# The TypeScript Handbook

## Basic Types

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 the same types as you would expect in JavaScript, with an extra 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;[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAlgzgIgewHYgFwQEYIaAhkiAXggDNdgYQBuIA)

## Number

As in JavaScript, all numbers in TypeScript are either floating point values or BigIntegers. These floating point numbers get the type number, while BigIntegers get the type bigint. 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;

let big: bigint = 100n;[Try](https://www.typescriptlang.org/play/#code/PTAEAEBcEMCcHMCmkBcoCiBlATABjwFAA2yoAJogMYCWAttEWgHYCutARorKALygBsAbmKkAFogAezNp259cEgGa5cZYSUih21JnACe0jl16hc7AIy5L60gHtKMRqFZG5p2wHYALF5ubt8GgBOpp8lrhMgkA)

## 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';[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAxg9sMCcBcEDOYEEsB2BzCAvBAEQBGwAriMQNwBQA9AxAA4LhiYgIC0mu2RCDqx4CQhADk7ACaSaQA)

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.`;[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAZgrswDkCGBbEAuCBnMAnAlgHYDmEAvBAAYBCA9gEYR331HFi2GUDcAUKJCTEMEQjBT0QuchADMAdj4DsIQmFUBjETgIkZlABIh4tADQQUAT1GoQEfFggASAN6x4yNAF8AdL14AkgDk8BCSzi5CdgDUEACMXhCWIEi4jrTAACaiIAAekCicYAAWPjxAA)

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.";[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAZgrswDkCGBbEAuCBnMAnAlgHYDmEAvBAAYBCA9gEYR331HFi2GUDcAUKJCTEMEQjBT0QuchADMAdj4B6JRAC0GgMYwwGtf3DYQhMMc0icBEuV4QIAIgASIeLQA0EFAE9RqEBHwsBwgAaltoOEQ-UPD7ADoAHUIk+xi7ewBJAHJ4CElgsLsACiF-EIgARgBKNOCvECRcINpgABNREAAPSBROMAALOPtuIA)

## 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];[Try](https://www.typescriptlang.org/play/#code/DYUwLgBMCWDOYC4IDsCuBbARiATgbQF0IBeCPARgBoIAmagZgIG4g)

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

let list: Array<number> = [1, 2, 3];[Try](https://www.typescriptlang.org/play/#code/DYUwLgBMCWDOYC4IEEBOqCGBPAPAOwFcBbAIxFQD4IBeCAbQEYAaCAJhYGYBdAbiA)

## Tuple

Tuple types allow you to express an array with a fixed number of elements whose types are 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

Type 'number' is not assignable to type 'string'.  
Type 'string' is not assignable to type 'number'.Type 'number' is not assignable to type 'string'.  
Type 'string' is not assignable to type 'number'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oFAlACKQDGANgIbSSjmgAuArgA6nV0CeTkOrdoAHqgDaiOtACWAOwDmAGlCSGAWwBGMALoBuPGACSk8XXHlS4gF7VDOfqAC8oIQCIAFpFKk4j+QEYADFtB8AHkAaR1QfUNjUwtQQzjJYgQqYjpSdms7Bz95FzcPRwD8AFFYBCA)

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

// OK

console.log(x[0].substring(1));

console.log(x[1].substring(1));

Property 'substring' does not exist on type 'number'.Property 'substring' does not exist on type 'number'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4BQAbSAF1AA9UBtRI6ASwDsBzAGlHoFcBbAIxgF0A3LlKgAvKAoAiABaR8+OJNYBGAAyDQIUAHkA0rhBgAtCYDG7IiaMGwe3Kbj1EcQgDoFjABSkK614nZuajomT2UASnChe0dnNw9vCmU+f0Dghi8IqKA)

Accessing an element outside the set of known indices fails with an error:

x[3] = "world";

Tuple type '[string, number]' of length '2' has no element at index '3'.Tuple type '[string, number]' of length '2' has no element at index '3'.

console.log(x[5].toString());

