**TypeScript**

**TypeScript:** It is a programming language that builds upon JavaScript by adding useful features to make your code easier to write, maintain, understand and error free as it provides type safety.

**TypeScript = JavaScript + Types + Error Free (Type Safety)**

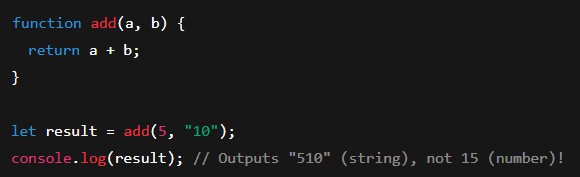
Key Features:

* **Adds type**s: In TypeScript, you can specify what kind of values your variables and functions should have, such as numbers, text, or objects.
* **Helps prevent errors**: By setting these types, TypeScript can catch mistakes before your code runs, saving you from bugs.

The Problem in JavaScript:

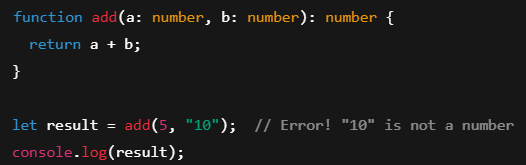
* In JavaScript, variables and functions don't have a fixed type, which can lead to unexpected problems.

Example: You might expect a number, but accidentally get a string or another type, causing bugs in your program.



How TypeScript Helps:

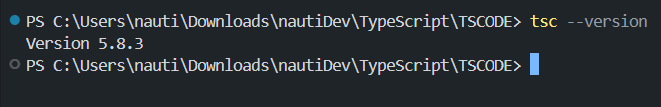
* TypeScript checks your types before the code runs, preventing these kinds of issues and making your code more reliable.



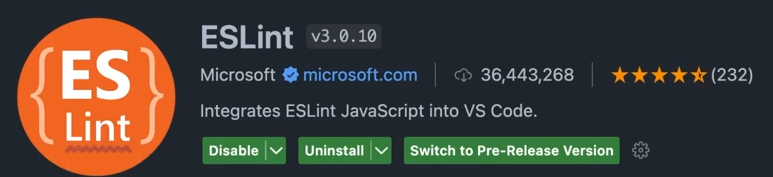
It helps you catch mistakes early, making your code cleaner and easier to understand. Finally, even though you write in TypeScript, the code gets converted into JavaScript because browsers and environments only understand JavaScript.

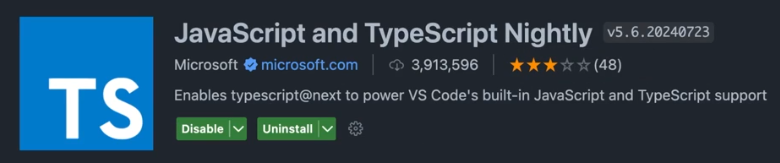
# **Installation Part**

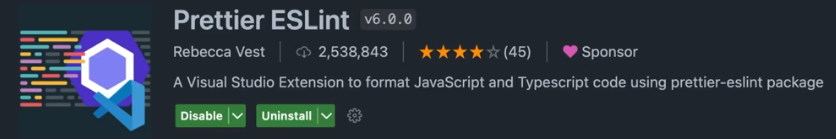
1. Install NODE js
2. npm i typescript -g

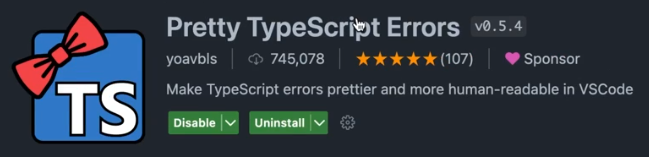


1. Install the following, VS Code Extension:





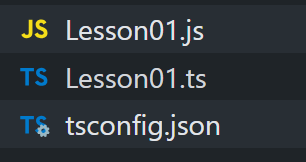




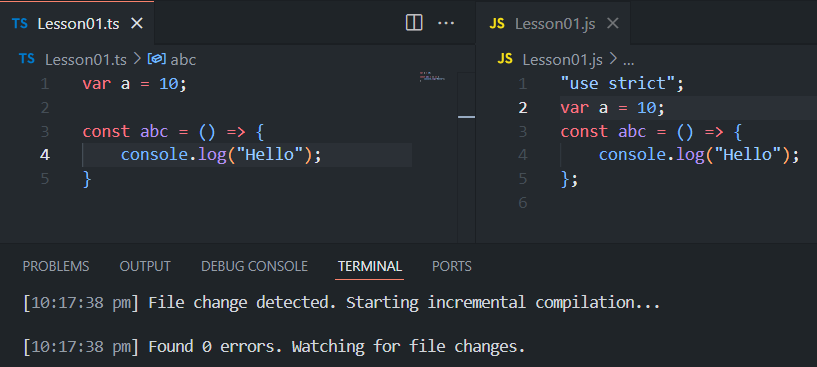
1. Now, create “**tsconfig.json**” file using the command: tsc –init or npx tsc –init

Now, create a Lesson01.ts file and write some code in it and then to run it: tsc Lesson01.ts, this will generate a JavaScript File.

Note: Even though you write in TypeScript, the code gets converted into JavaScript because browsers and environments only understand JavaScript.



**Note**: Open One terminal to compile each “ts file” using the command: "tsc --watch" Open Second terminal, and write "node Lesson01.js" to run the JavaScript file.



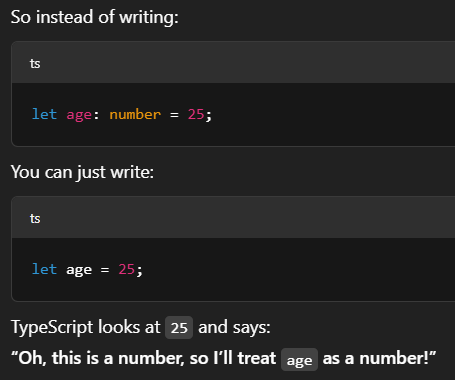
**1. Basic Types**

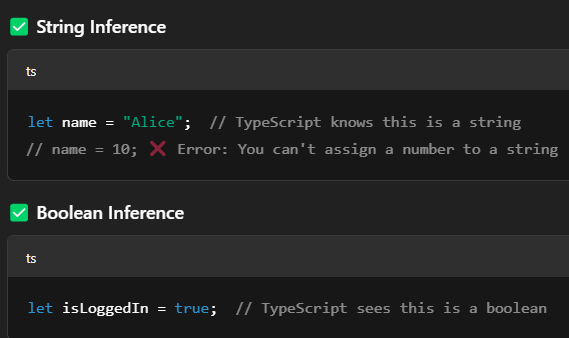
* **Number**: Represents numbers [Primitive]
* **String**: Represents text [Primitive]
* **Boolean**: Represents true or false. [Primitive]
* **Array**: A collection of items of the same type [Reference]
* **Tuple**: An array with fixed types and length [Reference]
* **Enum**: A way to define a set of named constants. [Reference]
* **Any**: Can represent any value (use with caution). [Avoid]
* **Void**: Used for functions that don’t return anything.
* **Null & Undefined**: Represents absence of value.

