

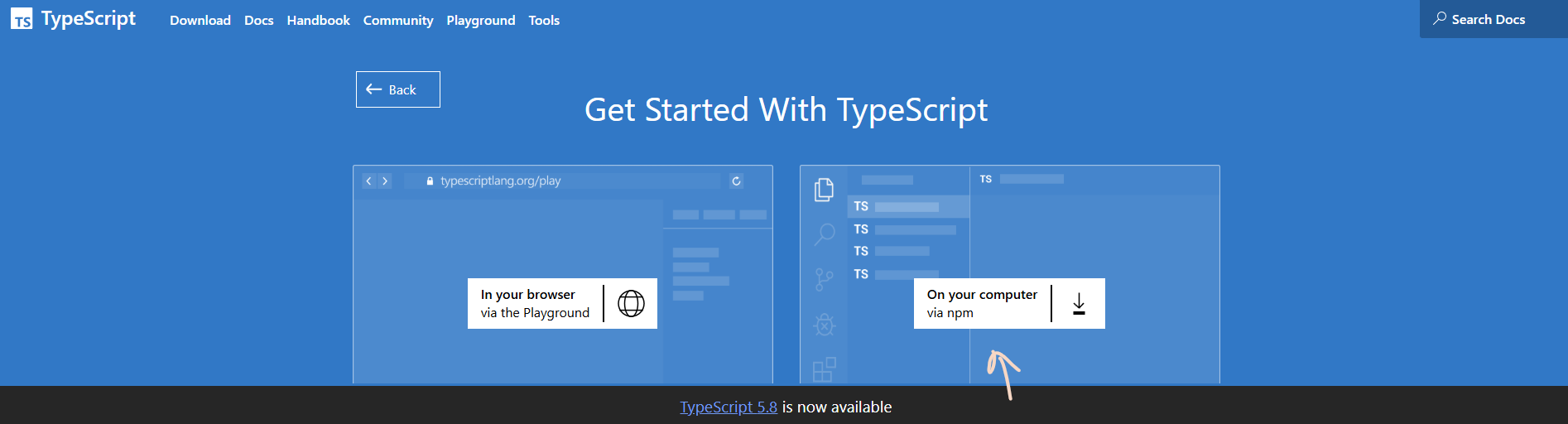
Using JavaScript we will find type errors during run on browser, we want to find those errors during development 🡪 This is where TypeScript shines.

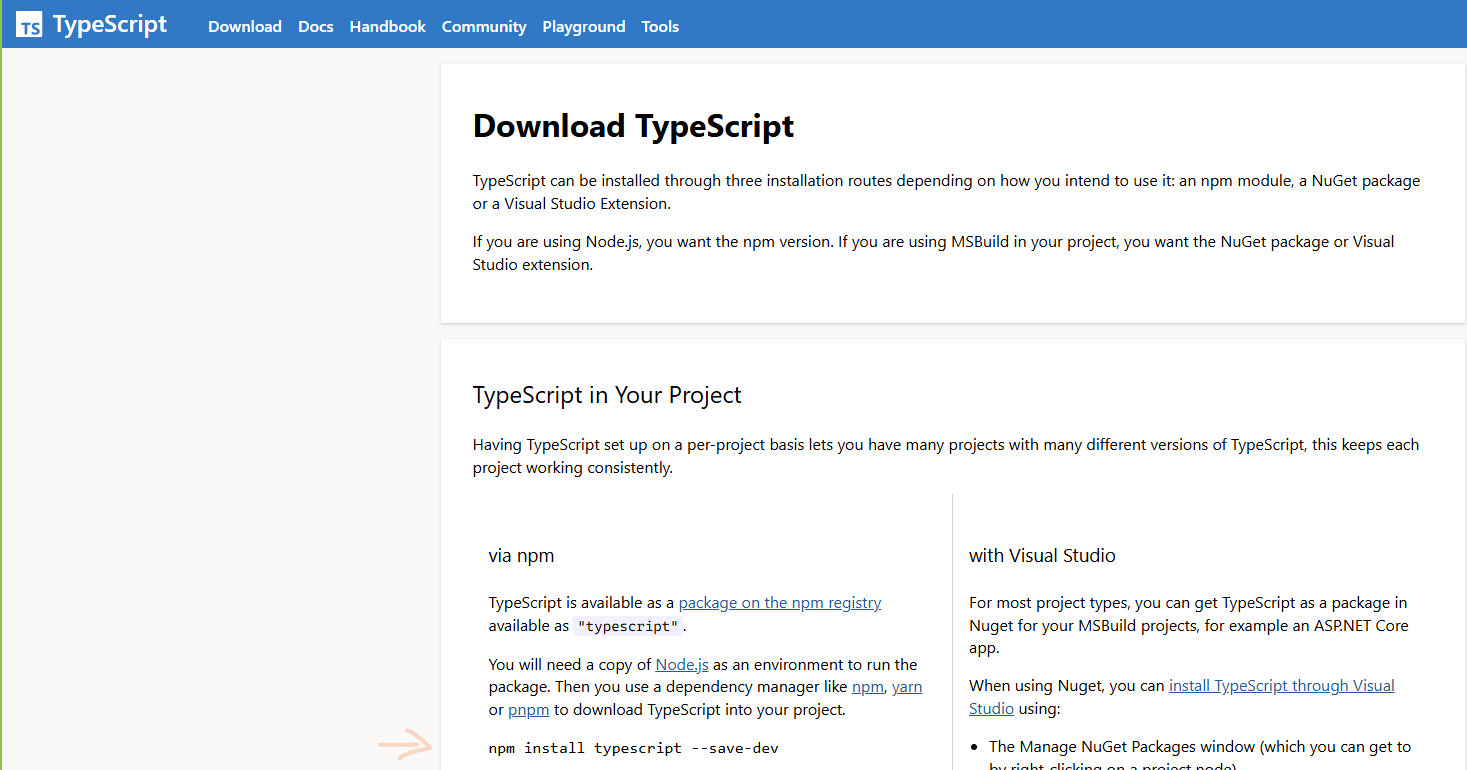
TypeScript is a language that can check the data types of our variables, it’s much stricter than JavaScript but worth it in the end, we can improve the developer experience by switching to TypeScript.

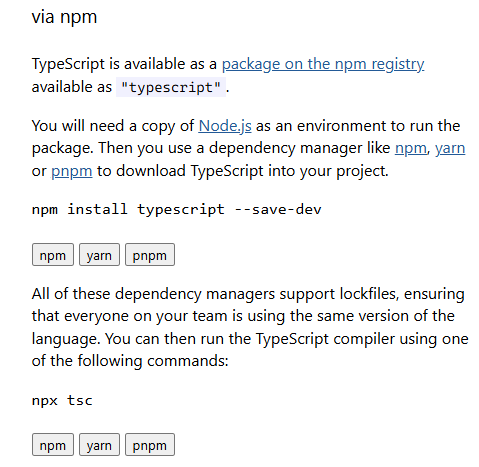


<https://www.typescriptlang.org/>









**npm install typescript –save-dev**

<https://www.npmjs.com/package/typescript>

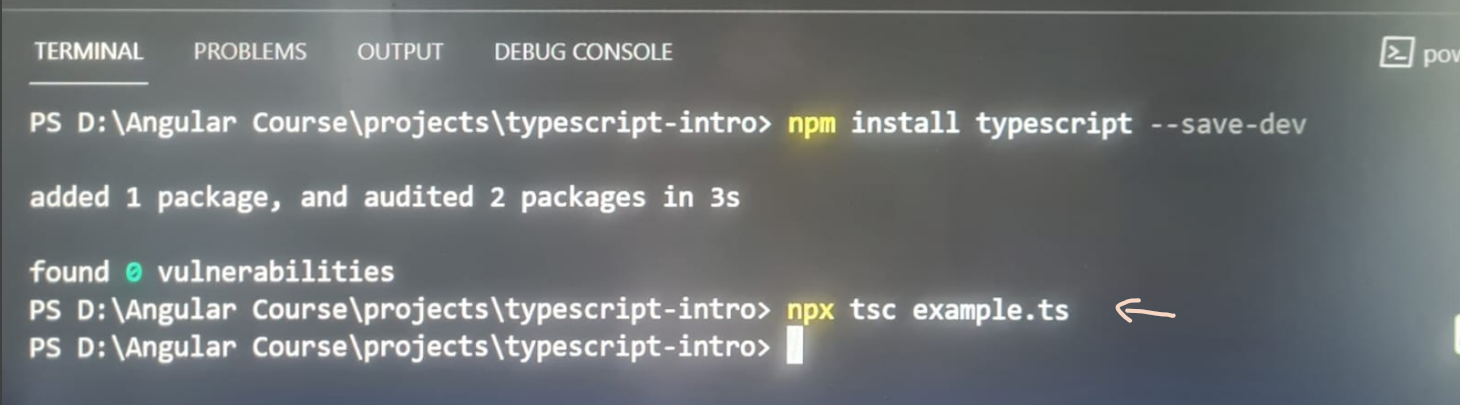


Compiling TS to JS in the command line do:

