Everyday Types

In this chapter, we'll cover some of the most common types of values you'll find in JavaScript code, and explain the corresponding ways to describe those types in TypeScript. This isn't an exhaustive list, and future chapters will describe more ways to name and use other types.

Types can also appear in many more *places* than just type annotations. As we learn about the types themselves, we'll also learn about the places where we can refer to these types to form new constructs.

We'll start by reviewing the most basic and common types you might encounter when writing JavaScript or TypeScript code. These will later form the core building blocks of more complex types.

The primitives: `string`, `number`, and `boolean`

JavaScript has three very commonly used primitives: `string`, `number`, and `boolean`. Each has a corresponding type in TypeScript. As you might expect, these are the same names you'd see if you used the JavaScript `typeof` operator on a value of those types:

- `string` represents string values like `"Hello, world"`
- `number` is for numbers like `42`. JavaScript does not have a special runtime value for integers, so there's no equivalent to `int` or `float` - everything is simply `number`
- `boolean` is for the two values `true` and `false`

> The type names `String`, `Number`, and `Boolean` (starting with capital letters) are legal, but refer to some special built-in types that will very rarely appear in your code. *Always* use `string`, `number`, or `boolean` for types.

To specify the type of an array like `[1, 2, 3]`, you can use the syntax `number[]`; this syntax works for any type (e.g. `string[]` is an array of strings, and so on). You may also see this written as `Array<number>`, which means the same thing. We'll learn more about the syntax `T<U>` when we cover *generics*.

> Note that `[number]` is a different thing; refer to the section on Tuples.

any

TypeScript also has a special type, `any`, that you can use whenever you don't want a particular value to cause typechecking errors.

When a value is of type `any`, you can access any properties of it (which will in turn be of type `any`), call it like a function, assign it to (or from) a value of any type, or pretty much anything else that's syntactically legal:

```
let obj: any = { x: 0 };
// None of the following lines of code will throw compiler errors.
// Using `any` disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.foo();
obj();
obj.bar = 100;
obj = "hello";
const n: number = obj;
```

The `any` type is useful when you don't want to write out a long type just to convince TypeScript that a particular line of code is okay.

noImplicitAny

When you don't specify a type, and TypeScript can't infer it from context, the compiler will typically default to `any`.

You usually want to avoid this, though, because `any` isn't type-checked. Use the

Type Annotations on Variables

When you declare a variable using `const`, `var`, or `let`, you can optionally add a type annotation to explicitly specify the type of the variable:

```
let myName: string = "Alice";
```

> TypeScript doesn't use "types on the left"-style declarations like `int x = 0;` Type annotations will always go *after* the thing being typed.

In most cases, though, this isn't needed. Wherever possible, TypeScript tries to automatically *infer* the types in your code. For example, the type of a variable is inferred based on the type of its initializer:

```
// No type annotation needed -- 'myName' inferred as type 'string'
let myName = "Alice";
```

For the most part you don't need to explicitly learn the rules of inference. If you're starting out, try using fewer type annotations than you think - you might be surprised how few you need for TypeScript to fully understand what's going on.

Functions

Functions are the primary means of passing data around in JavaScript. TypeScript allows you to specify the types of both the input and output values of functions.

Parameter Type Annotations

When you declare a function, you can add type annotations after each parameter to declare what types of parameters the function accepts. Parameter type annotations go after the parameter name:

When a parameter has a type annotation, arguments to that function will be checked:

```
// Would be a runtime error if executed!
greet(42);
```

```
Argument of type 'number' is not assignable to parameter of type
'string'.
```

> Even if you don't have type annotations on your parameters, TypeScript will still check that you passed the right number of arguments.

Return Type Annotations

You can also add return type annotations. Return type annotations appear after the parameter list:

```
function getFavoriteNumber(): number {
  return 26;
}
```

Much like variable type annotations, you usually don't need a return type annotation because TypeScript will infer the function's return type based on its `return` statements. The type annotation in the above example doesn't change anything. Some codebases will explicitly specify a return type for documentation purposes, to prevent accidental changes, or just for personal preference.

Anonymous Functions

Anonymous functions are a little bit different from function declarations. When a function appears in a place where TypeScript can determine how it's going to be called, the parameters of that function are automatically given types.

```
Property 'toUppercase' does not exist on type 'string'. Did you mean
'toUpperCase'?
```

```
// Contextual typing for function
names.forEach(function (s) {
  console.log(s.toUppercase());

Property 'toUppercase' does not exist on type 'string'. Did you
mean 'toUpperCase'?

});

// Contextual typing also applies to arrow functions
names.forEach((s) => {
  console.log(s.toUppercase());

Property 'toUppercase' does not exist on type 'string'. Did you
mean 'toUpperCase'?

});
```

Even though the parameter `s` didn't have a type annotation, TypeScript used the types of the `forEach` function, along with the inferred type of the array, to determine the type `s` will have.