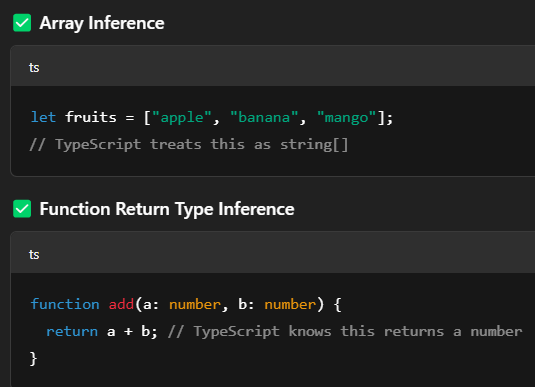
**2. Type Inference**

* TypeScript can often automatically determine the type of a variable based on the assigned value, so you don’t always need to specify it explicitly.

TypeScript is smart, when you give a variable a value, it can **guess the type** of that value without you having to say it.





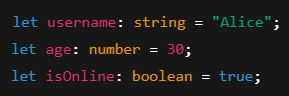


Why it is useful:

* It saves you time — less typing.
* Your code is still safe and type-checked.
* You can still **explicitly define types** when needed.

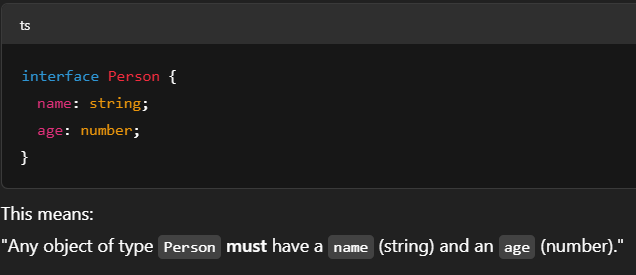
**3. Type Annotations**

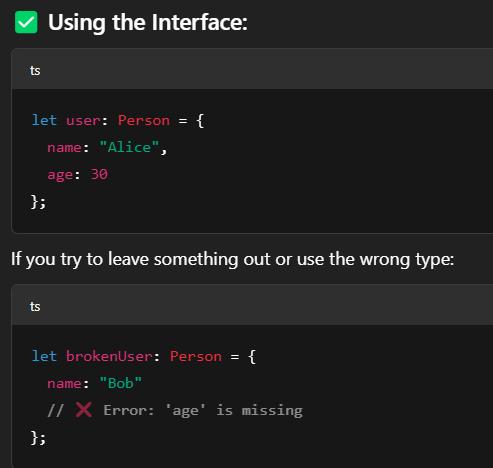
* You can explicitly specify the type of variables, function parameters, and return types.



**Interfaces**

An interface like a blueprint for an object [**Object ki shakal**]. It tells TypeScript: "This object should have these properties, with these types."

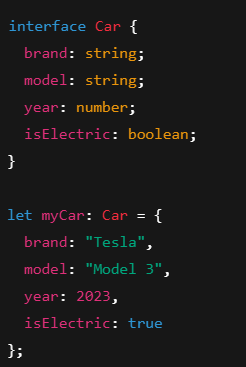


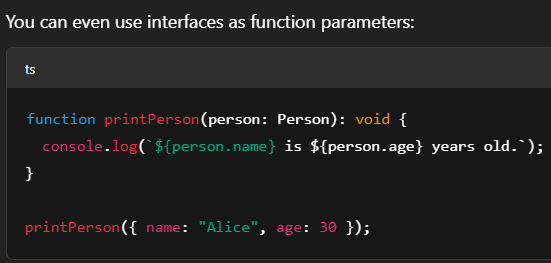


**🚀 Why Use Interfaces?**

* They **help organize your code**
* They make sure your objects are **consistent**
* TypeScript will **warn you** if something is missing or wrong.

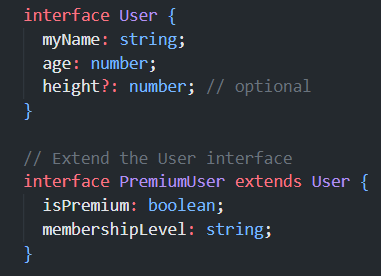
Note: Interface can be used to define complex types and ensure objects adhere to a specific structure.





# **Extending Interfaces**

It is a powerful way to **reuse** and **build on existing structures**.

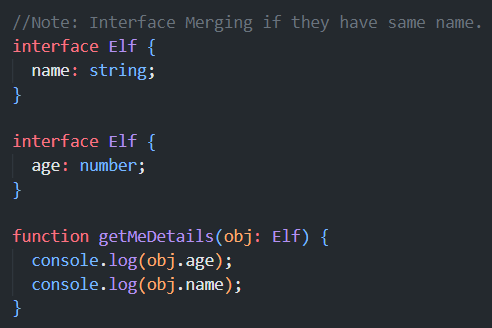


# **Interface Merging**

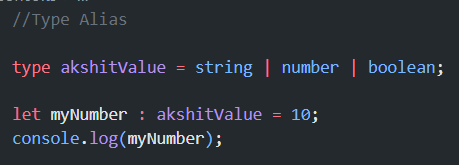
In TypeScript, if you declare multiple interfaces with the same name, TypeScript automatically combines (merges) them into one. This is only possible with interfaces, not with types or classes.

**⚠️ Note:**

* This is mostly used in libraries or large codebases where interfaces might be defined in different files.
* It can be **confusing** if you're not aware it's happening, so use it **carefully**.

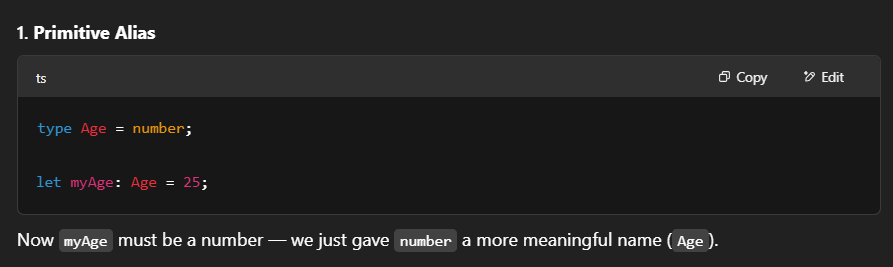


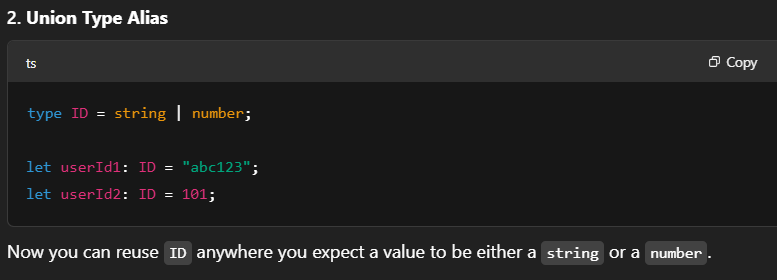
**# Type Aliases:** A **Type Alias** lets you **create a custom name** for any type: primitive, union, array, object, etc.