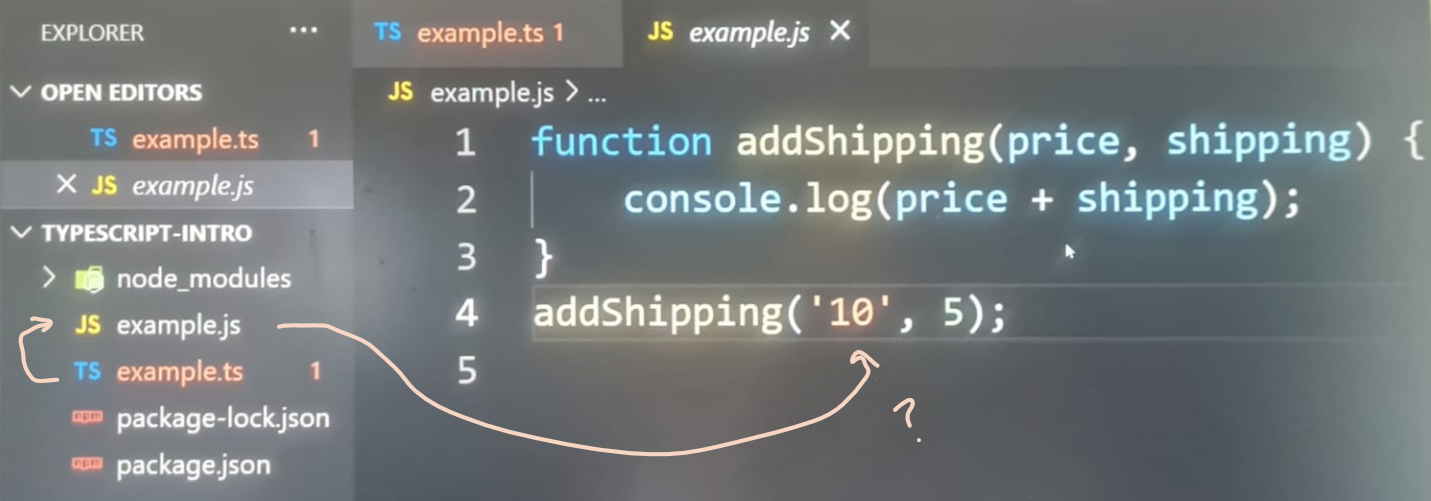
**npx tsc <file\_name.ts>**

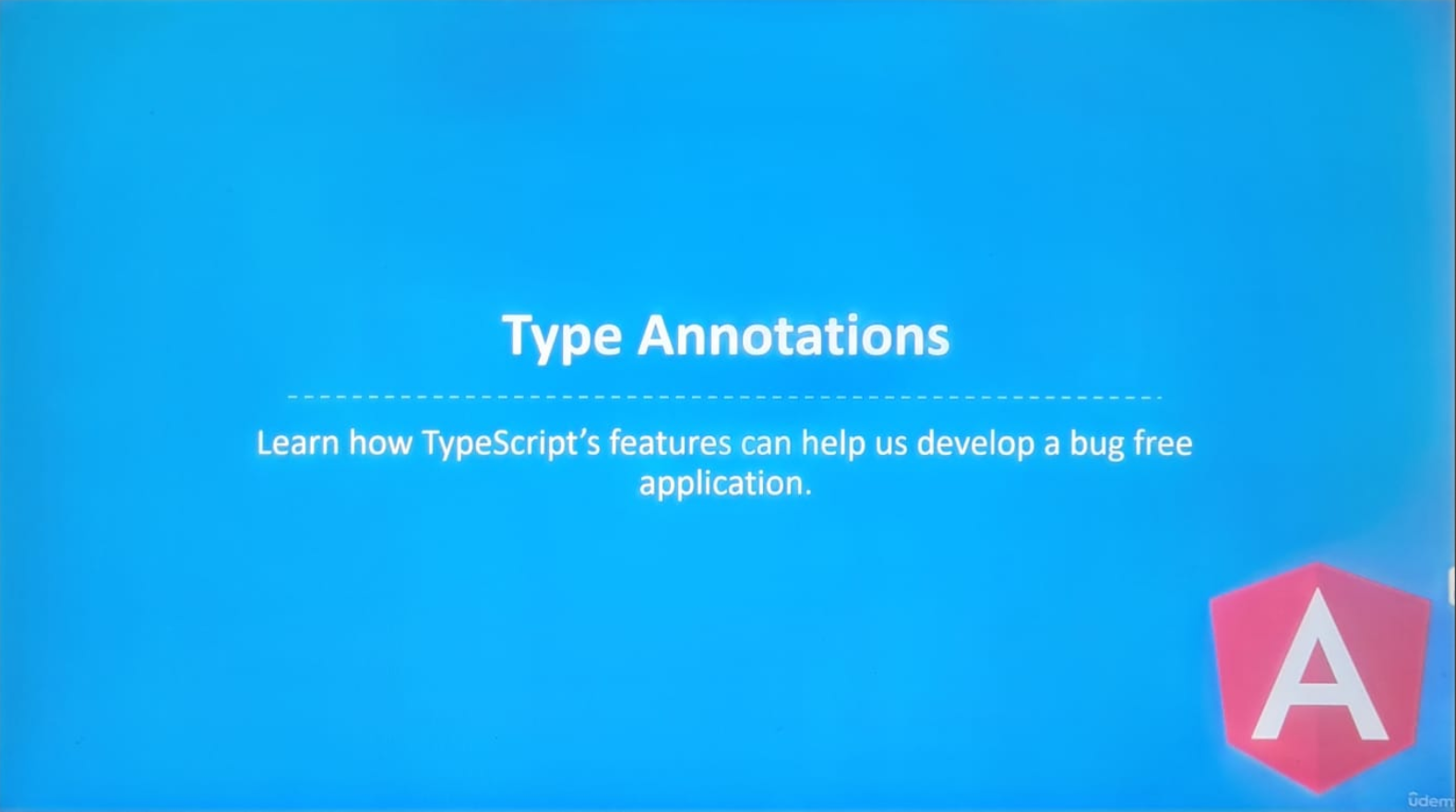
In this example:

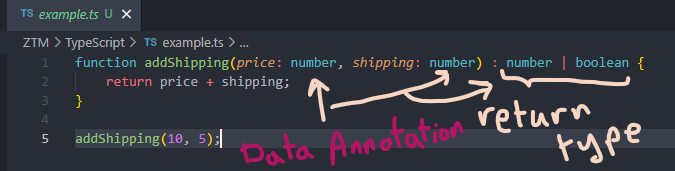
npx tsc example.ts

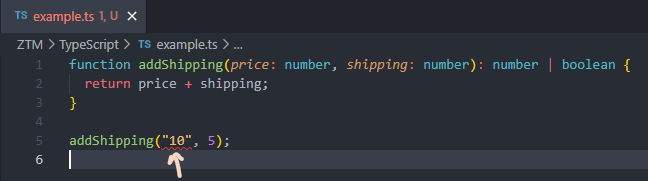


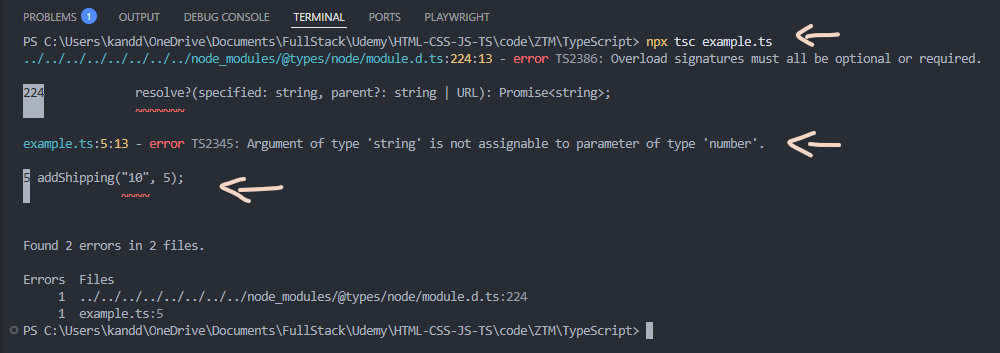
This will create the example.js file, however, when opening it, we see that the translated code is still have the same issue, so next we will address this.



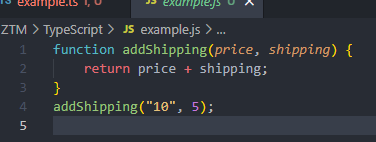








Even though there is an error, JS file will be created:



Let’s fix it:

npx tsc example.ts ../../../../../../../../node\_modules/@types/node/module.d.ts:224:13 - error TS2386: Overload signatures must all be optional or required. 224 resolve?(specified: string, parent?: string | URL): Promise<string>; ~~~~~~~ Found 1 error in ../../../../../../../../node\_modules/@types/node/module.d.ts:224 PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript>

**ChatGPT said:**

You're encountering this TypeScript compiler error:

swift

CopyEdit

error TS2386: Overload signatures must all be optional or required.

**Reason:**

This error is coming from a **type definition file** in @types/node, specifically in:

bash

CopyEdit

../../../../../../../../node\_modules/@types/node/module.d.ts:224

The specific line:

ts

CopyEdit

resolve?(specified: string, parent?: string | URL): Promise<string>;

**Cause:**

TypeScript 5.x introduced **stricter checking for overload signatures**.

In this case, the Node.js types define an **optional method overload**, but another overload of the same method is marked as required, causing a mismatch.

In short: **All overloads for a function must either be optional or all required. They cannot be mixed.**

**Why this happens to you now:**

You likely have one or both of the following:

* **Old version of @types/node** (written before TS 5.x stricter rules)
* **New TypeScript version (TS 5.x or higher)**

This mismatch triggers the error.

**✅ Solutions:**

**Option 1: Upgrade @types/node**

If your TypeScript version is new (TS 5.x or above), update Node type definitions:

bash

CopyEdit

npm install --save-dev @types/node@latest

Or explicitly for latest compatible with TS 5.x:

bash

CopyEdit

npm install --save-dev @types/node@20

(Adjust version as needed based on your Node version.)

**Option 2: Downgrade TypeScript (if you want to keep older Node types)**

For example:

bash

CopyEdit

npm install --save-dev typescript@4.9

That way older Node types won’t conflict.