This process is called *contextual typing* because the *context* that the function occurred within informs what type it should have.

Similar to the inference rules, you don't need to explicitly learn how this happens, but understanding that it *does* happen can help you notice when type annotations aren't needed. Later, we'll see more examples of how the context that a value occurs in can affect its type.

Object Types

Apart from primitives, the most common sort of type you'll encounter is an *object type*. This refers to any JavaScript value with properties, which is almost all of them! To define an object type, we simply list its properties and their types.

For example, here's a function that takes a point-like object:

```
}
printCoord({ x: 3, y: 7 });
```

Here, we annotated the parameter with a type with two properties - `x` and `y` - which are both of type `number`. You can use `,` or `;` to separate the properties, and the last separator is optional either way.

The type part of each property is also optional. If you don't specify a type, it will be assumed to be `any`.

Optional Properties

Object types can also specify that some or all of their properties are *optional*. To do this, add a `?` after the property name:

```
function printName(obj: { first: string; last?: string }) {
  // ...
}
// Both OK
printName({ first: "Bob" });
printName({ first: "Alice", last: "Alisson" });
```

In JavaScript, if you access a property that doesn't exist, you'll get the value `undefined` rather than a runtime error. Because of this, when you *read* from an optional property, you'll have to check for `undefined` before using it.

```
function printName(obj: { first: string; last?: string }) {
  // Error - might crash if 'obj.last' wasn't provided!
  console.log(obj.last.toUpperCase());
```

```
Object is possibly 'undefined'.
```

```
  if (obj.last !== undefined) {
    // OK
    console.log(obj.last.toUpperCase());
  }
```

Union Types

TypeScript's type system allows you to build new types out of existing ones using a large variety of operators. Now that we know how to write a few types, it's time to start *combining* them in interesting ways.

Defining a Union Type

The first way to combine types you might see is a *union* type. A union type is a type formed from two or more other types, representing values that may be *any one* of those types. We refer to each of these types as the union's *members*.

Let's write a function that can operate on strings or numbers:

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
// OK
printId(101);
// OK
printId("202");
// Error
printId({ myID: 22342 });
```

```
Argument of type '{ myID: number; }' is not assignable to parameter
of type 'string | number'.
```

Working with Union Types

It's easy to *provide* a value matching a union type - simply provide a type matching any of the union's members. If you *have* a value of a union type, how do you work with it?

TypeScript will only allow an operation if it is valid for *every* member of the union. For example, if you have the union `string | number`, you can't use methods that are only available on `string`:

```
    Property 'toUpperCase' does not exist on type 'number'.
}
```

The solution is to *narrow* the union with code, the same as you would in JavaScript without type annotations. *Narrowing* occurs when TypeScript can deduce a more specific type for a value based on the structure of the code.

For example, TypeScript knows that only a `string` value will have a `typeof` value `"string"`:

```
function printId(id: number | string) {
  if (typeof id === "string") {
    // In this branch, id is of type 'string'
    console.log(id.toUpperCase());
  } else {
    // Here, id is of type 'number'
    console.log(id);
  }
}
```

Another example is to use a function like `Array.isArray`:

```
function welcomePeople(x: string[] | string) {
  if (Array.isArray(x)) {
    // Here: 'x' is 'string[]'
    console.log("Hello, " + x.join(" and "));
  } else {
    // Here: 'x' is 'string'
    console.log("Welcome lone traveler " + x);
  }
}
```

Notice that in the `else` branch, we don't need to do anything special - if `x` wasn't a `string[]` then it must have been a `string`.

```ts
// Return type is inferred as number[] | string
function getFirstThree(x: number[] | string) {
  return x.slice(0, 3);
}
```

> It might be confusing that a *union* of types appears to have the *intersection* of those types' properties. This is not an accident - the name *union* comes from type theory. The *union* `number | string` is composed by taking the union *of the values* from each type. Notice that given two sets with corresponding facts about each set, only the *intersection* of those facts applies to the *union* of the sets themselves. For example, if we had a room of tall people wearing hats, and another room of Spanish speakers wearing hats, after combining those rooms, the only thing we know about *every* person is that they must be wearing a hat.

Type Aliases

We've been using object types and union types by writing them directly in type annotations. This is convenient, but it's common to want to use the same type more than once and refer to it by a single name.

A *type alias* is exactly that - a *name* for any *type*. The syntax for a type alias is:

```ts
type Point = {
  x: number;
  y: number;
};

// Exactly the same as the earlier example
function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

```
type ID = number | string;
```