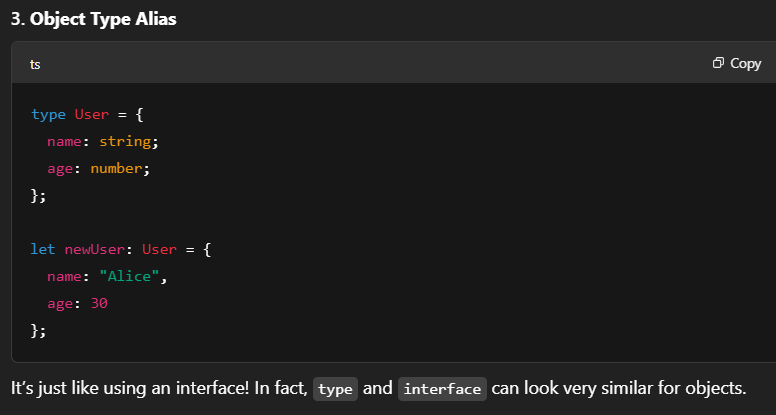


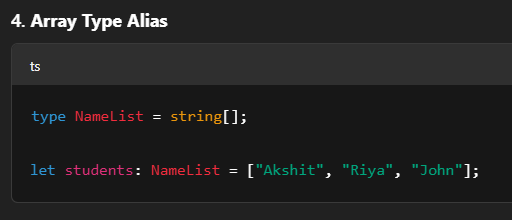
Note: There is no Type Merging as you can do in Interface.

**# Types of Type Aliases**



****

****

****

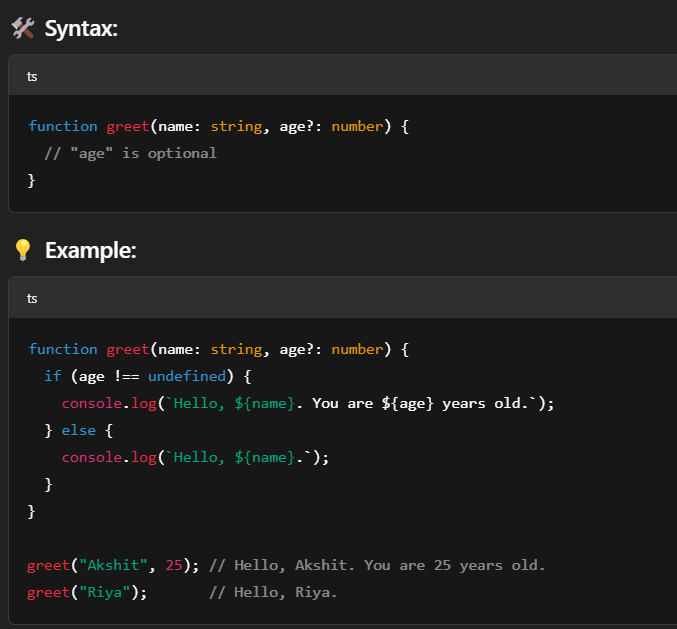
****

**Note:**

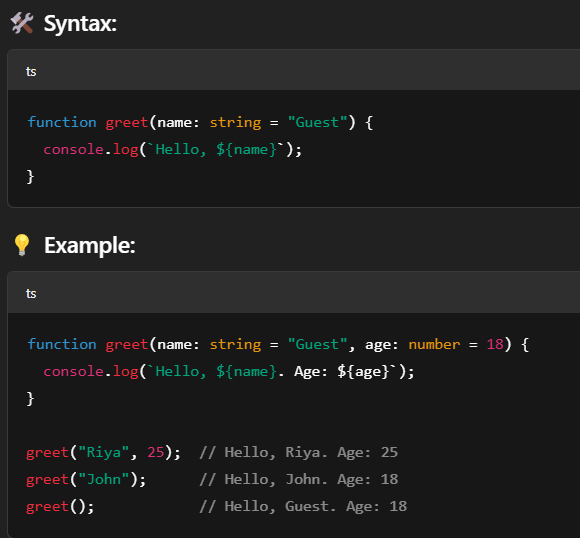
* Use interface for **object shapes** (like defining what a "User" looks like).
* Use type when you want to name **anything else**: unions, arrays, functions, etc.
* type is a **bit more powerful** and **flexible**, but both are useful and often interchangeable for objects.

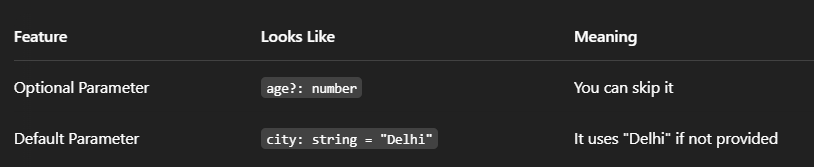
**# Optional & Default Parameters**

* Functions can have **optional parameters**. Optional parameters are **not required** when calling a function. If you don’t pass them, they will be undefined.



* You can also provide **default values** for parameters. Default parameters have a value **automatically assigned** if you don’t provide one.

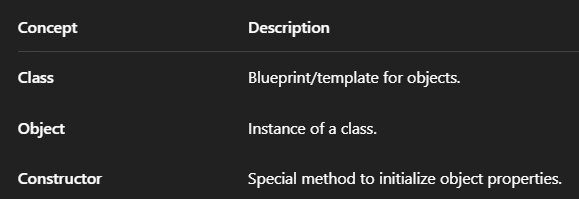


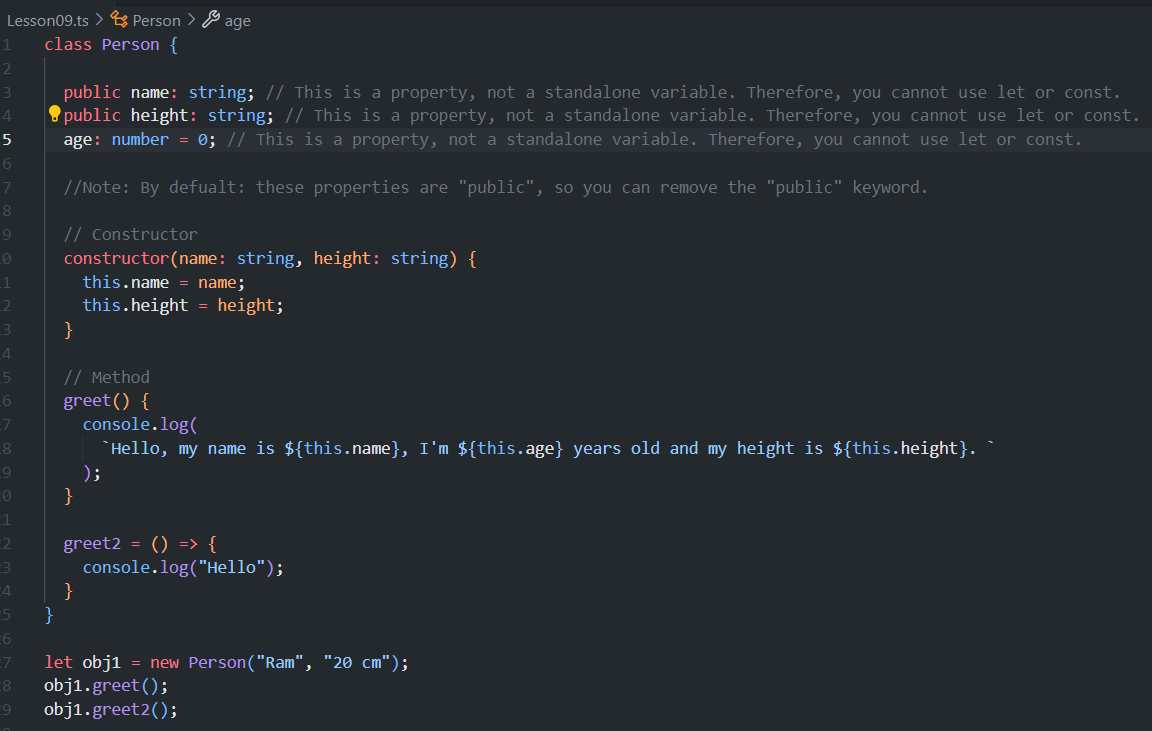


**Classes And Objects**

TypeScript supports **OOP** (Object-Oriented Programming) with **classes**, **constructors**, **methods**, and **inheritance**.