**Option 3: Suppress the error (Not Recommended for production)**

You could add this to tsconfig.json temporarily:

json

CopyEdit

{

"compilerOptions": {

"skipLibCheck": true

}

}

This will **skip type-checking for all node\_modules/@types/\***, including Node types.

**✅ Recommended Fix for Now:**

If you want a quick solution with latest TypeScript:

1. Run:

bash

CopyEdit

npm install --save-dev @types/node@latest

1. If problem persists:

bash

CopyEdit

npm install --save-dev typescript@5.4

1. If still problematic → temporarily use:

json

CopyEdit

"skipLibCheck": true

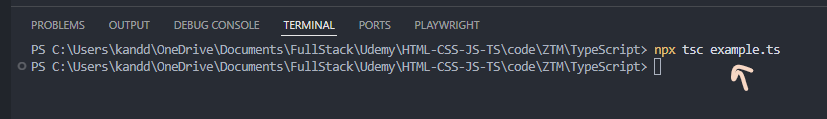
**✅ Summary:**

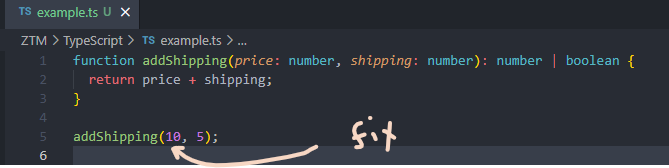
This is a common issue for TypeScript 5.x users using older DefinitelyTyped packages.  
**Best long-term fix:** Keep both TypeScript and your type definitions up-to-date and compatible.

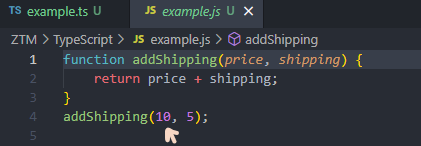
If you tell me your current TypeScript version (npx tsc --version), I can give you the exact safest fix for your setup.

* I will downgrade to TypeScript 4.9

**npm install --save-dev typescript@4.9**







The purpose of using TypeScript is to help us debug our application before running it in an environment, if the compiler can successfully compile our code, we should be safe from type errors.

# Everyday Types

<https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#the-primitives-string-number-and-boolean>

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](https://developer.mozilla.org/en-US/docs/Glossary/Primitive): 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.

Arrays

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](https://www.typescriptlang.org/docs/handbook/2/objects.html#tuple-types).

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;

[Try](https://www.typescriptlang.org/play/#code/DYUwLgBA9gRgVgLggQwHYE8IF4IG8IAeSADBAL4DcAsAFAD0dEAclKiNAGYRgAW7HUYMCgB3AJaoA5hGASQAZ04QAxlAAm7cUO48ATqJVQAtgAcxoXRBC79u+QDpaDCAFV5E6QAM06TxDVi8sgwoIrI2hwArrq81tzoJuzKfMoA1h4ANCioahBikIEo8vKRRiBqTozoUJEQqagGsVaoAG5i+qhlqJAw4GBxvGgQACoJIADKyrpiJmCONLBw9gJQABQAlNQL8Btbi-YwyJY4AIzExHvw2BAARHxCUDdbqqjykKhIqKW9x9DwFEA)

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 compiler flag [noImplicitAny](https://www.typescriptlang.org/tsconfig#noImplicitAny) to flag any implicit any as an error.

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";

[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAtgngcgQyiAXBAzmATgSwHYDmEAvBAEQCCwOAxiGQNwCwAUAPRsRfdcB6-A-hAAqMAA4gICPHgD2YBGByy8QA)

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";

[Try](https://www.typescriptlang.org/play/#code/PTAEDkHtQFwTwA4FNQEMB27I1TAlpOqOkkgCbmgC0VoA5ALZzioNJ2h7oBmSATn0qoAzrEQo6wmHy4BzOgFgAUABskMUExZtQAXlAAiAIIq8AYyQGA3EA)

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:

// Parameter type annotation

function greet(name: string) {

console.log("Hello, " + name.toUpperCase() + "!!");

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAUEMCdIWwKYBcHVEgngBwaSA7fAeyUiQEsj8BYAKADMBXfAYwqtAHNoFkAKfPAQAuUAGck0cvk4BKUAG86oUCBXqNm9QD1de3ctAsqYogBsEAOjNFOfAEQAJBGZsAaUPdABqUIMSWSEQAqlg40ADCkGIIfPK+9gCEifayANx0AL5AA)

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'.Argument of type 'number' is not assignable to parameter of type 'string'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFAAmkAxgDYCG0koAZgK4B2pALgJZxOgDmNkrABRNKAW0ipEraOyY8AlKgBucdkQDchEKAC0e0g1Z6dWsAHU4DckVAAjWpVDRmHcaBjxoodnXcAPMkNIIgBCQj5IAUEsNHl1IA)

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;

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAcwKZQGIEMBucBOMUqAciALYBGq+AFAJQBciYF1+iA3gLABQiiAPSCBoseIliAejNky+A-OhD4kAJgBsAbj4BfIA)

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.

**Functions Which Return Promises**

If you want to annotate the return type of a function which returns a promise, you should use the Promise type:

async function getFavoriteNumber(): Promise<number> {

return 26;

}

[Try](https://www.typescriptlang.org/play/#code/IYZwngdgxgBAZgV2gFwJYHsIwOYFNkBiwAbugE6rK4ByCAtgEa5kAUAlAFwwAKZ6dqELgA8EekzIA+GAG8AsACgYMMvgRksAJgBsAbkUBfIA)

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.

Here’s an example:

const names = ["Alice", "Bob", "Eve"];

// Contextual typing for function - parameter s inferred to have type string

names.forEach(function (s) {

console.log(s.toUpperCase());

});

// Contextual typing also applies to arrow functions

names.forEach((s) => {

console.log(s.toUpperCase());

});

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwFYMEYCwAoAYzgDtEAXUEgQwFtJFQBeUAbQCIBBAGwEtDI7ADSh2AITgAjYaICiAN0EBdANwECIUAGFS5SAA9yAV2rdQ5AJ4AHXiQDmoAGYInRkoXK9SoALSgr1NB0kHrQoIy2jjDQkAAm5nCgABbUiubWkOHk0LZ2BDT0iAB0ztCy1IRJABSObh5eJKBViACUoADeBKCgxGRw3JBF3HB2zUXkcACqVlYwWtSIkFUtLWr4AL6r6viaOiR6hiZmljb2oKaIidQzfAwJ57BwAO6u7p6kiPnBxaXllVXNNpMAB8HS6PQ+-UGw1GxQm01m0Hmi2WWw2qyAA)

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:

// The parameter's type annotation is an object type

function printCoord(pt: { x: number; y: number }) {

console.log("The coordinate's x value is " + pt.x);

console.log("The coordinate's y value is " + pt.y);

}

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

[Try](https://www.typescriptlang.org/play/#code/PTAEBUAsFNQBwIYCcEFtoBdpIOQGdQMBPOWBAO3IHsMEMBLK80egi0KgIwCtoBjDIRLQAsACgAZgFdyAxszhJ65DAGEqVJABMAFHAwAuUAG9QADyPkpqTtgDcoIpeu2koAL4BKE+NCgQfoFBwSF+AHoRkVHRMZG+oHxMeFQANtAAdClUAOY6AERQsImaWsp00PjmoABuCClSsKygeaAA1PAY6WaedvGJ5MlpmTn5hQka2mVYlUQ1dQ0sBC3t+ulEPeLu4orKahO6phagAMwANI5GAOwePUA)

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" });

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABABwE4zFAcgQwLYCmAFHAEYBWAXIgN6LAyoDOU1L6YA5gNyIA2OFgH42UDp0QBfAJS0AsAChEiAPQrEAOi2LJitYgBCcKAAtEAeQDSitBmz5idBs1aIAREdJup07jY72hEROjCzUbgCCfDAQBG4ANPyCrpHRTEwI3jLcQA)

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());

'obj.last' is possibly 'undefined'.'obj.last' is possibly 'undefined'.

if (obj.last !== undefined) {

// OK

console.log(obj.last.toUpperCase());

}

// A safe alternative using modern JavaScript syntax:

console.log(obj.last?.toUpperCase());

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygIwA4AMAWDBYAKADMBXAOwGMAXASznNAAdpbzqA5AQwFtIAKOACMAVqgDeoYrSTVUiaq3IBzANygANlwUB+eYrbLQAXwCUocUVCgQoAKKwEoALSgetZQAtqoStG2eoLTEoADkwiIAdFoKoaAA7trkoT4scAButAAmkFkAhFa+DIhwGpDRcMqCotHa1JHUcACqTEwwAMLaAqamqoXBoNVRMT55ALxjoBQ50uS55paE1ta2APIA0oXWlMWl5RqVQ7UKDc2tHV38PX1LJkSFtgCCoIhcxJCgXBrUMORcdOkPqREIY3HActBGAApLjpLgAZT8tCYPkQAE92FwAB7IQo7cglMoVKoRY7UHSnFptaCdRDdXpEYxAA)

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'.Argument of type '{ myID: number; }' is not assignable to parameter of type 'string | number'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFABmArgHYDGALgJZxmgAO0NZVAkgCYAUNnqZEgFsARjFAAfUIiosyAcwCUoAN6FQoCvURwANpAB0uuPO4AiAJpwS0UOwAioGilBnQAaiedFAbkIBfQhBQAHkAaUJmVg4eAEYABljfILBwyLkY8zR4tDNkgmCAUVgEdOiubhVQIQBPB1Q0TCw0UH9fIA)

The separator of the union members is allowed before the first element, so you could also write this:

function printTextOrNumberOrBool(

textOrNumberOrBool:

| string

| number

| boolean

) {

console.log(textOrNumberOrBool);

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABABwE4zFAKgUwB5QDyqAciALYBGOqxAQnHADYAUAsAFCKJT5GkVqtVA2YAuTt24AfRAGco6MAHNJU2WEE01MxJUZMcAQzCcAlIgDeaiAjnMcAOiZxlLXgWJkqNegbMA3JwAvkA)

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:

function printId(id: number | string) {

console.log(id.toUpperCase());

Property 'toUpperCase' does not exist on type 'string | number'.

