

# Introduction to TypeScript

A programming language primarily used in Angular.



What is TypeScript?

TypeScript is a **superset** of JavaScript.



Udemy



## Data Types

- Boolean
- Number
- String
- Object
- Function
- undefined
- And More!

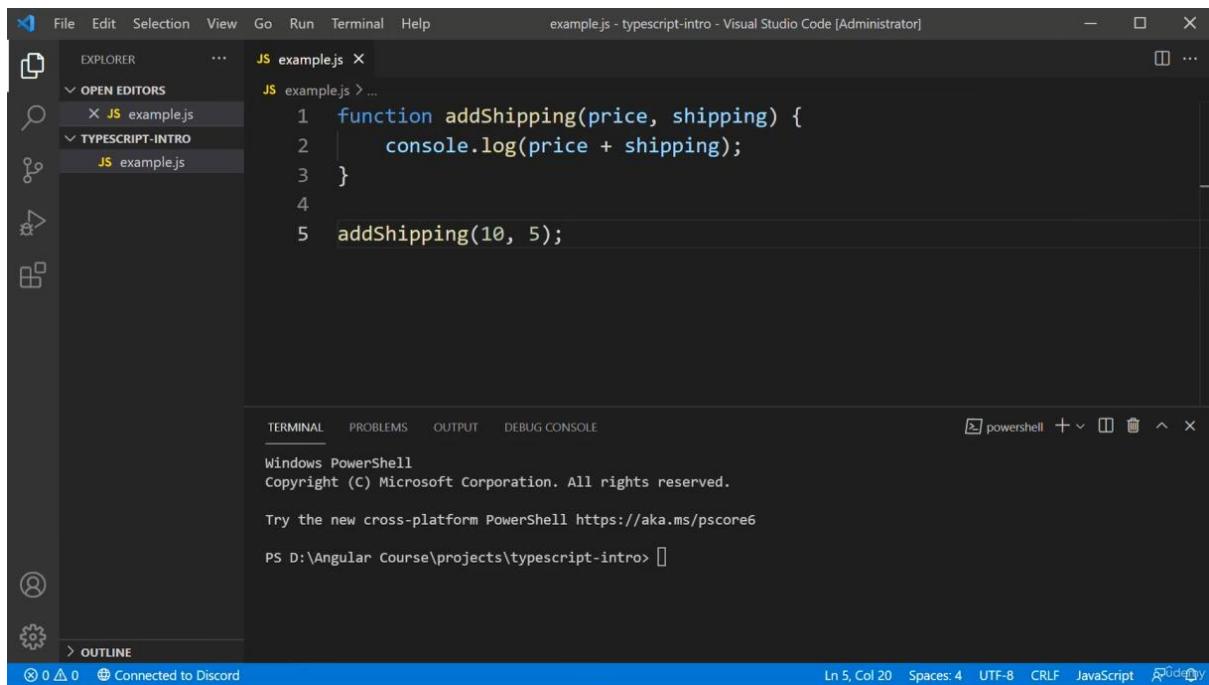


```
let price = 5; // Number  
price = '10'; // Number -> String
```

Odemy



Odemy



The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there are two files: 'example.js' under 'OPEN EDITORS' and 'example.js' under 'TYPEDSCRIPT-INTRO'. The main editor area displays the following code:

```
function addShipping(price, shipping) {
    console.log(price + shipping);
}

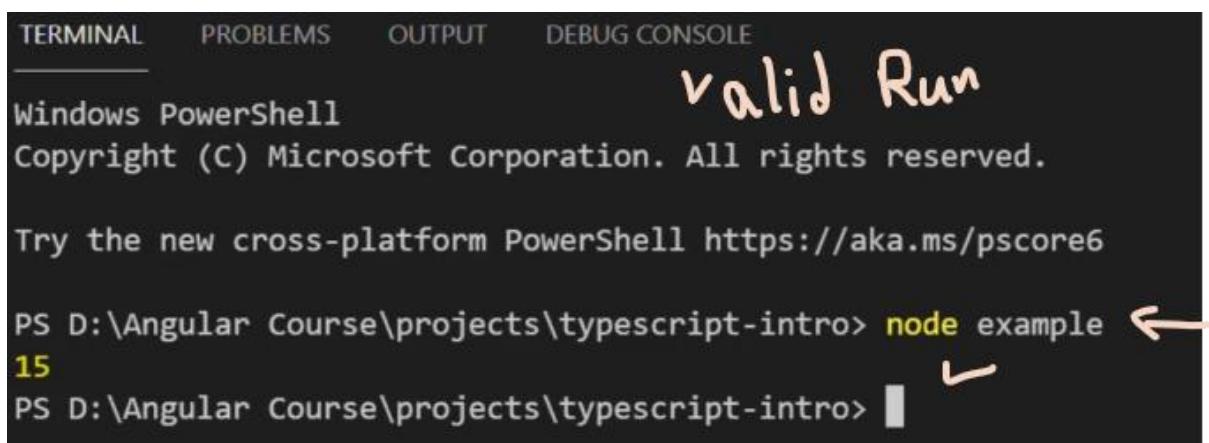
addShipping(10, 5);
```