* **Class**: A **class** is like a blueprint for creating objects. It defines properties (variables) and methods (functions) that the objects created from it will have.
* **Object**: An **object** is an **instance** of a class. It’s created using the new keyword.
* **Constructor**: A **constructor** is a special method that runs when you create a new object. It’s used to initialize the properties of the object.



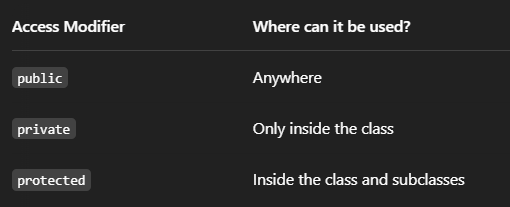


**Access Modifiers**

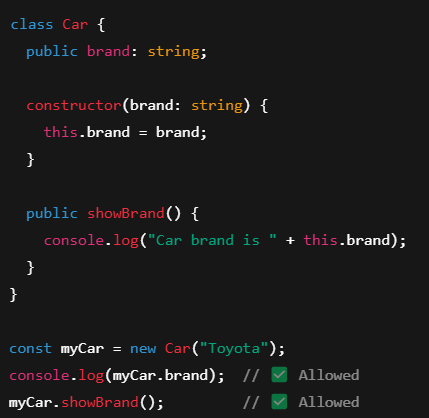
**Access modifiers** control **who can access** the **properties (Class Members or Class Variables)** and **methods** in a class.

Think of a class like a **house** 🏠:

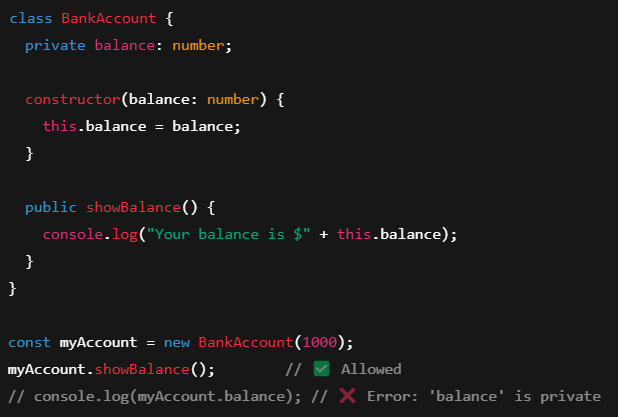
* Some rooms are open to everyone (like the living room) — public
* Some are locked — only you can go in (like your bedroom) — private
* Some are shared with family only (like the kitchen) — protected



* Public: Public means anyone can access it. A public property or method can be used from inside or outside the class. It is the default type. You don’t need to write public every time. Use it when you want something open to all parts of your code.



* Private: Private means the property or method is only for the class itself. You cannot use it from outside the class. It is good for keeping things safe or hidden. For example, a bank account class can have a private balance. This helps protect important data from being changed directly.



* Protected: Protected means only the class and its child classes can use the property or method. It is like private, but shared with subclasses. Use it when you want to keep things hidden from outside, but still allow child classes to access and use them when they extend the parent class.

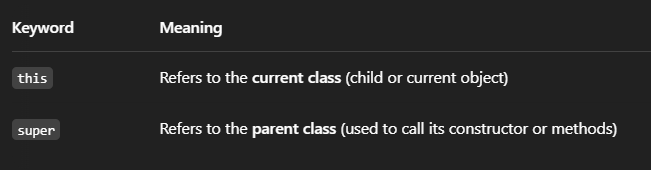


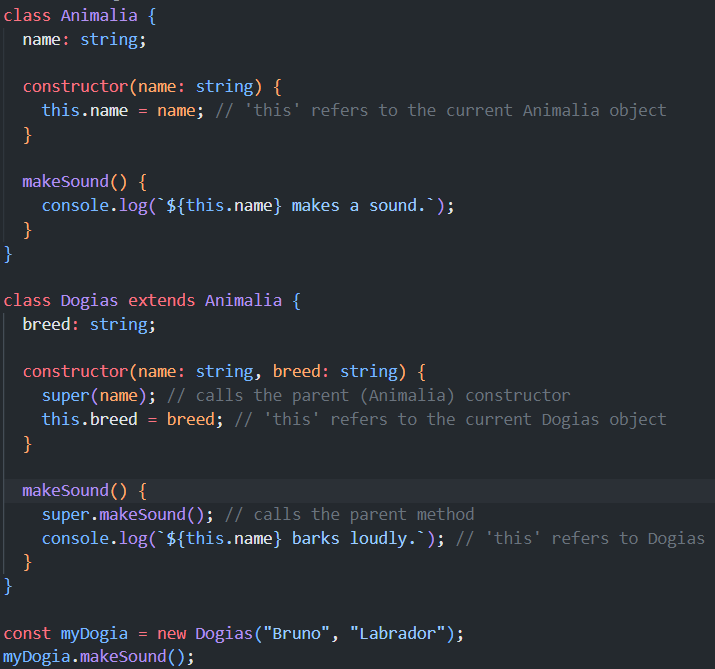
**Inheritance**

**Inheritance** means one class (called a **child** or **subclass**) can **reuse** the features of another class (called a **parent** or **superclass**). It helps us avoid repeating code and build relationships between classes.

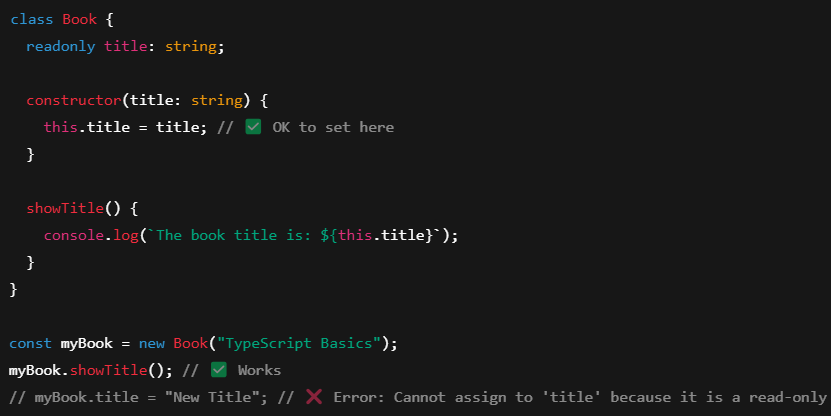


# **this vs super keywords**



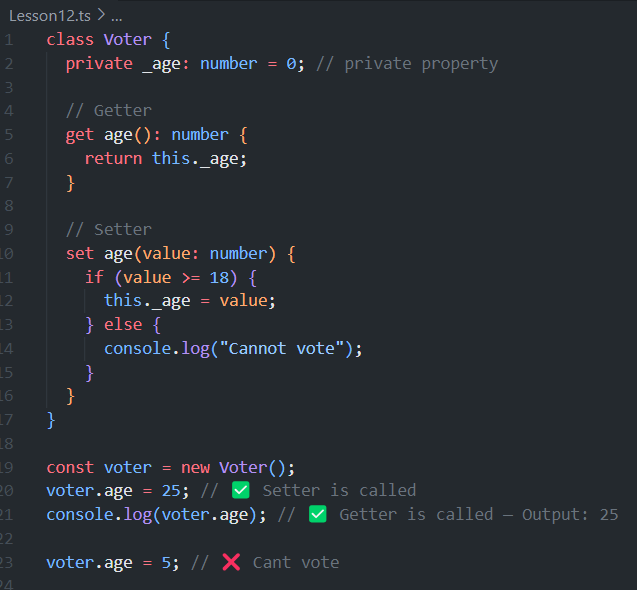


Note: **readonly**: It means the **value cannot be changed** after it’s set — like a **constant property** inside a class. You can assign the value **once** (usually in the constructor), and after that, it’s **locked**.