Property 'toUpperCase' does not exist on type 'number'.Property 'toUpperCase' does not exist on type 'string | number'.

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

}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYME4CwAoAMwFcA7AYwBcBLOU0AB2mtMoEkATACmo9VOIBbAEYxQAH1CJKzUgHMAlKADeBUKHJ1EcADaQAdDrhyeHfZTgBVBgxgBhAIaJIXBQoDcBAL5A)

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);

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABABwE4zFAkgEwBQw4BciYIAtgEYCmqiAPogM5TpgDmAlIgN4CwAKESIYwRHigBPZNThjCiALzLEAIhZt2q7vyHDEAegOIsSKAAsYTRJVQBDSOYA0InCOtzEUmYgDkGjHZfQX1ECAQmOAAbagA6KLh2AhxYqDgAVWQZVABhOyZqPE5OAG4QxABfRGoogt5y4SNEAAlaahcFK0RPb2o-MipaYL1hcLBImPjE5NLyisEKoA)

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);

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAdwKYBsJwLaoAqpwAO6qAFAB4BciAzlAE4xgDmA2gLqIA+djzLAJSIA3gFgAUIkQxgiMgEEGDAIYBPAHQxaS1WsqDh4qdMQB6M4gASqBqhoByCg5m1ED+k1acHk04iwwWjhSDXQ4FjIAIht0cIAaRCjEAGpECg0AKzhmaMQVMAATJMMAbj9EAF9EDFpUUQrpC2tbe3dnV3dPAV8TaUDg0PDIqIB1DCxcRHCwesYVADcMWyTU9MFyk0rJSqA)

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.

Sometimes you’ll have a union where all the members have something in common. For example, both arrays and strings have a slice method. If every member in a union has a property in common, you can use that property without narrowing:

// Return type is inferred as number[] | string

function getFirstThree(x: number[] | string) {

return x.slice(0, 3);

}

[Try](https://www.typescriptlang.org/play/#code/PTAECUFMBcFcCcB2poE8AOlQEsDOPEAzSeeSAE1AEN9FYBbAIxIG0BdUAH1F2nm0QBzALAAoQrEQBjaNgD2yQTABi2eLwAqACzKQAFAA8AXKDpNWHbr35CAlKADeY0KDJwkoAwDpcAG2xS+gAMADSgAMy2ANxiAL5AA)

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:

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 });

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBACg9gSwHbCgXigbwLACgpQAeAXFEgK4C2ARhAE4DceBIpFN9TuAvl3gPT8oAUUIBDAMbAANiCjAAFtADOYytDHL5SqBDF1pCervGUw0iHgBm5JFIRwkUMHWTAAwnDh0AJgAowYFJ4NwBKLGYoCUdlOAsAOmk4AHM-ACIAFR1o7x9kMWAIAHItQigANzFpcmgELTSoAGpnYHjCUK4CaKRYhKTUzOyvX3zCkqg5Sura+qaW+JAOvG48PBc3T1y-TCJSAEYABgOAGgn9o6huDqA)

You can actually use a type alias to give a name to any type at all, not just an object type. For example, a type alias can name a union type:

type ID = number | string;

[Try](https://www.typescriptlang.org/play/#code/C4TwDgpgBAkgIlAvFAdgVwLYCMICcoA+UAzsLgJYoDmA3EA)

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";

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwHMQMBJVAB2QwAoBKALngGcMYtUCBuAWAChRIsBCnTY8zKKizYAXiGosYjRewINmrVT14B6HfAC0RsFSMG+GAJ7kEAVSYgYZShgDKk6VjnBXmjvABeDTYObT4RTFx8Jg9ZEGcqBVZlPzVGe0cEt1ivEB9U+ABvPnh4OAxkGGicuSSYWm0AXz4+PXgAYTgoDAQoCSk44Hh2Fz4IYnhkBycKKkD+zzks6iJSWZpaBpbdfXbJDSwICHgAIwQ4AygmJiwCVDz4AHdpAAt4PpV-DBecZAIXvhTTLreYAInuj2G61BnCAA)

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 });

[Try](https://www.typescriptlang.org/play/#code/JYOwLgpgTgZghgYwgAgAoHtRmQbwLABQyyAHgFzIgCuAtgEbQDchxAnhdfU4QL6GEwqIBGGDoQyAA5QsAYXTooAEwAUksBQxYAlLhbIE4gM7oANhAB0p9AHMVAIgAqACxSHFS0HEgByI6WQANzhTKhRgf3tkAGopMAsSbWYiA2MzS2s7J1dUjy9ff1YgkLDkCOQo2PULViTefgJpOQVlFRxSCgBGAAZugBpkdmQe7uQeJKA)

Just like when we used a type alias above, the example works just as if we had used an anonymous object type. TypeScript is only concerned with the *structure* of the 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  interface Animal {  name: string;  }  interface Bear extends Animal {  honey: boolean;  }  const bear = getBear();  bear.name;  bear.honey; | Extending a type via intersections  type Animal = {  name: string;  }  type Bear = Animal & {  honey: boolean;  }  const bear = getBear();  bear.name;  bear.honey; |
| Adding new fields to an existing interface  interface Window {  title: string;  }  interface Window {  ts: TypeScriptAPI;  }  const src = 'const a = "Hello World"';  window.ts.transpileModule(src, {}); | A type cannot be changed after being created  type Window = {  title: string;  }  type Window = {  ts: TypeScriptAPI;  }  // Error: Duplicate identifier 'Window'. |

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](https://www.typescriptlang.org/play?#code/PTAEGEHsFsAcEsA2BTATqNrLusgzngIYDm+oA7koqIYuYQJ56gCueyoAUCKAC4AWHAHaFcoSADMaQ0PCG80EwgGNkALk6c5C1EtWgAsqOi1QAb06groEbjWg8vVHOKcAvpokshy3vEgyyMr8kEbQJogAFND2YREAlOaW1soBeJAoAHSIkMTRmbbI8e6aPMiZxJmgACqCGKhY6ABGyDnkFFQ0dIzMbBwCwqIccabcYLyQoKjIEmh8kwN8DLAc5PzwwbLMyAAeK77IACYaQSEjUWZWhfYAjABMAMwALA+gbsVjoADqgjKESytQPxCHghAByXigYgBfr8LAsYj8aQMUASbDQcRSExCeCwFiIQh+AKfAYyBiQFgOPyIaikSGLQo0Zj-aazaY+dSaXjLDgAGXgAC9CKhDqAALxJaw2Ib2RzOISuDycLw+ImBYKQflCkWRRD2LXCw6JCxS1JCdJZHJ5RAFIbFJU8ADKC3WzEcnVZaGYE1ABpFnFOmsFhsil2uoHuzwArO9SmAAEIsSFrZB-GgAjjA5gtVN8VCEc1o1C4Q4AGlR2AwO1EsBQoAAbvB-gJ4HhPgB5aDwem-Ph1TCV3AEEirTp4ELtRbTPD4vwKjOfAuioSQHuDXBcnmgACC+eCONFEs73YAPGGZVT5cRyyhiHh7AAON7lsG3vBggB8XGV3l8-nVISOgghxoLq9i7io-AHsayRWGaFrlFauq2rg9qaIGQHwCBqChtKdgRo8TxRjeyB3o+7xAA), 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](https://www.typescriptlang.org/play?#code/PTAEEEDtQS0gXApgJwGYEMDGjSfdAIx2UQFoB7AB0UkQBMAoEUfO0Wgd1ADd0AbAK6IAzizp16ALgYM4SNFhwBZdAFtV-UAG8GoPaADmNAcMmhh8ZHAMMAvjLkoM2UCvWad+0ARL0A-GYWVpA29gyY5JAWLJAwGnxmbvGgALzauvpGkCZmAEQAjABMAMwALLkANBl6zABi6DB8okR4Jjg+iPSgABboovDk3jjo5pbW1d6+dGb5djLwAJ7UoABKiJTwjThpnpnGpqPBoTLMAJrkArj4kOTwYmycPOhW6AR8IrDQ8N04wmo4HHQCwYi2Waw2W1S6S8HX8gTGITsQA).
* Interfaces may only be used to [declare the shapes of objects, not rename primitives](https://www.typescriptlang.org/play?#code/PTAEAkFMCdIcgM6gC4HcD2pIA8CGBbABwBtIl0AzUAKBFAFcEBLAOwHMUBPQs0XFgCahWyGBVwBjMrTDJMAshOhMARpD4tQ6FQCtIE5DWoixk9QEEWAeV37kARlABvaqDegAbrmL1IALlAEZGV2agBfampkbgtrWwMAJlAAXmdXdy8ff0Dg1jZwyLoAVWZ2Lh5QVHUJflAlSFxROsY5fFAWAmk6CnRoLGwmILzQQmV8JmQmDzI-SOiKgGV+CaYAL0gBBdyy1KCQ-Pn1AFFplgA5enw1PtSWS+vCsAAVAAtB4QQWOEMKBuYVUiVCYvYQsUTQcRSBDGMGmKSgAAa-VEgiQe2GLgKQA).
* Interface names will [*always* appear in their original form](https://www.typescriptlang.org/play?#code/PTAEGEHsFsAcEsA2BTATqNrLusgzngIYDm+oA7koqIYuYQJ56gCueyoAUCKAC4AWHAHaFcoSADMaQ0PCG80EwgGNkALk6c5C1EtWgAsqOi1QAb06groEbjWg8vVHOKcAvpokshy3vEgyyMr8kEbQJogAFND2YREAlOaW1soBeJAoAHSIkMTRmbbI8e6aPMiZxJmgACqCGKhY6ABGyDnkFFQ0dIzMbBwCwqIccabcYLyQoKjIEmh8kwN8DLAc5PzwwbLMyAAeK77IACYaQSEjUWY2Q-YAjABMAMwALA+gbsVjNXW8yxySoAADaAA0CCaZbPh1XYqXgOIY0ZgmcK0AA0nyaLFhhGY8F4AHJmEJILCWsgZId4NNfIgGFdcIcUTVfgBlZTOWC8T7kAJ42G4eT+GS42QyRaYbCgXAEEguTzeXyCjDBSAAQSE8Ai0Xsl0K9kcziExDeiQs1lAqSE6SyOTy0AKQ2KHk4p1V6s1OuuoHuzwArMagA) in error messages, but *only* when they are used by name.
* Using interfaces with extends [can often be more performant for the compiler](https://github.com/microsoft/TypeScript/wiki/Performance#preferring-interfaces-over-intersections) than type aliases with intersections