Note that aliases are *only* aliases - you cannot use type aliases to create different/distinct "versions" of the same type. When you use the alias, it's exactly as if you had written the aliased type. In other words, this code might *look* illegal, but is OK according to TypeScript because both types are aliases for the same type:

```
type UserInputSanitizedString = string;

function sanitizeInput(str: string): UserInputSanitizedString {
  return sanitize(str);
}

// Create a sanitized input
let userInput = sanitizeInput(getInput());

// Can still be re-assigned with a string though
userInput = "new input";
```

Interfaces

An *interface declaration* is another way to name an object type:

```
interface Point {
  x: number;
  y: number;
}

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

value we passed to `printCoord` - it only cares that it has the expected properties. Being concerned only with the structure and capabilities of types is why we call TypeScript a *structurally typed* type system.

Differences Between Type Aliases and Interfaces

Type aliases and interfaces are very similar, and in many cases you can choose between them freely. Almost all features of an `interface` are available in `type`, the key distinction is that a type cannot be re-opened to add new properties vs an interface which is always extendable.

Interface	**Type**
Extending an interface	Extending a type via intersections

`interface Animal {`
`  name: string`
`}`
 	
`interface Bear extends Animal {`
`  honey: boolean`
`}`
 	
`const bear = getBear()`
`bear.name`
`bear.honey`
	`type Animal = {`
	`  name: string`
	`}`
	
	`type Bear = Animal & {`
	`  honey: boolean`
	`}`
	
	`const bear = getBear();`
	`bear.name;`
	`bear.honey;`

Adding new fields to an existing interface	A type cannot be changed after being crea

`interface Window {`
`  title: string`
`}`
	`type Window = {`
	`  title: string`
	`}`

```
window.ts.transpileModule(src, {});
```

You'll learn more about these concepts in later chapters, so don't worry if you don't understand all of these right away.

- Prior to TypeScript version 4.2, type alias names *may* appear in error messages, sometimes in place of the equivalent anonymous type (which may or may not be desirable). Interfaces will always be named in error messages.
- Type aliases may not participate in declaration merging, but interfaces can.
- Interfaces may only be used to declare the shapes of objects, not rename primitives.
- Interface names will *always* appear in their original form in error messages, but *only* when they are used by name.

For the most part, you can choose based on personal preference, and TypeScript will tell you if it needs something to be the other kind of declaration. If you would like a heuristic, use `interface` until you need to use features from `type`.

Type Assertions

Sometimes you will have information about the type of a value that TypeScript can't know about.

For example, if you're using `document.getElementById`, TypeScript only knows that this will return *some* kind of `HTMLElement`, but you might know that your page will always have an `HTMLCanvasElement` with a given ID.

In this situation, you can use a *type assertion* to specify a more specific type:

```
const myCanvas = document.getElementById("main_canvas") as HTMLCanvas
```

which is equivalent:

```
const myCanvas = <HTMLCanvasElement>document.getElementById("main_can
```

> Reminder: Because type assertions are removed at compile-time, there is no runtime checking associated with a type assertion. There won't be an exception or `null` generated if the type assertion is wrong.

TypeScript only allows type assertions which convert to a *more specific* or *less specific* version of a type. This rule prevents "impossible" coercions like:

```
const x = "hello" as number;
```

```
Conversion of type 'string' to type 'number' may be a mistake because
neither type sufficiently overlaps with the other. If this was
intentional, convert the expression to 'unknown' first.
```

Sometimes this rule can be too conservative and will disallow more complex coercions that might be valid. If this happens, you can use two assertions, first to `any` (or `unknown`, which we'll introduce later), then to the desired type:

```
const a = (expr as any) as T;
```

Literal Types

In addition to the general types `string` and `number`, we can refer to *specific* strings and numbers in type positions.

One way to think about this is to consider how JavaScript comes with different ways to declare a variable. Both `var` and `let` allow for changing what is held inside the variable, and `const` does not. This is reflected in how TypeScript creates types for literals.

```
// Because `changingString` can represent any possible string, that
// is how TypeScript describes it in the type system
changingString;
```

```
let changingString: string
```

```
const constantString = "Hello World";
// Because `constantString` can only represent 1 possible string, it
// has a literal type representation
constantString;
```

```
const constantString: "Hello World"
```

By themselves, literal types aren't very valuable:

```
let x: "hello" = "hello";
// OK
x = "hello";
// ...
x = "howdy";
```

```
Type '"howdy"' is not assignable to type '"hello"'.
```

It's not much use to have a variable that can only have one value!

But by *combining* literals into unions, you can express a much more useful concept - for example, functions that only accept a certain set of known values:

```
function printText(s: string, alignment: "left" | "right" | "center")
  // ...
}
printText("Hello, world", "left");
printText("G'day, mate", "centre");
```

```
Argument of type '"centre"' is not assignable to parameter of type
```

Numeric literal types work the same way:

```
function compare(a: string, b: string): -1 | 0 | 1 {
  return a === b ? 0 : a > b ? 1 : -1;
}
```

Of course, you can combine these with non-literal types:

```
interface Options {
  width: number;
}
function configure(x: Options | "auto") {
  // ...
}
configure({ width: 100 });
configure("auto");
configure("automatic");
```

```
Argument of type '"automatic"' is not assignable to parameter of type
'Options | "auto"'.
```

There's one more kind of literal type: boolean literals. There are only two boolean literal types, and as you might guess, they are the types `true` and `false`. The type `boolean` itself is actually just an alias for the union `true | false`.

Literal Inference

When you initialize a variable with an object, TypeScript assumes that the properties of that object might change values later. For example, if you wrote code like this:

```
const obj = { counter: 0 };
if (someCondition) {
  obj.counter = 1;
}
```

`number`, not `0`, because types are used to determine both *reading* and *writing* behavior.

The same applies to strings:

```
const req = { url: "https://example.com", method: "GET" };
handleRequest(req.url, req.method);
```

```
Argument of type 'string' is not assignable to parameter of type
'"GET" | "POST"'.
```

In the above example `req.method` is inferred to be `string`, not `"GET"`. Because code can be evaluated between the creation of `req` and the call of `handleRequest` which could assign a new string like `"GUESS"` to `req.method`, TypeScript considers this code to have an error.

There are two ways to work around this.

1. You can change the inference by adding a type assertion in either location:

   ```
   // Change 1:
   const req = { url: "https://example.com", method: "GET" as "GET"
   // Change 2
   handleRequest(req.url, req.method as "GET");
   ```

   Change 1 means "I intend for `req.method` to always have the *literal type* `"GET"`", preventing the possible assignment of `"GUESS"` to that field after. Change 2 means "I know for other reasons that `req.method` has the value `"GET"`".

2. You can use `as const` to convert the entire object to be type literals:

   ```
   const req = { url: "https://example.com", method: "GET" } as con
   handleRequest(req.url, req.method);
   ```

`string` or `number`.

`null` and `undefined`

JavaScript has two primitive values used to signal absent or uninitialized value: `null` and `undefined`.

TypeScript has two corresponding *types* by the same names. How these types behave depends on whether you have the `strictNullChecks` option on.

`strictNullChecks` off

With `strictNullChecks` *off*, values that might be `null` or `undefined` can still be accessed normally, and the values `null` and `undefined` can be assigned to a property of any type. This is similar to how languages without null checks (e.g. C#, Java) behave. The lack of checking for these values tends to be a major source of bugs; we always recommend people turn `strictNullChecks` on if it's practical to do so in their codebase.

`strictNullChecks` on

With `strictNullChecks` *on*, when a value is `null` or `undefined`, you will need to test for those values before using methods or properties on that value. Just like checking for `undefined` before using an optional property, we can use *narrowing* to check for values that might be `null`:

```
function doSomething(x: string | null) {
  if (x === null) {
    // do nothing
  } else {
    console.log("Hello, " + x.toUpperCase());
  }
}
```

Non-null Assertion Operator (Postfix `!`)

```ts
function liveDangerously(x?: number | null) {
  // No error
  console.log(x!.toFixed());
}
```

Just like other type assertions, this doesn't change the runtime behavior of your code, so it's important to only use `!` when you know that the value *can't* be `null` or `undefined`.

Enums

Enums are a feature added to JavaScript by TypeScript which allows for describing a value which could be one of a set of possible named constants. Unlike most TypeScript features, this is *not* a type-level addition to JavaScript but something added to the language and runtime. Because of this, it's a feature which you should know exists, but maybe hold off on using unless you are sure. You can read more about enums in the Enum reference page.

Less Common Primitives

It's worth mentioning the rest of the primitives in JavaScript which are represented in the type system. Though we will not go into depth here.

`bigint`

From ES2020 onwards, there is a primitive in JavaScript used for very large integers, `BigInt`:

```ts
// Creating a bigint via the BigInt function
const oneHundred: bigint = BigInt(100);

// Creating a BigInt via the literal syntax
const anotherHundred: bigint = 100n;
```

There is a primitive in JavaScript used to create a globally unique reference via the function `Symbol()`:

```
const firstName = Symbol("name");
const secondName = Symbol("name");

if (firstName === secondName) {
This condition will always return 'false' since the types 'typeof firstName' and 'typeof secondName' have no overlap.
  // Can't ever happen
}
```

Previous

The Basics

Step one in learning TypeScript: The basic types.

Next

Narrowing

Understand how TypeScript uses JavaScript knowledge to reduce the amount of type syntax in your projects.

Community

Using TypeScript

Made with ♥ in Redmond,
Boston, SF & Dublin