Below the editor is the Terminal panel, which shows the output of running the script:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6
PS D:\Angular Course\projects\typescript-intro> [REDACTED]
```

The status bar at the bottom indicates the file has 5 lines, 20 columns, 4 spaces, and is in UTF-8 encoding.



The screenshot shows the Terminal panel in Visual Studio Code. The terminal output is as follows:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6
PS D:\Angular Course\projects\typescript-intro> node example [REDACTED]
15
PS D:\Angular Course\projects\typescript-intro> [REDACTED]
```

A handwritten note 'valid Run' is overlaid on the terminal window, with arrows pointing to the command 'node example' and the resulting output '15'.

```

2   console.log(price + shipping);
3 }
4
5 addShipping('10', 5);
      ↑
string

```

TERMINAL   PROBLEMS   OUTPUT   DEBUG CONSOLE

```

Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS D:\Angular Course\projects\typescript-intro> node example
15
PS D:\Angular Course\projects\typescript-intro> node example ←
105
PS D:\Angular Course\projects\typescript-intro> ✘ Invalid
Run
In 5 Col

```

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/>

The screenshot shows the main TypeScript website. At the top, there's a navigation bar with links for Download, Docs, Handbook, Community, Playground, and Tools. A search bar labeled "Search Docs" is on the right. The main content area has a blue header with the text "TypeScript is JavaScript with syntax for types." Below this, a paragraph explains that TypeScript is a strongly typed programming language built on JavaScript. A "Try TypeScript Now" button is present, along with a note about "TypeScript 5.8 is now available". On the right, there's a code editor window showing a TypeScript file with an error message: "Property 'name' does not exist on type '{ firstname: string; lastname: string; role: string; }'".

This screenshot shows the "Get Started With TypeScript" page. It features a "Back" button and a "Get Started With TypeScript" title. Below the title, there are two sections: "In your browser via the Playground" and "On your computer via npm". Both sections show a screenshot of the TypeScript playground interface. A note at the bottom says "TypeScript 5.8 is now available".

This screenshot shows the "Download TypeScript" page. It starts with a "Download TypeScript" heading and a note that TypeScript can be installed through three routes: npm, NuGet, or Visual Studio Extension. Below this, there's a section titled "TypeScript in Your Project" with a note about having many projects with different TypeScript versions. The page then splits into two columns: "via npm" and "with Visual Studio". The "via npm" column contains instructions for using Node.js and a command line example: "npm install typescript --save-dev". The "with Visual Studio" column contains instructions for getting TypeScript as a NuGet package and a note about the Manage NuGet Packages window.

## via npm

TypeScript is available as a [package on the npm registry](#) available as `"typescript"`.

You will need a copy of [Node.js](#) as an environment to run the package. Then you use a dependency manager like [npm](#), [yarn](#) or [pnpm](#) to download TypeScript into your project.

```
npm install typescript --save-dev
```

[npm](#) [yarn](#) [pnpm](#)

All of these dependency managers support lockfiles, ensuring that everyone on your team is using the same version of the language. You can then run the TypeScript compiler using one of the following commands:

```
npx tsc
```

[npm](#) [yarn](#) [pnpm](#)

**npm install typescript –save-dev**

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

# TypeScript



TypeScript is a language for application-scale JavaScript. TypeScript adds optional types to JavaScript that support tools for large-scale JavaScript applications for any browser, for any host, on any OS. TypeScript compiles to readable, standards-based JavaScript. Try it out at the [playground](#), and stay up to date via [our blog](#) and [Twitter account](#).

Find others who are using TypeScript at [our community page](#).