# **Getters And Setters**

* **Getter (get)**: Used to **read** a property.
* **Setter (set)**: Used to **change** a property **with some control or logic**.

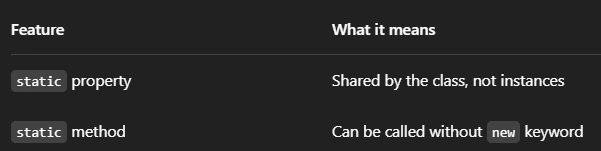


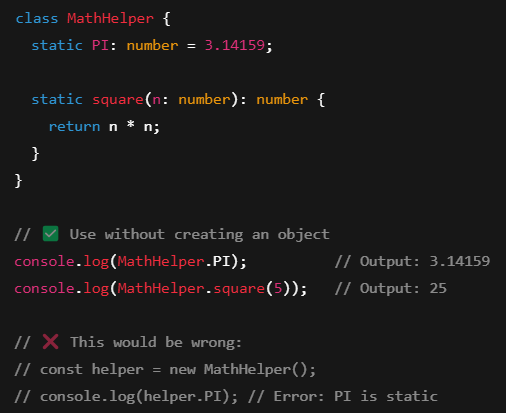
**⚠️ Note:**

* The actual variable is often named with an underscore: \_age.
* You **don’t call** the getter/setter like a function (voter.age ✅ not voter.age() **X**).

# **Static keyword**

* static means the **property or method belongs to the class itself**, **not to the object**.
* You can use it **without creating an object**.
* Static members are useful for **utility functions**, **constants**, or anything **shared across all objects**.





**# Abstract Class**

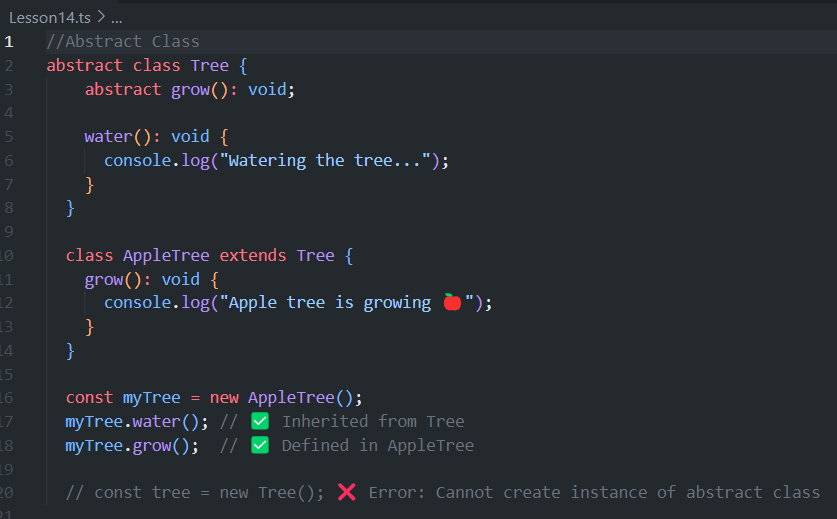
* An **abstract class** is like a **blueprint**.
* You **can’t create an object** from it.
* It’s meant to be **inherited** by other classes.

Example: A Tree is an abstract class but an apple tree is its implementation. **Tree** is a **general idea**, you don’t grow a “generic” tree in real life.

But you **can grow** specific trees like: **Apple Tree**, **Mango Tree** and **Banana Tree**. These are the **real implementations** of the idea called Tree.

# **Abstract Method**

* An **abstract method** is a method that has **no body** (no code).
* It just says: “All child classes **must** write their own version of this method.”



**Functions**

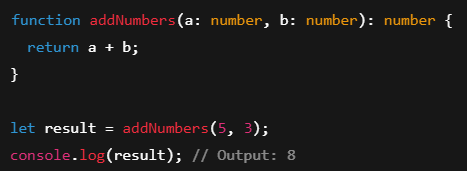
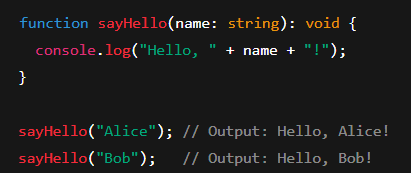
**Functions**: A **function** is like a **mini-program** inside your code. It helps you **organize**, **reuse**, and **run** code when needed.

You give a function:

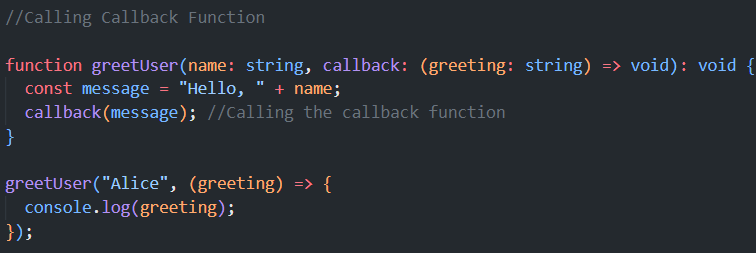
* A **name**
* Some **input** (optional)
* And it gives you **output** (optional)

Why use functions**:**

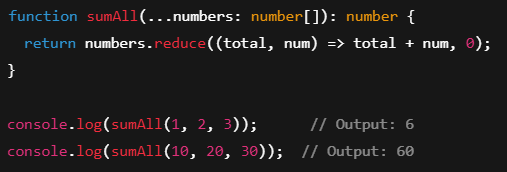
* Avoid repeating the same code
* Make code easier to understand
* Break big tasks into smaller parts



A **callback** is a **function that you pass to another function**, and it gets **called later**.

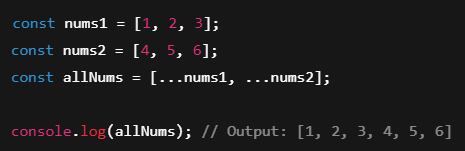


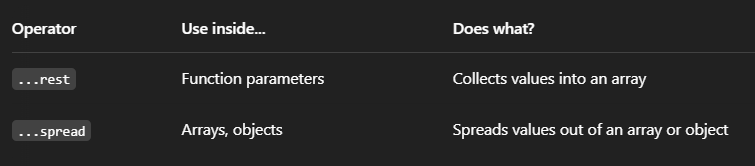
# REST Operator: It is used to **gather** multiple values into a single variable (usually inside functions).



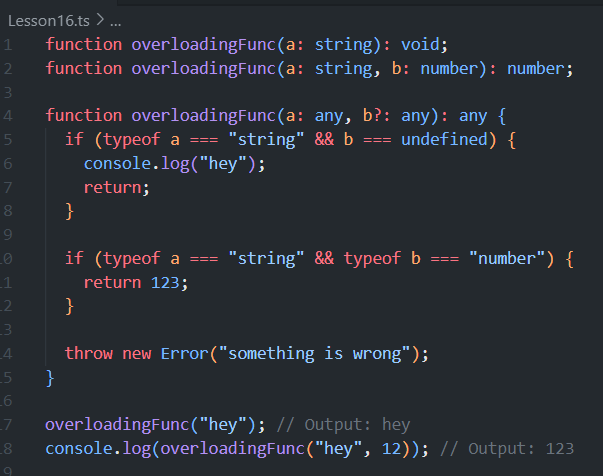
...numbers: number [] means: “Take any number of numbers and put them in an array called numbers.”