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 HTMLCanvasElement;

[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBAtgTwMIEMwDcURgXhgExGAFc4BTMKAOgHMyoBRAGzPMoCEEBJfACgCI4KAJZgA+sDSYI-AJQwsMABIAVALIAZVBizNWFKAG4gA)

Like a type annotation, type assertions are removed by the compiler and won’t affect the runtime behavior of your code.

You can also use the angle-bracket syntax (except if the code is in a .tsx file), which is equivalent:

const myCanvas = <HTMLCanvasElement>document.getElementById("main\_canvas");

[Try](https://www.typescriptlang.org/play/#code/MYewdgzgLgBAtgTwMIEMwDcURgXhgHgAkAVAWQBlUMsBRAGwFM4GwoA+AExGAFdnWAdAHMGUekxZQAQggCSHABQAiOCgCWYAPrA0mCEoCUAbiA)

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.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.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYCsaCwAoAYzgDtEAXUAD1AF5QAiAC0gBtW4HQBDRUEgK4BbAEYwA3EA)

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;

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEYD2A7AzgF3iAHgBxgC54oUBPAbgFgAoDMvBAFXgF54BvE4gRgvgBGxAEz8wxAMzwAvtRoB6efAC0qsAFcMq5bWTosUNtnwwSaEuTPwmFIA)

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.

let changingString = "Hello World";

changingString = "Olá Mundo";

// 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"

[Try](https://www.typescriptlang.org/play/#code/DYUwLgBAxgFghgOwOYEtkGUwCc1IgXggCIAJEYYAewgHVKtgATIgbgFgAoWRVDbXAsQDywAIcQAsgFcEjSq04B6RRABCIKHCkBnEBAAG3ZLkw5k+6IghYQABxu6EkRAE8Itytu0oARqAja-MgANBBg8GBKKijaEDCUAO4QACoutiDoUDi2kIwg2lm++RAokGhhMHpgaXraLoEgALacRrxIprjsHMoQAHoA-JwtlAiB0COBiGAdyIKk5FS09EwK3Srqmjp6hhNgUzNIFpoIECPAbjb2+SBOEACM7p7efrVBSKGlUXFwsXAQwKUQFg4MAwjVrHYHDc9mAUCNhqM9k4Dl0egMgA)

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"'.Type '"howdy"' is not assignable to type '"hello"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGY1oLACgAbSAF1AA9UAiAC0gILktAF5Qa6HKBufEUAeQDS+Miza16jHnj4A6eSLE04AdwAmAT25A)

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 '"left" | "right" | "center"'.Argument of type '"centre"' is not assignable to parameter of type '"left" | "right" | "center"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFABmArgHYDGALgJZxmgAO0NZVAKpAB5UAUKoRFRZkA5gBpQAQwA2NUWQC2kNqgBEMyESprQAH1BqWogBY79hiiqow1ASlABvQqFAhQAOi+EAvoWasHNx8agASkDIycJIA7ggyACZqkhpaOnYA3P4iQTy8agDiAOQJUgCekopSNsmW1tCQ9hlAA)

Numeric literal types work the same way:

function compare(a: string, b: string): -1 | 0 | 1 {

return a === b ? 0 : a > b ? 1 : -1;

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABBOBbADgQwE4FMAUmAXIgM5TYxgDmANIgEYnmU0CUJAtAIyIA+iAAz9EvAN4BYAFCJEeKCGxJMiALzrGiAPxDEJFQD5NO3l24BuaQF8gA)

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"'.Argument of type '"automatic"' is not assignable to parameter of type 'Options | "auto"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFACWAdgC4wBmAhgMaSgDyADmUXCYqAN6GigB3IgBMyAC1QkArgFsARjADchAL6FKUkrTYdQtDpSIBzKdEgAKAB6oWOzqAA+oAETUpZOM4CUPPqBCgAHTBqoT6JIYmZubcgiLiqACMAAzJoCpeygThkaYWru6emWEGxnnmBR4y1Gy03opAA)

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;

}

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEYD2A7AzgF3mpBbEAwqsAJYYmoBc8ARkkhCFCgNwCwAUAPRfwC0AsAFcMAvp2TosSGgCt4AXngBvREiEoMIGNQAM8AL7sOJAGbwAFDnxEUpcqgCUKzvHgzZAOmQatMRfAAjMYGQA)

TypeScript doesn’t assume the assignment of 1 to a field which previously had 0 is an error. Another way of saying this is that obj.counter must have the type number, not 0, because types are used to determine both *reading* and *writing* behavior.

The same applies to strings:

declare function handleRequest(url: string, method: "GET" | "POST"): void;

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"'.Argument of type 'string' is not assignable to parameter of type '"GET" | "POST"'.[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYAsBWAsAFAAmkAxgDYCG0koAZgK4B2pALgJZxOgAWlTRcpABKkAI4NIiVgAoG0cqmnR2TAOYAaUAFtIrHnCKoARAHEAogBVjoAD6hjABQDyAZWsBKVADc47IgDchISkXNKgNGKgALygAN6g8ooOPKysAA4oIJAAHpTa6UIAdKHaxlq6+oYmFtagAL5BBHwCQqISUrKRRUla3ZUGRB4BQA)

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");

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwAspVgIQAlEAR2RAGcMAKZGCALngZi1QHMAaeAFsQGAjmAcARAHEAogBUp8AD7wpABQDyAZSUBKDgDccWYAG4AsACgA9LfgBaZ2GQZnjm-fgBhInwQARjYbMDwGeDgqeABeeABveBZ2dQIMDAAHOjZ7EAAPKCEMsgA6MKEpQRExCWl5JXgoOnV65QBfKzsHP2JeBAAmG39SCmpaBkYokuTBKerxYEbm2UUpfXMgA)

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"“.

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

const req = { url: "https://example.com", method: "GET" } as const;

handleRequest(req.url, req.method);

[Try](https://www.typescriptlang.org/play/#code/CYUwxgNghgTiAEAzArgOzAFwJYHtXwAspVgIQAlEAR2RAGcMAKZGCALngZi1QHMAaeAFsQGAjmAcARAHEAogBUp8AD7wpABQDyAZSUBKDgDccWYAG4AsACgA9LfgBaZ2GQZnjm2DwN4cKvAAvPAA3vAs7OoEGBgADnRs9iAAHlBCsWQAdN5CUoIiYhLS8krwAL7wUHTw3qgMVtZEJGSUNPRM-pkRgp0F4sD65kA)

The as const suffix acts like const but for the type system, ensuring that all properties are assigned the literal type instead of a more general version like 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](https://www.typescriptlang.org/tsconfig#strictNullChecks) option on.

strictNullChecks off

With [strictNullChecks](https://www.typescriptlang.org/tsconfig#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](https://www.typescriptlang.org/tsconfig#strictNullChecks) on if it’s practical to do so in their codebase.

strictNullChecks on

With [strictNullChecks](https://www.typescriptlang.org/tsconfig#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());

}

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAEzgZTgWwKZQBYxgDmAFAB4BciAzlAE6FGIA+iYIANhwJSIDeAWABQiRDGCJyiALyy2nHv2GjRAelUo4bOPkbLEAX0TYO1bEpEqICanA7YAdBzikARAAkTzgDSJXiAGpEMgcoOABVAAdI7DoAYQBDMxJubgBufQNhAyA)

Non-null Assertion Operator (Postfix !)

TypeScript also has a special syntax for removing null and undefined from a type without doing any explicit checking. Writing ! after any expression is effectively a type assertion that the value isn’t null or undefined:

function liveDangerously(x?: number | null) {

// No error

console.log(x!.toFixed());

}

[Try](https://www.typescriptlang.org/play/#code/GYVwdgxgLglg9mABAGxgNwKYBECGYDmGATnCAM7ICeAFAB4D8AXImCALYBGxiAPiyMmQBKRAG8AsAChEiAPSzEAOTiJiJIlJkQEZOMgwA6ZHHx0AhAahwAYjFoYAJtSFCA3FIC+QA)

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](https://www.typescriptlang.org/docs/handbook/enums.html).

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:

// Creating a bigint via the BigInt function

const oneHundred: bigint = BigInt(100);

// Creating a BigInt via the literal syntax

const anotherHundred: bigint = 100n;

[Try](https://www.typescriptlang.org/play/#code/PTAEAEBcEMCcHMCmkBcpEGcBMAGXBYAKCJFAGFZFpIBLAO3lGlACMb57JQA3G5yABaJQAIXYBJOlwBmAVzoBjWgHs6RBaoxdViABLyAJpQNo2HKaAC8oiVIAUARhw4AlAG4iJMBSq0GTG3hJLl5+IVAAGxpIRFhoCNAMAE8paAAPdU0uaDplQVj9OiNEE1Z2TitQJxw6NyA)

You can learn more about BigInt in [the TypeScript 3.2 release notes](https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-2.html#bigint).