## Installing

For the latest stable version:

```
npm install -D typescript
```

For our nightly builds:

```
npm install -D typescript@next
```

Compiling TS to JS in the command line do:

```
npx tsc <file_name.ts>
```

In this example:

```
npx tsc example.ts
```

```
TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE [?] pov

PS D:\Angular Course\projects\typescript-intro> npx tsc example.ts ←
PS D:\Angular Course\projects\typescript-intro> npx tsc example.ts ←
PS D:\Angular Course\projects\typescript-intro>
```

The terminal window shows the following commands and output:

- `PS D:\Angular Course\projects\typescript-intro> npx tsc example.ts`
- `added 1 package, and audited 2 packages in 3s`
- `found 0 vulnerabilities`
- `PS D:\Angular Course\projects\typescript-intro> 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.

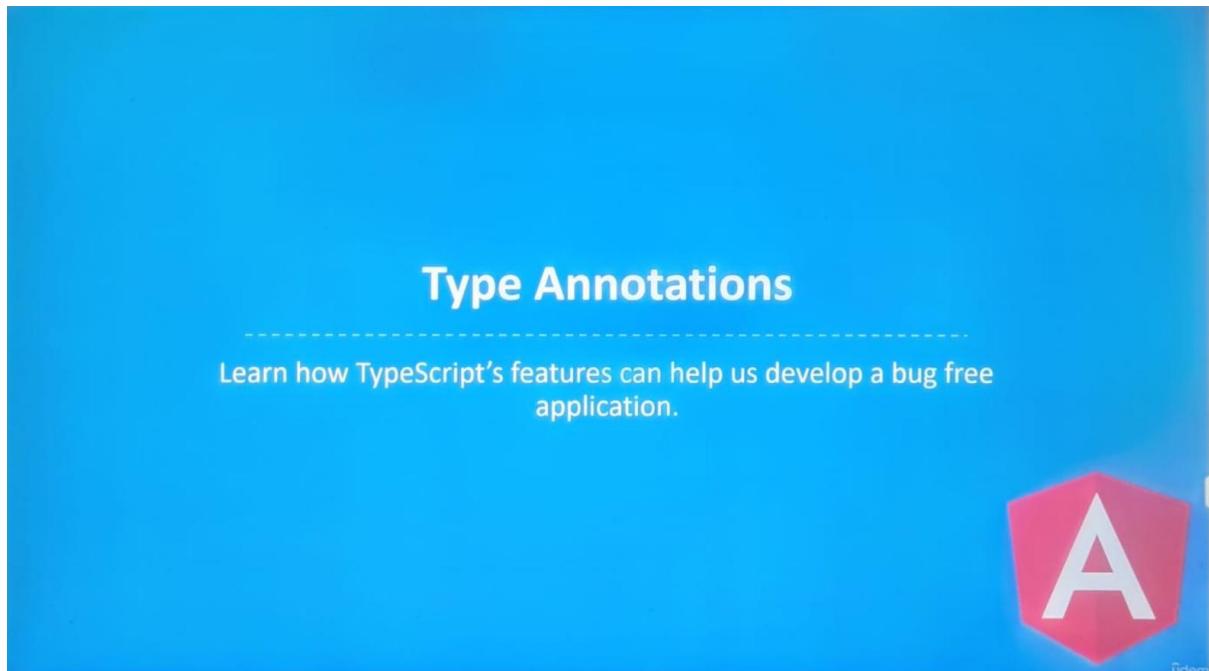
The screenshot shows the VS Code interface. In the Explorer sidebar, there are files: `example.ts`, `example.js`, `node_modules`, `example.js`, `example.ts`, `package-lock.json`, and `package.json`. The `example.ts` file is open in the editor, containing the following code:

```

1 function addShipping(price, shipping) {
2   console.log(price + shipping);
3 }
4 addShipping('10', 5);
5

```

A yellow curly brace highlights the entire function body. A question mark icon is placed at the end of the last line of code.



The screenshot shows the `example.ts` file again. The function definition is highlighted with a pink box. Handwritten annotations in pink and black are overlaid on the code:

```

1 function addShipping(price: number, shipping: number): number | boolean {
2   return price + shipping;
3 }
4 addShipping(10, 5);
5

```

The word "Data Annotation" is written over the parameter types, and "return type" is written over the return type annotation. Arrows point from these handwritten labels to their respective parts in the code.