Object is possibly 'undefined'.  
Tuple type '[string, number]' of length '2' has no element at index '5'.Object is possibly 'undefined'.  
Tuple type '[string, number]' of length '2' has no element at index '5'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwBYCcBmdBWHNdItAKABtIAXUAD1QG1FroBLAOwHMAaUDgK4BbAEYwAugG4ydUAF5QjAEQALSBQpwlfAIwAGKaBCgA8gGkyIMAFpbAYwHVb1mYxzj5oJQHcEFACZK0mR2cByIcFQAdJpcABR0jPjiUdRwAMqsnPEAlDmSQA)

## 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;[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAwg9gGzgJygbwFBSgJWAEwBosoBxZYUY7AIQTGGIF8MFgAXKAYwC5ZEUUALz8kyAHTlKIANxA)

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;[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAwg9gGzgJygbwFBSgJWAEygF4oBGAGiygHFlhRLsAhBMYSgXwwWABcoAxgC5YiFMVFJkAOlr0QAbiA)

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

enum Color {

Red = 1,

Green = 2,

Blue = 4,

}

let c: Color = Color.Green;[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAwg9gGzgJygbwFBSgJWAEygF4oBGAGiygHFlhRioAmS7AIQTGEYBZKBfDAmAAXKAGMAXLEQpG8JMgB0teiADcQA)

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];

// Displays 'Green'

console.log(colorName);[Try](https://www.typescriptlang.org/play/#code/KYOwrgtgBAwg9gGzgJygbwFBSgJWAEygF4oBGAGiygHFlhRLsAhBMYSgXwwWABcoAxohQA5AIYRgALigBnXsgCWIAObFYw5AG0ATAF0A3BgwB6E1AAii2QAcEYgJ6yoAclr0QLjEJCzEwADokFQAKISRkcUkASgMgA)

## Unknown

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 we may want to intentionally accept all values in our API. In these cases, we want to provide a type that tells the compiler and future readers that this variable could be anything, so we give it the unknown type.

let notSure: unknown = 4;

notSure = "maybe a string instead";

// OK, definitely a boolean

notSure = false;[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAdg9mDKBXATiAXBRUDWsDuUEAvBACwDcAULAiiMRAEQC2AhgJ4BG9rEAzmGQBLKAHMIIgSFYATRlUoB6RRADyAaQA0EGSABmIoWBDB2EXpxgxQrKNThJUDPa2B8Q5IA)

If you have a variable with an unknown type, you can narrow it to something more specific by doing typeof checks, comparison checks, or more advanced type guards that will be discussed in a later chapter:

declare const maybe: unknown;

// 'maybe' could be a string, object, boolean, undefined, or other types

const aNumber: number = maybe;

Type 'unknown' is not assignable to type 'number'.Type 'unknown' is not assignable to type 'number'.

if (maybe === true) {

// TypeScript knows that maybe is a boolean now

const aBoolean: boolean = maybe;

// So, it cannot be a string

const aString: string = maybe;

Type 'boolean' is not assignable to type 'string'.Type 'boolean' is not assignable to type 'string'.}

if (typeof maybe === "string") {

// TypeScript knows that maybe is a string

const aString: string = maybe;

// So, it cannot be a boolean

const aBoolean: boolean = maybe;

Type 'string' is not assignable to type 'boolean'.Type 'string' is not assignable to type 'boolean'.}[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1vVnnsAoAE0gGMAbAQ2klDLgDtEAXUAWyoE8AjSVAK6MA1ozgB3RgG5CIUAHJOvSPPpwBFYqD6gqoVtACWjAOYAaUHB4Arciws84cCpCqMLQ0gDNjkYhYRLFgALGFAWLgAHSERCBmY2KgA5AXY+aFRGVPTQAF4Obj4ZQkMvUAAKJR1cmvDoAUgASlAAb0JQUDkAFSjIAGUyI0i2UQlEcOCqNiq6Q3G9R2dXRlAxcXa1BN0AIScXN1RF-ZX8mZkOuT64C0M2MjcxNh09A2MTDfjWXT6WI1NUV6mPIFZQyAC+hBKZXKEWicDKMzytQARICTMjmm0LmAetEBkMRmtxiEpiCdHNdPpfm8PkwvlQfn8TADqUDToVIOdOmArjc7g84E86As9staVsqLslgdtKK3MCzoQwUA)