**symbol**

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 comparison appears to be unintentional because the types 'typeof firstName' and 'typeof secondName' have no overlap.This comparison appears to be unintentional because the types 'typeof firstName' and 'typeof secondName' have no overlap.

// Can't ever happen

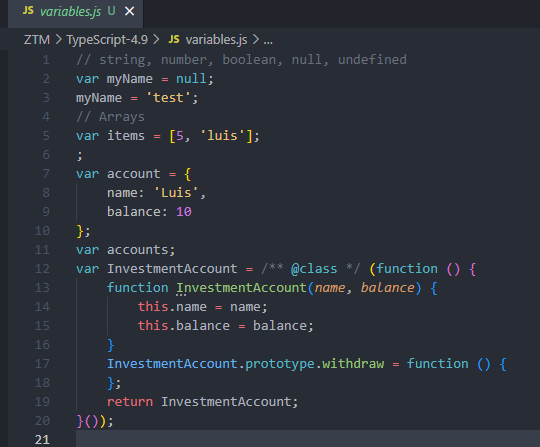
}

[Try](https://www.typescriptlang.org/play/#code/PTAEAEFMCdoe2gZwFygEwGYBsB2AsAFADGcAdogC6gBmAlkhQHICGAtpKALygDKAnqwBGcADYAKAESk2kCQEoA3IRLkqiSCoAmLdl14Dh4qTPlKChWtVBi6DHR06PQ6rfbmgA3oVCgQoAMLMpADkVJAAbjCgABbMAA5xkKSEAL5AA)

You can learn more about them in [Symbols reference page](https://www.typescriptlang.org/docs/handbook/symbols.html).

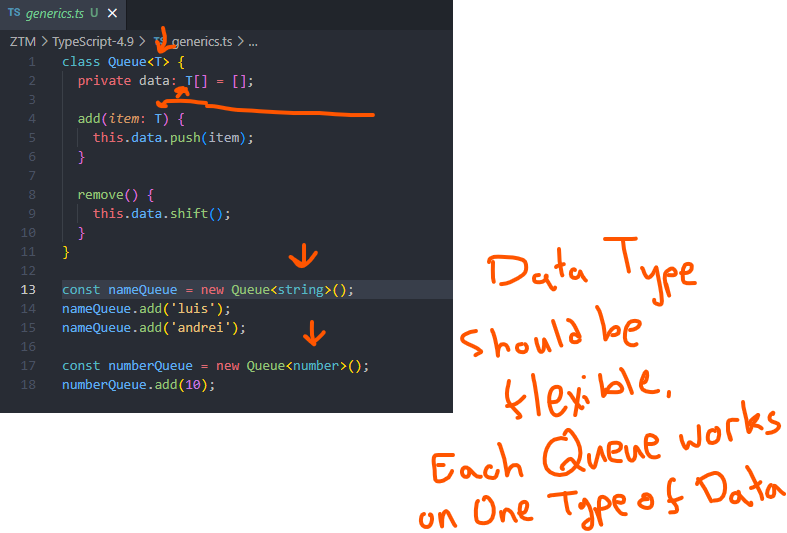


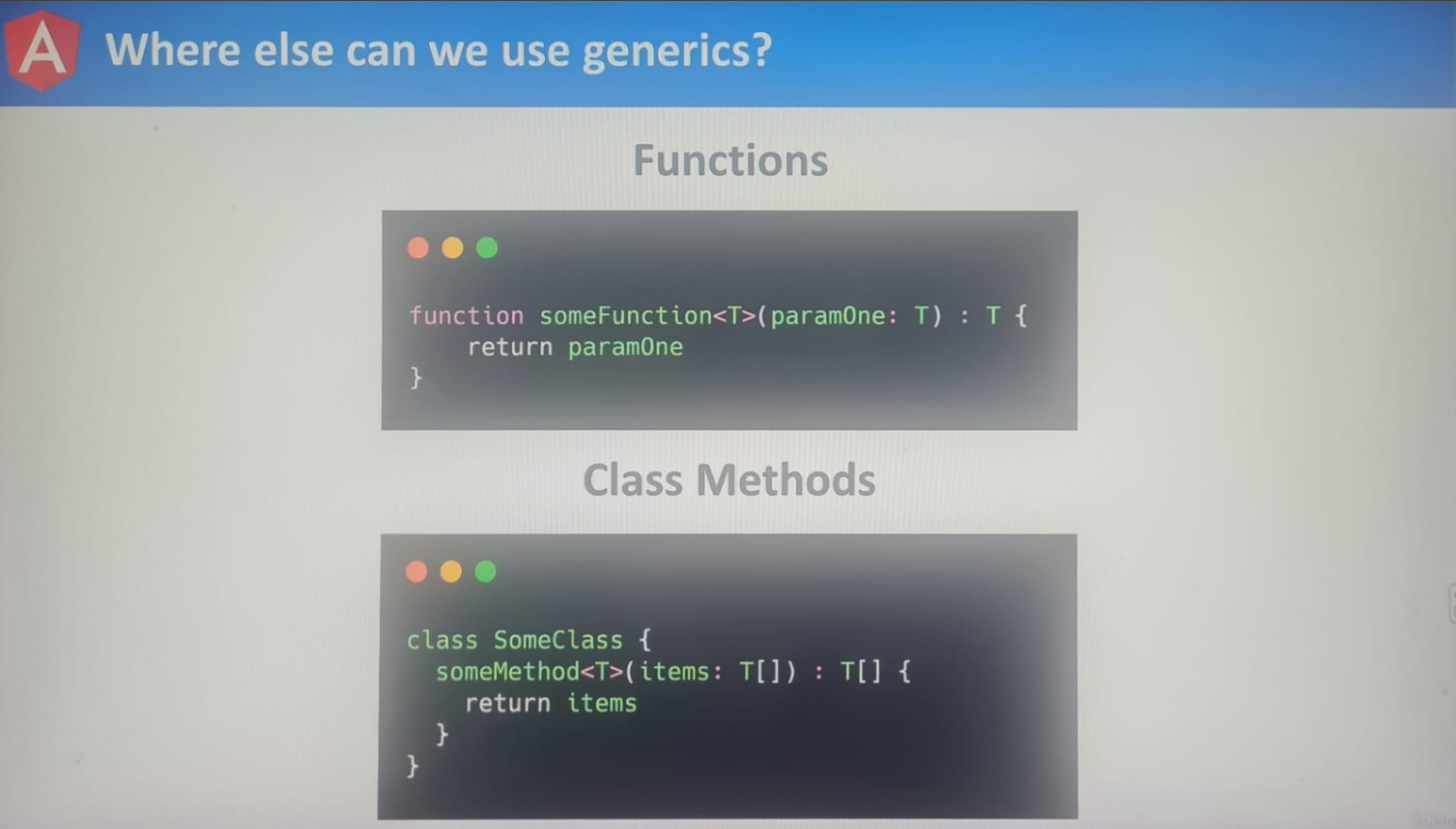




# Generics

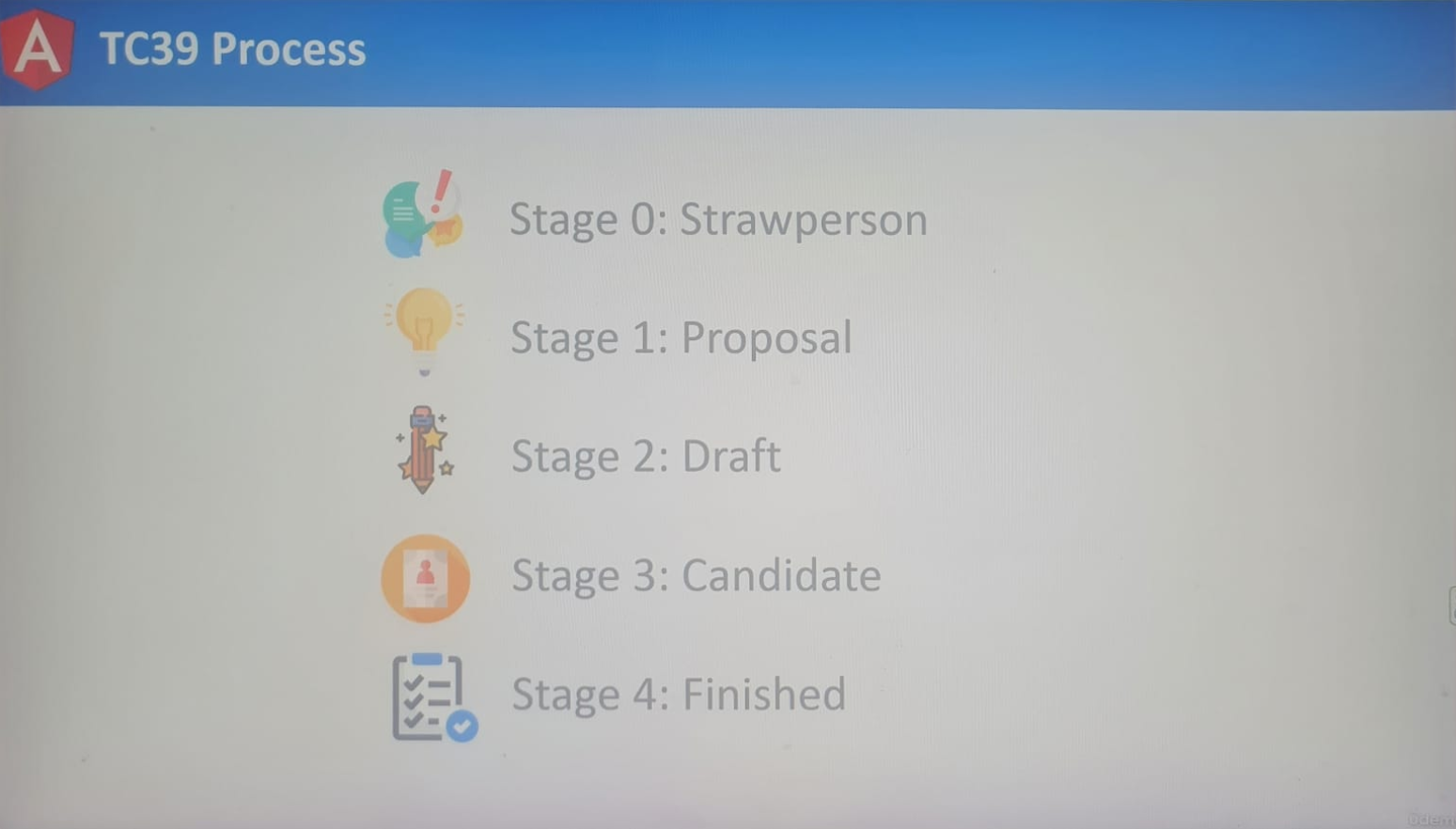
Generics allow functions or classes to be strict and flexible at the same time.