The screenshot shows the `example.ts` file. The fifth line contains the code `addShipping("10", 5);`. A pink arrow points from the handwritten note "String Argument" to the string value "10". The code is annotated with the following errors:

```

1 function addShipping(price: number, shipping: number): number | boolean {
2   return price + shipping;
3 }
4 addShipping("10", 5);
5

```

The first two lines of the function have a red underline, and the line `addShipping("10", 5);` has a red underline under the string value "10".

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT
PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript> 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>;
~~~~~
example.ts:5:13 - error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
5 addShipping("10", 5);
~~~~~
Found 2 errors in 2 files.

Errors  Files
1  ../../../../../../node_modules/@types/node/module.d.ts:224
1  example.ts:5
○ PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript>

```

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

```

example.ts 1, 0    example.js 0 ✘
ZTM > TypeScript > example.js > ...
1  function addShipping(price, shipping) {
2  |    return price + shipping;
3  }
4  addShipping("10", 5);
5

```

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
```

2. If problem persists:

```
bash
CopyEdit
npm install --save-dev typescript@5.4
```

3. 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`

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT
PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript> npx tsc example.ts
PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript> ↴
```

```
TS example.ts U X
ZTM > TypeScript > TS example.ts > ...
1  function addShipping(price: number, shipping: number): number | boolean {
2    return price + shipping;
3  }
4
5  addShipping(10, 5);
6
```

```
TS example.ts U JS example.js U X
ZTM > TypeScript > JS example.js > addShipping
1  function addShipping(price, shipping) {
2    return price + shipping;
3  }
4  addShipping(10, 5);
5
```

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](#): `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](#).

## `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](#)

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` 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](#)

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](#)

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
```

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
```

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
```

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
```

## 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](#)

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](#)

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
```

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
```

## 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
```

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
```

## 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](#)

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](#)

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](#)

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**

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 });

```

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;

```

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

```

## 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

```

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	Extending a type via intersections
<pre>interface Animal {   name: string; }</pre>	<pre>type Animal = {   name: string; }</pre>
<pre>interface Bear extends Animal {   honey: boolean;</pre>	<pre>type Bear = Animal &amp; {   honey: boolean;</pre>

```

}

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, {});

```

```

}

const bear = getBear();
bear.name;
bear.honey;

```

### 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](#), 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.
- Using interfaces with `extends` [can often be more performant for the compiler](#) 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
```

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
```

**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
```

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
```

## 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](#)

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](#)

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](#)

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](#)

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
```

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
```

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
```

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
```

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  
const;  
handleRequest(req.url, req.method);  
Try
```

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` 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());
  }
}
```

[Try](#)

## 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](#)

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`:

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

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

Try

You can learn more about BigInt in [the TypeScript 3.2 release notes](#).

### 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

You can learn more about them in [Symbols reference page](#).

```
TS variables.ts U X
ZTM > TypeScript-4.9 > TS variables.ts > InvestmentAccount
1 // string, number, boolean, null, undefined
2 let myName: string | null = null;
3 myName = "test";
4
5 // Arrays
6 let items = [5, "luis"];
7
8 interface IAccount {
9   name: string;
10  balance: number;
11  status?: string;
12  deposit?: () => void;
13 }
14
15 const account: IAccount = {
16   name: "Luis",
17   balance: 10,
18 };
19
20 let accounts: IAccount[];
21
22 class InvestmentAccount implements IAccount {
23   constructor(public name, public balance) {}
24
25   private withdraw() {}
26 }
```

TS variables.ts

```

ZTM > TypeScript-4.9 > TS variables.ts > InvestmentAccount
1 // string, number, boolean, null, undefined
2 let myName: string | null = null;
3 myName = "test";
4
5 // Arrays
6 let items = [5, "luis"];
7
8 interface IAccount {
9   name: string;
10  balance: number;
11  status?: string;
12  deposit?: () => void;
13 }
14
15 const account: IAccount = {
16   name: "Luis",
17   balance: 10,
18 };
19
20 let accounts: IAccount[];
21
22 class InvestmentAccount implements IAccount {
23   constructor(public name, public balance) {}
24
25   private withdraw() {}
26 }

```

? means optional

Union

object

properties

const account:

{  
name: string,  
balance: number  
} = {  
name: 'Luis',  
balance: 10  
}; object

JS variables.js