## Any

In some situations, not all type information is available or its declaration would take an inappropriate amount of effort. These may occur for values from code that has been written without TypeScript or a 3rd party library. In these cases, we might want to opt-out of type checking. To do so, we label these values with the any type:

declare function getValue(key: string): any;

// OK, return value of 'getValue' is not checked

const str: string = getValue("myString");[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwHMQMA1KCZEACgGsQBPALngGcMYtUCBKZqVegG4AUAHpR8APIBpADTw4GZDHwA3cpXg5E8AORFSGkLvhYW8VDgzwwAC3B1gwsHjat2zNhy7wAvIWIyCmoAIgBbegBldk4CEO5BIA)

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.

Unlike unknown, variables of type any allow you to access arbitrary properties, even ones that don’t exist. These properties include functions and TypeScript will not check their existence or type:

let looselyTyped: any = 4;

// OK, ifItExists might exist at runtime

looselyTyped.ifItExists();

// OK, toFixed exists (but the compiler doesn't check)

looselyTyped.toFixed();

let strictlyTyped: unknown = 4;

strictlyTyped.toFixed();

Object is of type 'unknown'.Object is of type 'unknown'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYDsBGAUADaQAuoBccikBAngCo0AOkAJqgIYB2NoAvKABYA3HhCgA8gGkANKACWAMwCSxAKIAPOYmKJQAWzkBzABalIm7aHaloAV07E5eyIQpVaDZiwB0ilRq0dAAoAShExKVliOAAxOXVWUHNA3SCAI1tSYmNIUABjOD1GOSJoUBY4SEROAHJSPJy8gGsQ10pqeiZWb2i4hJZQkUISUG1oOTziDy62UHsmzjgAd04+QRExianOrx7Y+NZBoA)

The any will continue to propagate through your objects:

let looselyTyped: any = {};

let d = looselyTyped.a.b.c.d;

// ^ = let d: any[Try](https://www.typescriptlang.org/play/#code/DYUwLgBMD20M4mATwCpIA4gCYC4IEMA7JCAXggG8BfAbgChRIsypYFk1MsA6fbgI24BjblnoB6cRAgA9APxA)

After all, remember that all the convenience of any comes at the cost of losing type safety. Type safety is one of the main motivations for using TypeScript and you should try to avoid using any when not necessary.

## 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");

}[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAdwIYCcwFUDOBTdACgEoAuRANzhgBNEBvAKEUQgRzgBs8A6TuAOaEARABUAFjByIpiALYBPFBjAwwA+Xhw5UAvMOIBuRgF8gA)

Declaring variables of type void is not useful because you can only assign null (only if --strictNullChecks is not specified, see next section) or undefined to them:

let unusable: void = undefined;

// OK if `--strictNullChecks` is not given

unusable = null;[Try](https://www.typescriptlang.org/play/#code/PTAEAEGcBcCcEsDG0BcoBmBDANpApgFDZ7SgCuAdmZJgEbFoBuA9vACagC85Fbe68CnjYBuAiFAB5ANKh46UAAMAtMpgJkAOTLZsAYQAWeRAGtIiuZFAVmpAObxGeCgUrU6xLtZ3YRQA)

## Null and Undefined

In TypeScript, both undefined and null actually have their 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;[Try](https://www.typescriptlang.org/play/#code/PTAEDkHsBdQWwK4GMAWoCmAbAzu0B3PJAQwDtRjtsBLAc3OklGhXV1ADdiAna4gI0xsAhACghsBAC5QCUgBN0AM2ql080AF5ZC5avUBucelikZpBJkxbQFqwaA)

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 unknown, any and their respective types (the one exception being that undefined is also assignable to void). 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.