# SPREAD Operator: It is used to **spread** or **unpack** items from arrays or objects.





# Function Overloading: You can have **multiple ways to call the same function** with **different input types** or numbers of arguments.



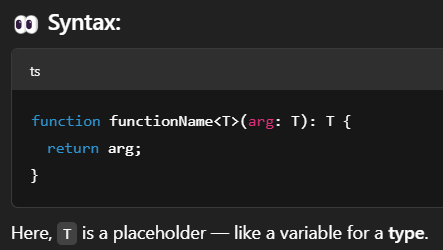
**Generics**

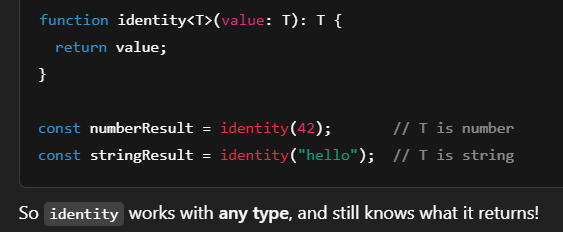
Imagine you’re making a **box**. Sometimes you want to store **toys**, sometimes **books**, sometimes **clothes**. Instead of making a separate box for each, you create **one flexible box** where you can **say what type of item you're putting in**.

That’s what **Generics** do. They let you write code that works with **any type**, but still keeps the **type safety**.

**Generics**: Allow you to create reusable functions and classes that work with any type.

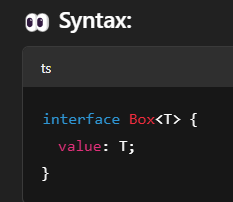
✅ **Generic Function**

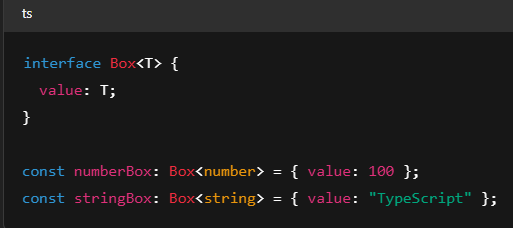




✅ **Generic Interface**

You can use generics in interfaces to define the **type of data** the interface will work with.

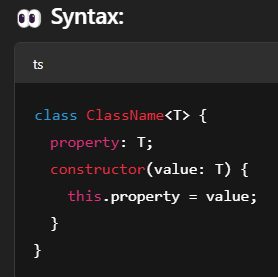




The interface Box works with **any type**, thanks to <T>.

✅ **Generic Class**

A class can also be generic to handle **different types of data**.

This way, Container class works with any type (number, string, etc.).

**🔁 Real World Analogy**

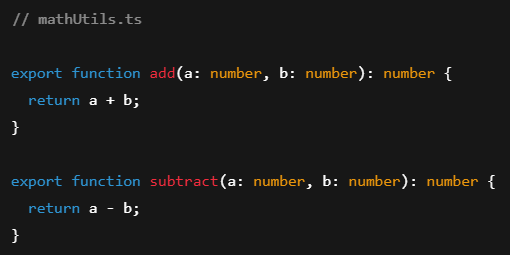
Think of a **reusable shopping bag**. You can carry apples 🧺🍎, books 🧺📚, or clothes 🧺👕. You don't need a new bag for each item — the **same bag works**, but you just tell it what you're carrying. That's **Generics**.

**# Modules:** TypeScript supports **modules**. Modules let you split your code into multiple files. You can keep related code in separate files and then import/export them when needed.

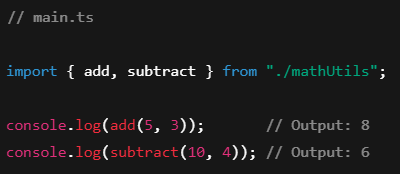
**Think of it like a toolbox**: You keep different tools in different drawers (files), and take out only what you need for a job.

Example:

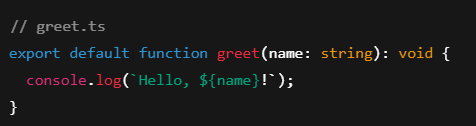
Step 1: Create a file mathUtils.ts (this is a module)

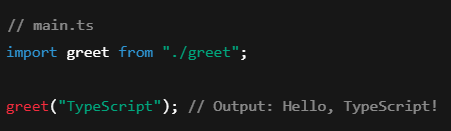


Step 2: Use this module in another file, say main.ts



**export default:** When you use export default, you're saying:"This is the **main thing** I'm exporting from this file."And when you import it, you **don't need to use curly braces {}**.





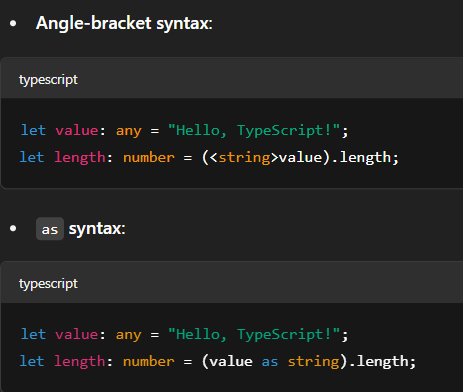
**# Type Casting**

In TypeScript, type casting refers to converting a value from one type to another. There are two main ways to do this: **Type Assertion** and **Type Conversion**.

* 1. **Type Assertion**

Type assertion doesn't change the actual type of the object but tells TypeScript to treat the value as a certain type.

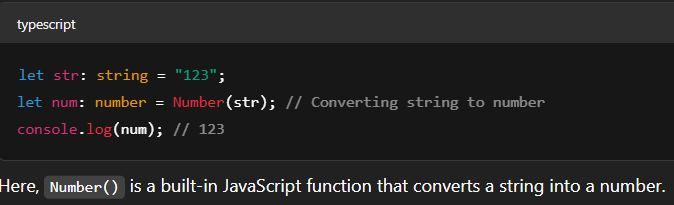
There are two syntax forms for type assertion:



Both ways are functionally equivalent. Here, you're telling TypeScript that value is of type string, so it knows it can safely use string-specific properties like .length.

* 1. **Type Conversion**

Type conversion, unlike type assertion, actually converts the value to a different type.



**Important Notes:**

* Type assertions do not perform any checks, so it's your responsibility to ensure the types are compatible. If you're wrong, it might lead to runtime errors.
* Type conversion actually changes the value, so it ensures the type is valid.

# **Strict Mode**

TypeScript has a **strict mode** to enforce stricter type checks, making your code safer and more reliable.

**TypeScript + React JS**

Now, you have learned the basics of TypeScript! When learning TypeScript with React, here’s a list of topics you can focus on to improve your skills: <https://react-typescript-cheatsheet.netlify.app/docs/basic/setup>

1. **Basic Setup for TypeScript with React using Vite**

Note: Understanding tsconfig.json or tsconfig.app.json settings for React projects.