```

ZTM > TypeScript-4.9 > JS variables.js > ...
1 // string, number, boolean, null, undefined
2 var myName = null;
3 myName = 'test';
4
5 var items = [5, 'luis'];
6
7 var account = {
8   name: 'Luis',
9   balance: 10
10 };
11 var accounts;
12 var InvestmentAccount = /** @class */ (function () {
13   function InvestmentAccount(name, balance) {
14     this.name = name;
15     this.balance = balance;
16   }
17   InvestmentAccount.prototype.withdraw = function () {
18   };
19   return InvestmentAccount;
20 })();
21

```

## Generics

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

```
ts generics.ts U X
ZTM > TypeScript-4.9 > generics.ts > ...
1  class Queue<T> {
2    private data: T[] = [];
3
4    add(item: T) {
5      this.data.push(item);
6    }
7
8    remove() {
9      this.data.shift();
10 }
11
12
13 const nameQueue = new Queue<string>();
14 nameQueue.add('luis');
15 nameQueue.add('andrei');
16
17 const numberQueue = new Queue<number>();
18 numberQueue.add(10);
```

Data Type  
should be  
flexible.  
Each Queue works  
on One Type of Data

**A** Where else can we use generics?

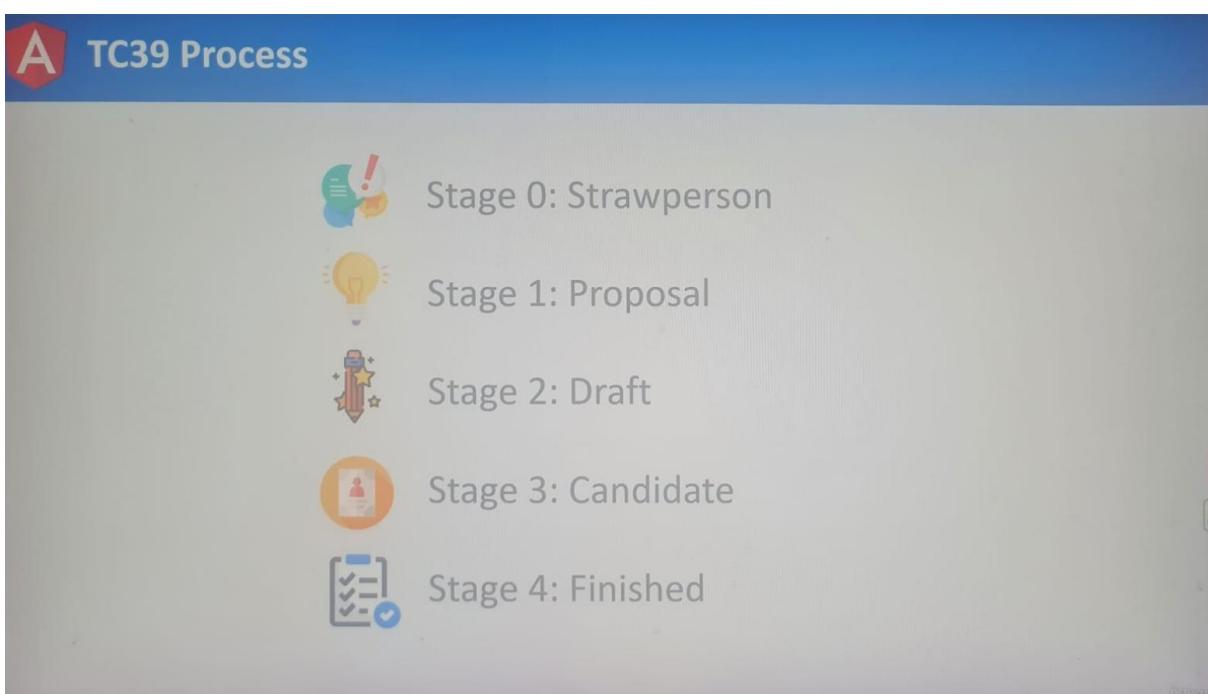
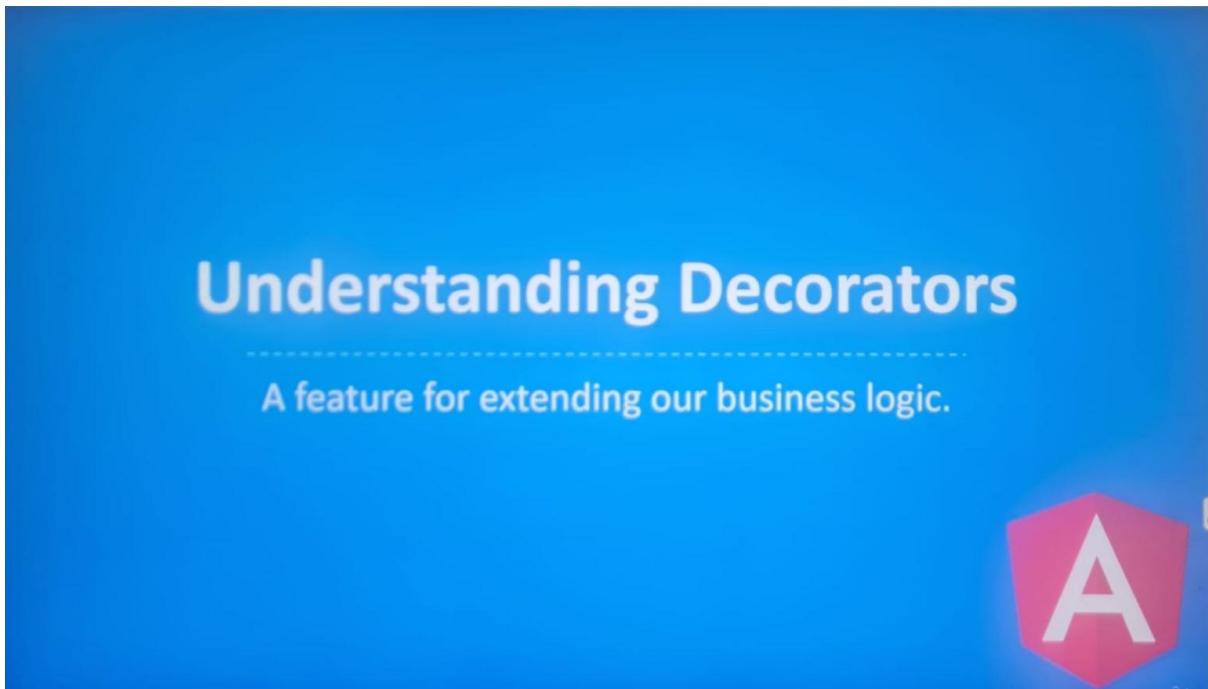
### Functions