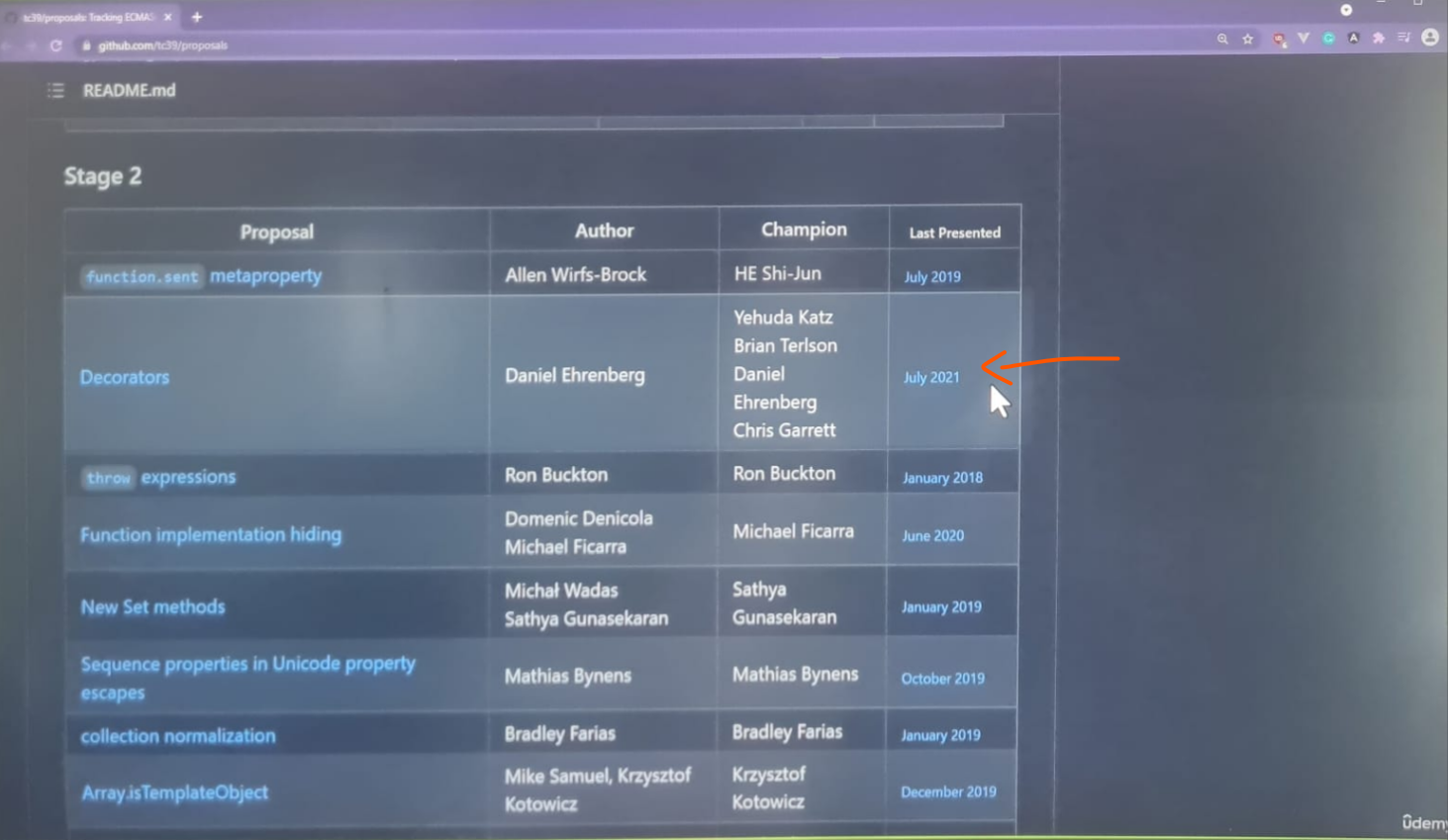


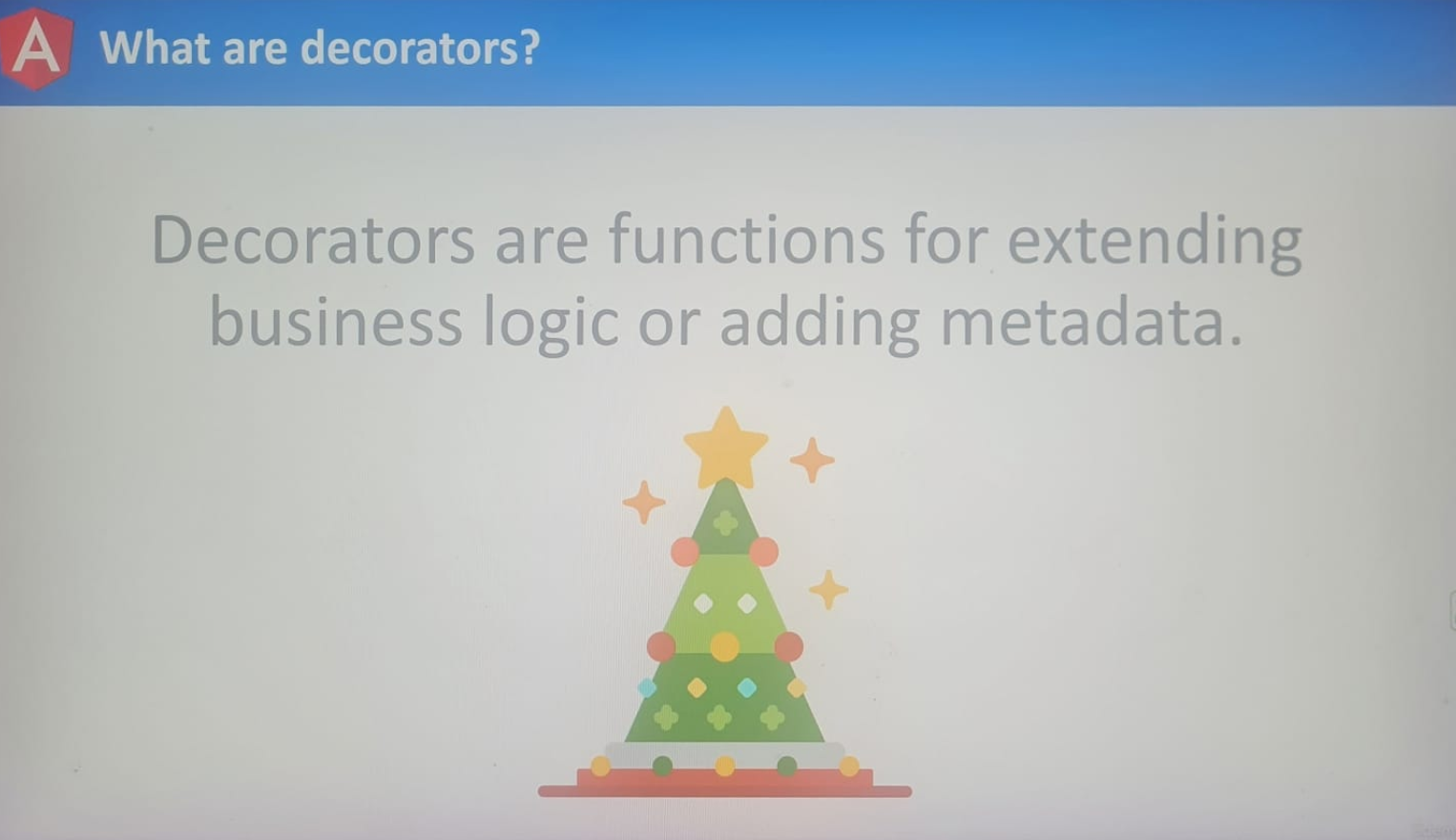
# Decorators





<https://github.com/tc39/proposals>



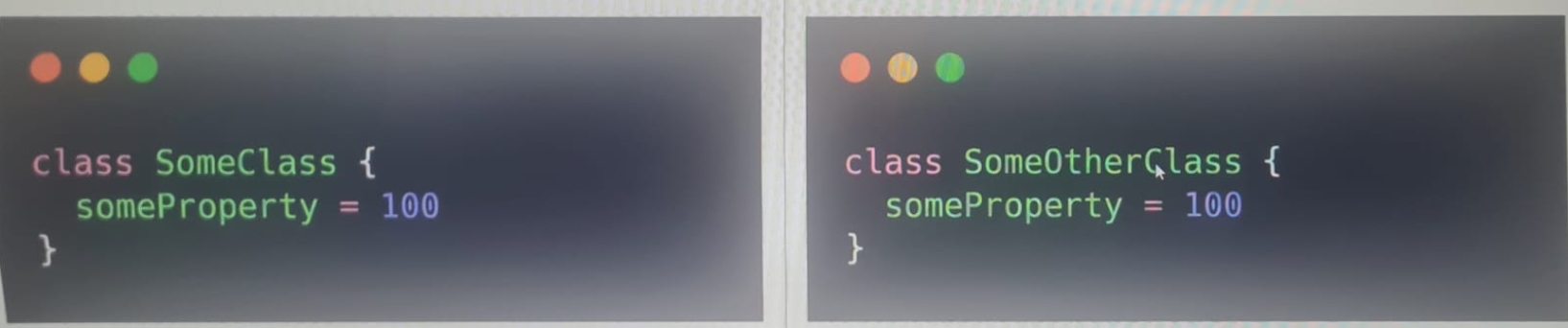


Decorators are functions for extending business logic or adding metadata.