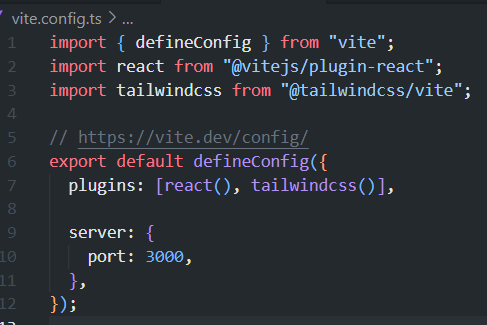
* **target**: This is set to esnext or ES2020, which means the latest ECMAScript version.
* **strict**: Enabling this makes TypeScript check for type safety across your code, ensuring better quality and catching more errors.
* **esModuleInterop**: Ensures that you can import CommonJS modules in a way that works with ES module syntax (important for many third-party libraries).

Note: Vite and Create React App are widely used Build tools to quickly set up React projects, akin to how Maven and Gradle are used for Spring Boot.

1. **Create React App** uses **Webpack** as its bundler. Webpack is responsible for bundling JavaScript, CSS, images, and other assets into optimized files for deployment. Along with Webpack, it also configures **Babel** for transpiling modern JavaScript and JSX into browser-compatible JavaScript. Create React App abstracts these configurations, so developers typically don’t need to manage Webpack or Babel directly and provide as a default setup.
2. **Vite: It** is a modern build tool that uses **esbuild** for fast development and **Rollup** as the bundler for production builds.It also provides a **Development Server** (local server) for serving files during development, typically running on **localhost**.

**Note: Webpack**, **Parcel** and **Rollup** are primarily **bundlers**, while **Vite** is both a bundler and a development server.

**# Vite and TypeScript Integration:** Vite uses **esbuild** internally for fast builds and TypeScript support. When you create a project using the react-ts template, Vite automatically handles the TypeScript compilation and the JSX transformation.



This file uses the Vite plugin for React (@vitejs/plugin-react), which automatically enables the TypeScript support for .tsx files in your React components.

**Key Points:**

* **Hot Module Replacement (HMR)**: Vite automatically enables HMR (hot reloading) during development, making React + TypeScript development fast and efficient.
* **Fast Compilation**: Vite leverages **esbuild**, which is faster than traditional bundlers like Webpack.
* **Automatic Type Checking**: Vite uses TypeScript’s built-in type checking when you run the development server.

1. **TypeScript and JSX**

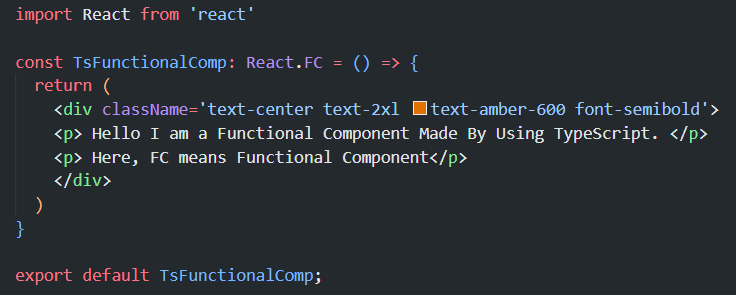
**Syntax**: In React, .jsx files are typically used to write JSX (JavaScript XML) syntax, which is a syntax extension for JavaScript that looks similar to HTML. TypeScript introduces the “.tsx” extension to JSX files. The key difference is that .tsx files allow you to combine JSX with TypeScript types, ensuring type safety for React components.

TypeScript files that support JSX syntax. These files allow you to use TypeScript's static type system, enabling better development experiences, such as autocompletion, type inference, and compile-time checks for bugs in your components.

1. **React Components with TypeScript**

# Creating functional components and defining props.

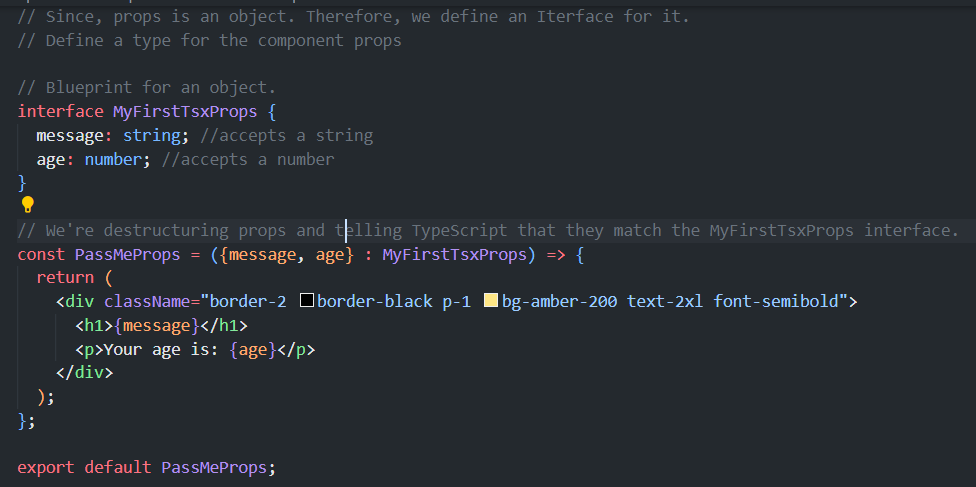
Functional Component: Add React.FC to your Arrow Functional component.



Now, it is must to return something.

Q-What is props?

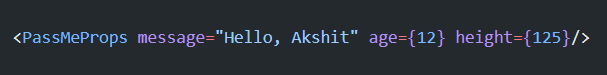
* **"Props"** is short for **properties** [Object].
* In React, **props are how you pass data into a component**.
* Imagine components like functions—props are like arguments you send in.
* Props let you **customize** a component's content or behavior.

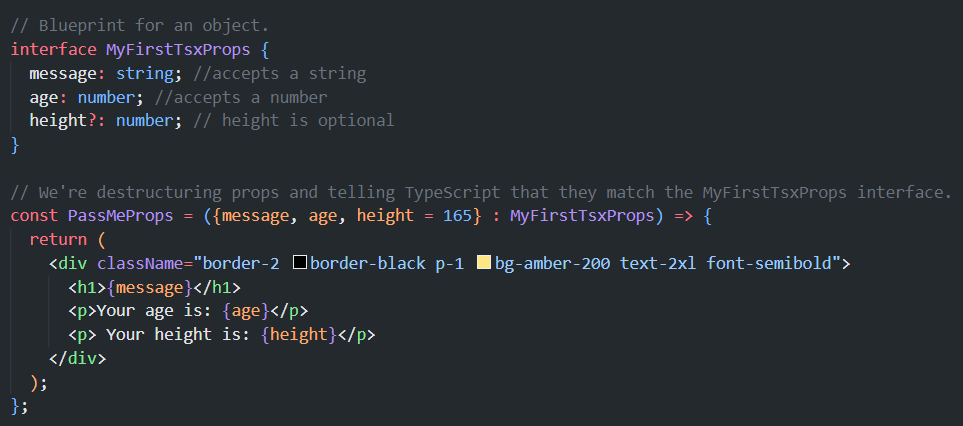




**Optional Props and Default Values**

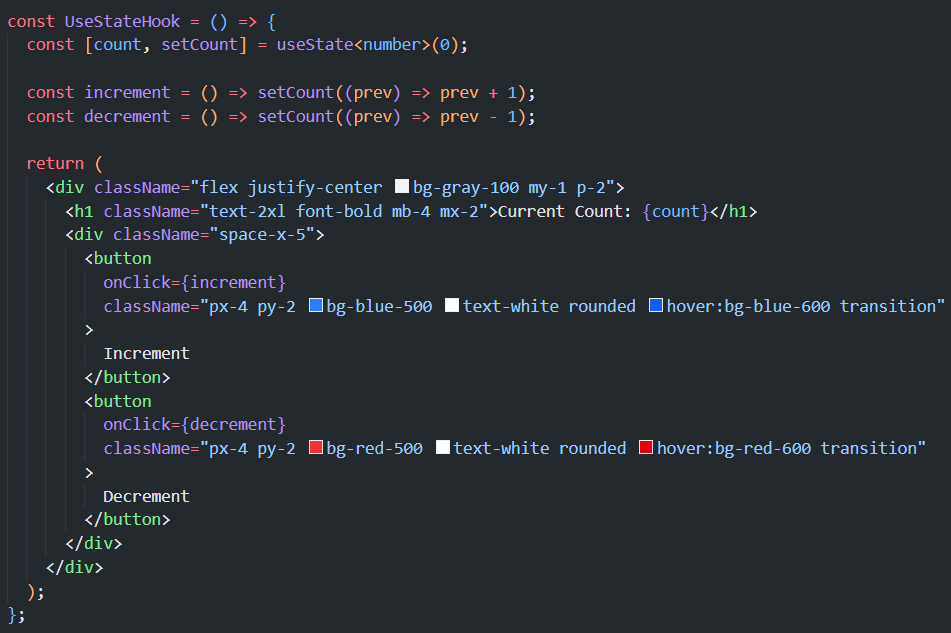
If a prop is optional, you can mark it with a ?. You can also set default values for props.





**useState Hook**

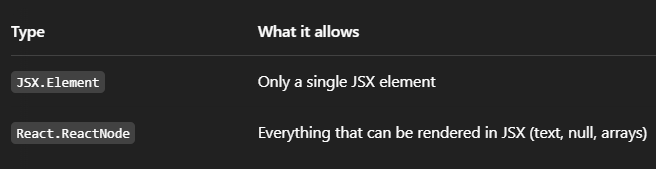
* **Typing the state**: The useState<number> hook is typed explicitly to indicate that count is a number.
* **Event handler**: The increment and decrement functions are simple and don’t need explicit types because React automatically infers their types based on the event, but you can always specify types for more complex handlers.

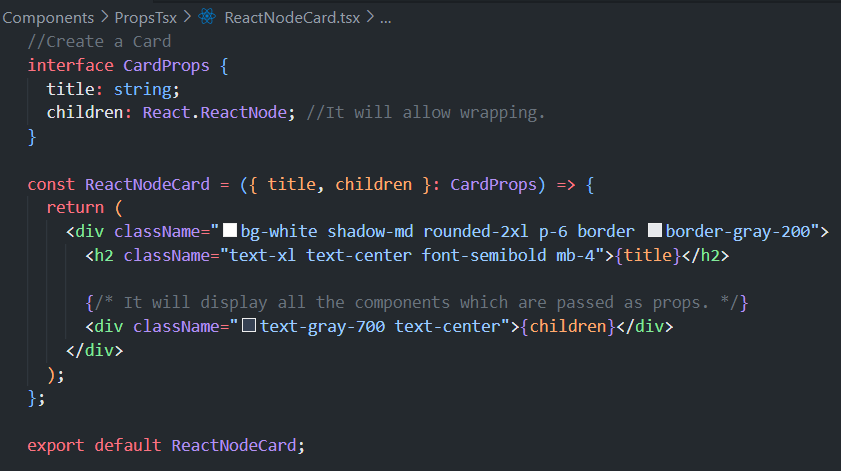


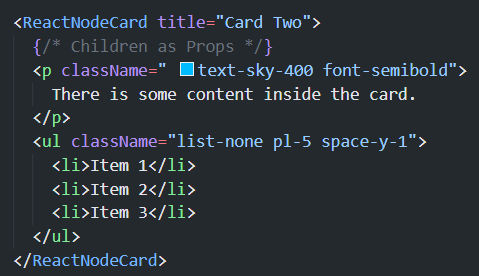
**Typing Children Prop**

**React.ReactNode**: It is a **TypeScript type** that represents **anything you can render inside a React component**.

* Use React.ReactNode for children props.
* It's the most flexible and reflects what React can actually render.





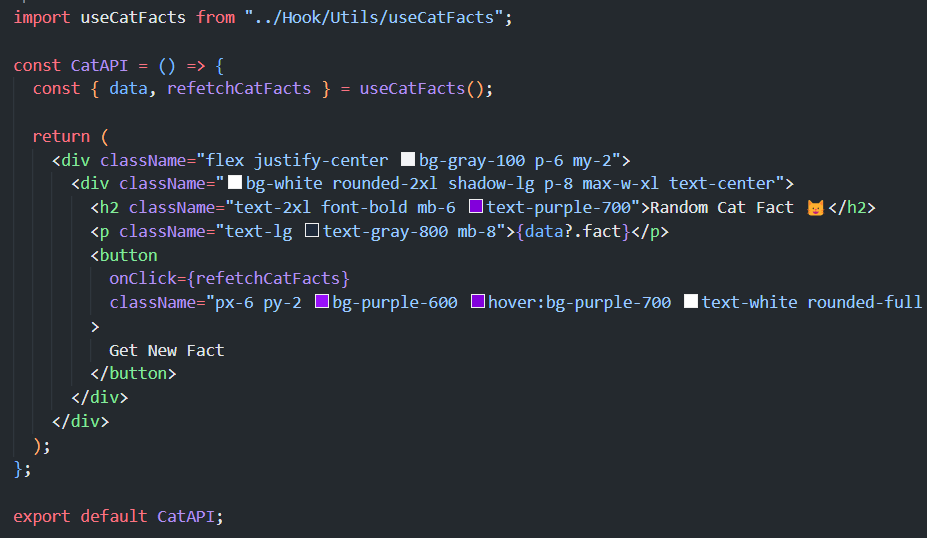


1. **Calling an API using Axios and Custom Hook**

Step 01: npm install axios  
Step 02: Create a Custom Hook

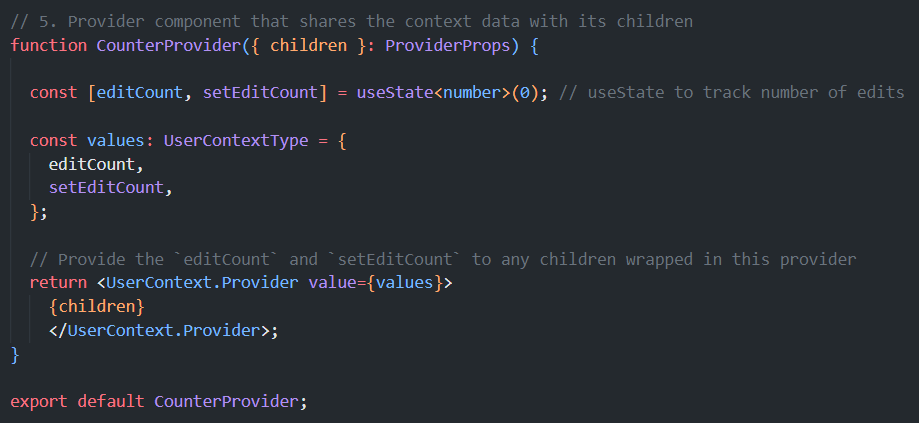


Step 03: Call this Custom Hook



1. **TypeScript and Context API**







1. **TypeScript with Form Libraries**