```
● ● ●
function someFunction<T>(paramOne: T) : T {
  return paramOne
}
```

### Class Methods

```
● ● ●
class SomeClass {
  someMethod<T>(items: T[]) : T[] {
    return items
  }
}
```

## Decorators



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

Screenshot of a GitHub repository showing a table of proposals in Stage 2. An orange arrow points to the 'Last Presented' column for the 'Decorators' proposal.

Proposal	Author	Champion	Last Presented
<code>function.sent</code> metaproxy	Allen Wirfs-Brock	HE Shi-Jun	July 2019
Decorators	Daniel Ehrenberg	Yehuda Katz Brian Terlson Daniel Ehrenberg Chris Garrett	July 2021
<code>throw expressions</code>	Ron Buckton	Ron Buckton	January 2018
Function implementation hiding	Domenic Denicola Michael Ficarra	Michael Ficarra	June 2020
New Set methods	Michał Wadas Sathya Gunasekaran	Sathya Gunasekaran	January 2019
Sequence properties in Unicode property escapes	Mathias Bynens	Mathias Bynens	October 2019
collection normalization	Bradley Farias	Bradley Farias	January 2019
<code>Array.isTemplateObject</code>	Mike Samuel, Krzysztof Kotowicz	Krzysztof Kotowicz	December 2019



## What are decorators?

Decorators are functions for extending business logic or adding metadata.



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 return the same piece of code, but with the modifications.

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

```
class SomeClass {  
    someProperty = 100  
}  
  
class SomeOtherClass {  
    someProperty = 100  
}
```

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:

## Alternative solutions

- HOC
- Composition
- Closures
- Mixins

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.

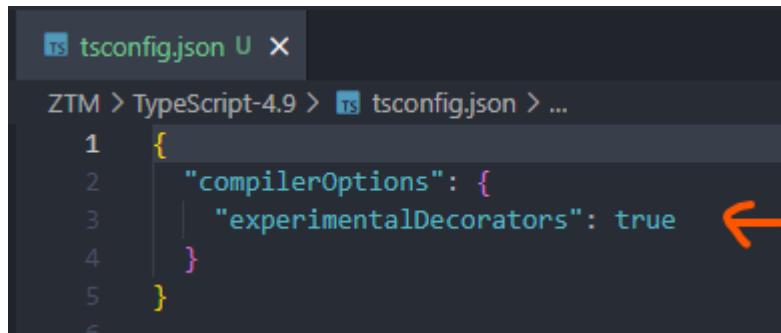
We are going to configure TypeScript.

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:



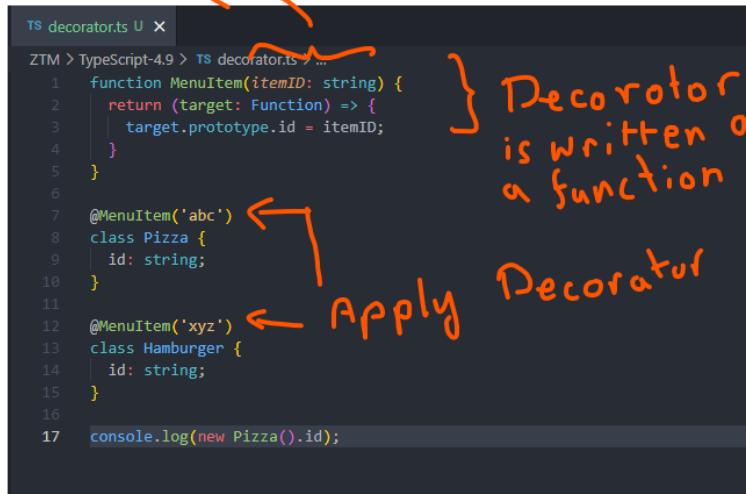
```

1  {
2    "compilerOptions": {
3      "experimentalDecorators": true ←
4    }
5  }

```

## Writing Decorators

2 parameters must be add to Decorators: value, context



```

1  function MenuItem(itemID: string) {
2    return (target: Function) => {
3      target.prototype.id = itemID;
4    }
5  }
6
7  @MenuItem('abc')
8  class Pizza {
9    id: string;
10 }
11
12 @MenuItem('xyz')
13 class Hamburger {
14   id: string;
15 }
16
17 console.log(new Pizza().id);

```

Decorator is written as a function

Apply Decorator

stores the value of the definition it is applied to

contains information of the definition itself

Decorators can be applied to classes, methods or properties

TS decorator.ts X

ZTM > TypeScript-4.9 > TS decorator.ts > `MenuItem` > `<class>`

```

19  function MenuItem(value) {
20    return class extends value {
21      id = "abc";
22    };
23  }
24
25  @MenuItem
26  class Pizza {
27    id: string;
28  }
29
30  @MenuItem
31  class Hamburger {
32    id: string;
33  }
34
35  console.log(new Pizza().id);
36

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT

- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udem\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> npx tsc
- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udem\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> node .\decorator.js
- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udem\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> [ ]

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

TS decorator.ts X

ZTM > TypeScript-4.9 > TS decorator.ts > `MenuItem` > `<function>`

```

1  function MenuItem(itemID: string) {
2    return function (value) {
3      return class extends value {
4        id = itemID;
5      };
6    };
7  }
8
9  @MenuItem("abc")
10 class Pizza {
11   id: string;
12 }
13
14 @MenuItem("xyz")
15 class Hamburger {
16   id: string;
17 }
18
19 console.log(new Pizza().id);
20

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PLAYWRIGHT

- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udem\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> npx tsc
- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udem\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> node .\decorator.js
- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udem\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> [ ]

OR:

```
TS decorator.ts U X
ZTM > TypeScript-4.9 > ts decorator.ts > Hamburger
1  function MenuItem(itemID: string) {
2    return (target: Function) => {
3      target.prototype.id = itemID; abc
4    };
5  }
6
7  @MenuItem("abc")
8  class Pizza {
9    id: string;
10 }
11
12 @MenuItem("xyz")
13 class Hamburger {
14   id: string;
15 }
16
17 console.log(new Pizza().id);
18
```

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    PLAYWRIGHT

- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> npx tsc
- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9> node .\decorator.js abc ←
- PS C:\Users\kandd\OneDrive\Documents\FullStack\Udemy\HTML-CSS-JS-TS\code\ZTM\TypeScript-4.9>