Let’s imagine we had a Christmas tree. By themselves, Christmas trees are plain and boring, that’s why we add decorations to them, Decorations can liven up a tree, it doesn’t matter how many decorations you’ve added to the tree, it’s still considered a tree.

In a similar sense, Decorators work the same way. They allow us to modify an existing piece of code after the decorator has made its modifications, we’ll returned the same piece of code, but with the modifications.

Let’s look at an example of what problems decorators solve:

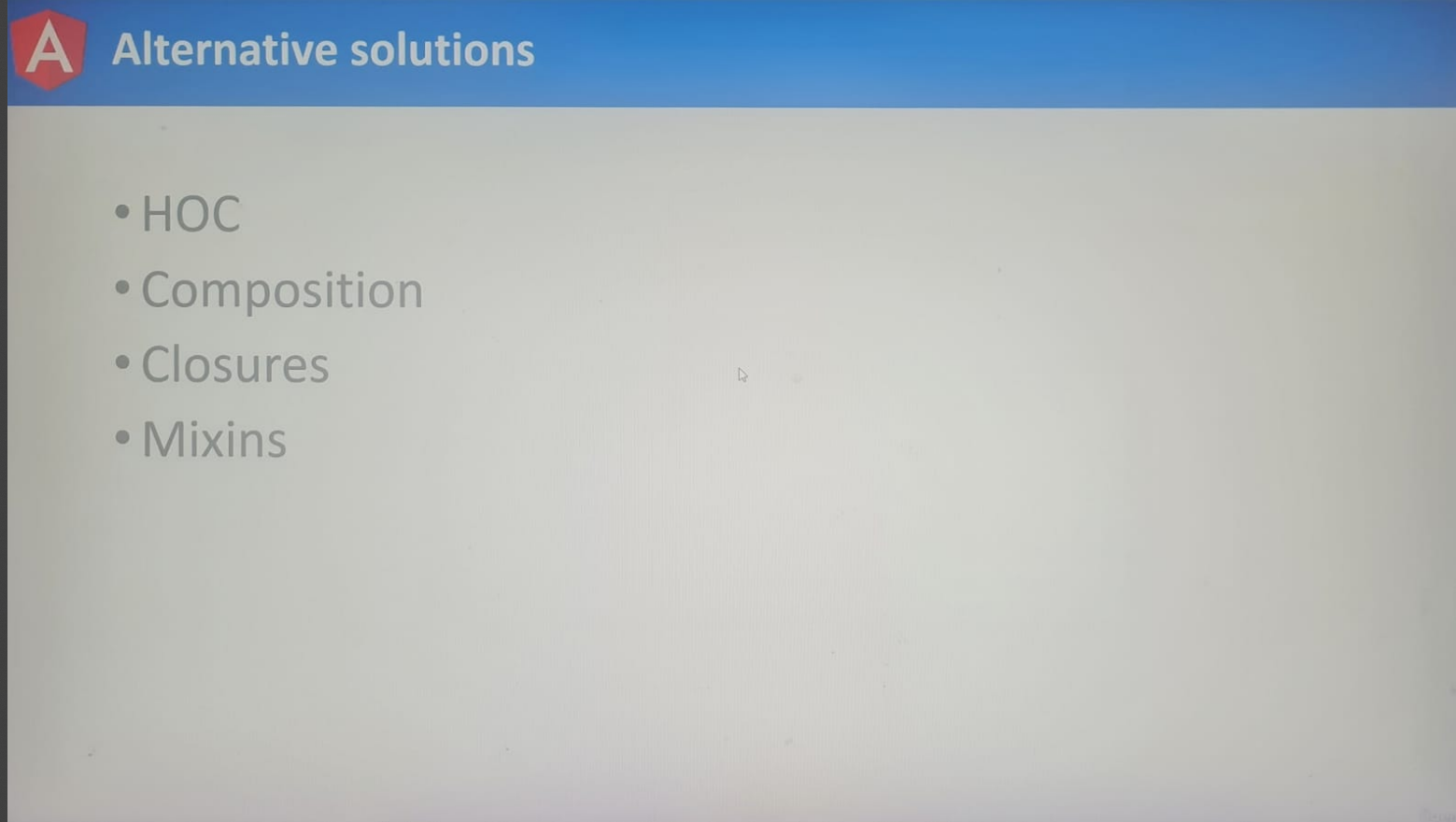


Imagine if we had two classes with dozens of similar properties and methods, we should avoid repeating the same code whenever possible.

This problem is as old as time.

Avoiding duplicate code is an issue developers face.

Through the years, Developers have come up with solutions for fixing this issue:



We can use:

* High Order Components
* Composable Functions
* Closures

You may be familiar with these solutions if you worked with React or Vue, each of these solutions works, but there are some drawbacks to each of them.

Decorators resolve a lot of these issues with these other patterns.

One of the advantages of Decorators is that we can apply them to classes, properties, methods, accessors and parameters.

Thus, they can reach certain areas in our code where other patterns can’t.

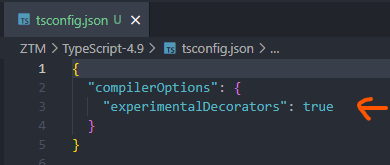
We are going to configure TypeScript.

Decorators are an experimental feature, if we want to use Decorators, we need to configure TypeScript to enable them, by creating a configuration file.

<https://www.typescriptlang.org/tsconfig/>

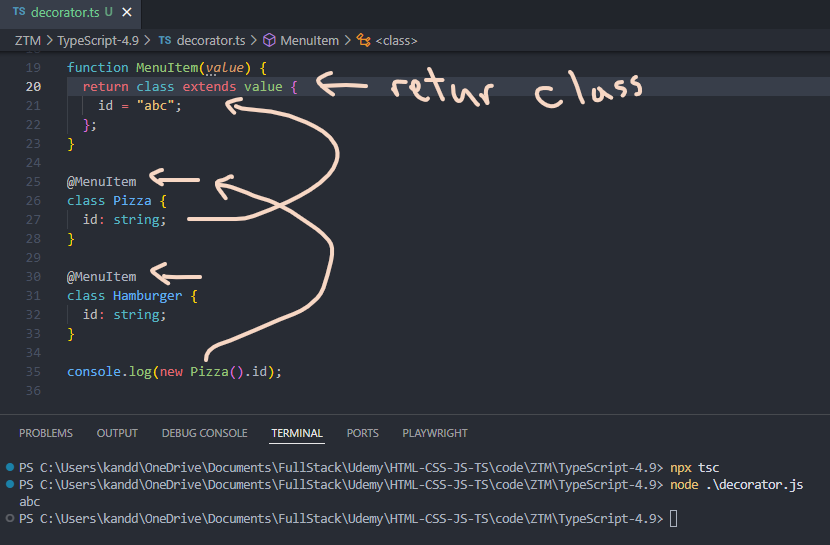
**npx tsc - -init**

After running this command, TypeScript will have created a file called: **tsconfig.json**, make sure to have the following inside it:

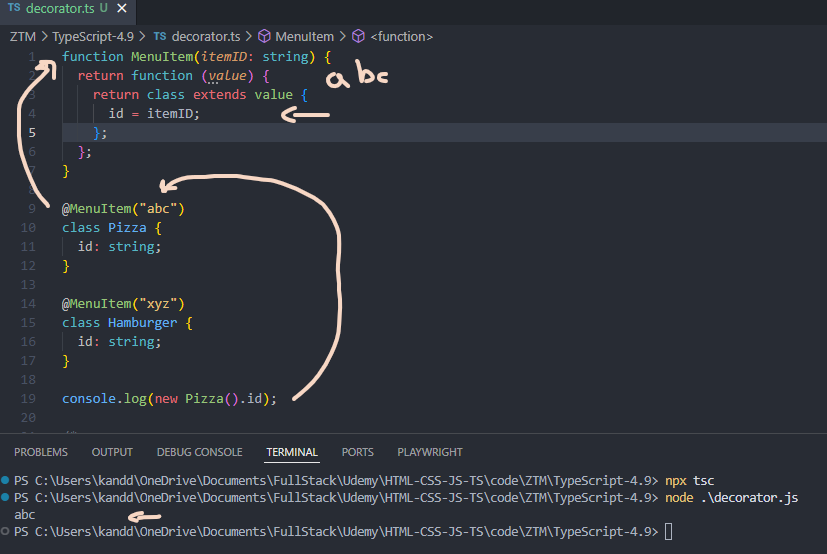


# Writing Decorators





We can take this a step further by passing on values to our decorators:



OR:

