesn't check)

let prettySure: Object = 4;
prettySure.toFixed(); // Error: Property 'toFixed' doesn't exist on type 'Object'.
```

The `any` type is also handy if you know some part of the type, but perhaps not all of it. For example, you may have an array but the array has a mix of different types:

```
let list: any[] = [1, true, "free"];

list[1] = 100;
```

Void

`void` is a little like the opposite of `any` : the absence of having any type at all. You may commonly see this as the return type of functions that do not return a value:

```
function warnUser(): void {
    console.log("This is my warning message");
}
```

Declaring variables of type `void` is not useful because you can only assign `undefined` or `null` to them:

```
let unusable: void = undefined;
```

Null and Undefined

In TypeScript, both `undefined` and `null` actually have their own types named `undefined` and `null` respectively. Much like `void` , they're not extremely useful on their own:

```
// Not much else we can assign to these variables!
let u: undefined = undefined;
let n: null = null;
```

By default `null` and `undefined` are subtypes of all other types. That means you can assign `null` and `undefined` to something like `number` .

However, when using the `--strictNullChecks` flag, `null` and `undefined` are only assignable to `void` and their respective types. This helps avoid *many* common errors. In cases where you want to pass in either a `string` or `null` or `undefined` , you can use the union type `string | null | undefined` . Once again, more on union types later on.

> As a note: we encourage the use of `--strictNullChecks` when possible, but for the purposes of this handbook, we will assume it is turned off.

Never

The `never` type represents the type of values that never occur. For instance, `never` is the return type for a function expression or an arrow function expression that always throws an exception or one that never returns; Variables also acquire the type `never` when narrowed by any type guards that can never be true.

The `never` type is a subtype of, and assignable to, every type; however, *no* type is a subtype of, or assignable to, `never` (except `never` itself). Even `any` isn't assignable to `never` .

Some examples of functions returning `never` :

```
// Function returning never must have unreachable end point
function error(message: string): never {
    throw new Error(message);
}

// Inferred return type is never
function fail() {
    return error("Something failed");
}

// Function returning never must have unreachable end point
function infiniteLoop(): never {
    while (true) {
    }
}
```

Object

`object` is a type that represents the non-primitive type, i.e. any thing that is not `number`, `string`, `boolean`, `symbol`, `null`, or `undefined`.

With `object` type, APIs like `Object.create` can be better represented. For example:

```
declare function create(o: object | null): void;

create({ prop: 0 }); // OK
create(null); // OK

create(42); // Error
create("string"); // Error
create(false); // Error
create(undefined); // Error
```

Type assertions

Sometimes you'll end up in a situation where you'll know more about a value than TypeScript does. Usually this will happen when you know the type of some entity could be more specific than its current type.

Type assertions are a way to tell the compiler "trust me, I know what I'm doing." A type assertion is like a type cast in other languages, but performs no special checking or restructuring of data. It has no runtime impact, and is used purely by the compiler. TypeScript assumes that you, the programmer, have performed any special checks that you need.

Type assertions have two forms. One is the "angle-bracket" syntax:

```
let someValue: any = "this is a string";

let strLength: number = (<string>someValue).length;
```

And the other is the `as`-syntax:

```
let someValue: any = "this is a string";

let strLength: number = (someValue as string).length;
```

The two samples are equivalent. Using one over the other is mostly a choice of preference; however, when using TypeScript with JSX, only `as` -style assertions are allowed.

A note about `let`

You may've noticed that so far, we've been using the `let` keyword instead of JavaScript's `var` keyword which you might be more familiar with. The `let` keyword is actually a newer JavaScript construct that TypeScript makes available. We'll discuss the details later, but many common problems in JavaScript are alleviated by using `let`, so you should use it instead of `var` whenever possible.