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

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 not have a reachable 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 not have a reachable end point

function infiniteLoop(): never {

while (true) {}

}[Try](https://www.typescriptlang.org/play/#code/PTAEDEFcDsGMBcCWB7aoBOBTel3UdAOajSYBum6oAtpAM7wnKMAWAhhaGxpm7OwCMANplCZoAE1AAHZAXgAoAGYwEKNJXTJ0ACmqY6dNoUwAuUA3QFCASnOkKVAN4LQoeCy0B3Eph8BRdC1dfUNjTBsAbgUAXwUFEFAASWglTUwpLBw8dwBPaVFEOl9HZVUkVFAlNkQhHRtQFzcs3A0g7R0AIgBlZH0PayqakQlOqNj4xKg4CrQWvEGHShp6RmhmUHZObiw+QRExSRk5aEUVGfVQAiUCRHhMABlkZGl6+3JlptAvFlrRHXg6EgEUacRiQA)

## Object

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

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

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

// OK

create({ prop: 0 });

create(null);

create(42);

Argument of type '42' is not assignable to parameter of type 'object | null'.Argument of type '42' is not assignable to parameter of type 'object | null'.create("string");

Argument of type '"string"' is not assignable to parameter of type 'object | null'.Argument of type '"string"' is not assignable to parameter of type 'object | null'.create(false);

Argument of type 'false' is not assignable to parameter of type 'object | null'.Argument of type 'false' is not assignable to parameter of type 'object | null'.create(undefined);

Argument of type 'undefined' is not assignable to parameter of type 'object | null'.Argument of type 'undefined' is not assignable to parameter of type 'object | null'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAUACaQDGANgIbSSgBmArgHbEAuAlnA6MVec5ABRxUcAEYArEs1AAfUAzqlSASlQA3OKwIBuPHhCgA8gGk83SLwEBvUAAd4N1AAZQAXyU6zF-vMXvdnvn4sND8AgQAiRGZoVgYAc3DQnkCaclJESCTzQMYiGljIAncgA)

Generally, you won’t need to use this.

## 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 it 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 as-syntax:

let someValue: unknown = "this is a string";

let strLength: number = (someValue as string).length;[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAzg9gtiAagQ2AVxALgmgdga1xgHdcIBeCAIjAAsBLKCRiZaMAJ3twHMqBuAFCDQkKJwAyIXnWy40cAEYgOFCAApYCFOhCsm4rrwCUAOlAza-IA)

The other version is the “angle-bracket” syntax:

let someValue: unknown = "this is a string";

let strLength: number = (<string>someValue).length;[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAzg9gtiAagQ2AVxALgmgdga1xgHdcIBeCAIjAAsBLKCRiZaMAJ3twHMqBuAFCDQkKJwAyIXnWy40cAEYgOFCAAoAPOK68AfLAQp0IAJQA6UDNr8gA)

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 have 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. You can read in the Handbook Reference on [Variable Declarations](https://www.typescriptlang.org/docs/handbook/variable-declarations.html) more about how let and const fix a lot of the problems with var.

## About Number, String, Boolean, Symbol and Object

It can be tempting to think that the types Number, String, Boolean, Symbol, or Object are the same as the lowercase versions recommended above. These types do not refer to the language primitives however, and almost never should be used as a type.

function reverse(s: String): String {

return s.split("").reverse().join("");

}

reverse("hello world");[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4BQAzAVwDsBjAFwEs5jRpIA3GRSAChVAGVzpLiBzAJSpuvAaADeuUHUjlC0WogB0iAA4AbSuVYAiXYOX0mSNoYBWcPnoMBuXAF9cuY8za6AFpA0a4oAO4IGgAmdkA)

Instead, use the types number, string, boolean, object and symbol.

function reverse(s: string): string {

return s.split("").reverse().join("");

}

reverse("hello world");