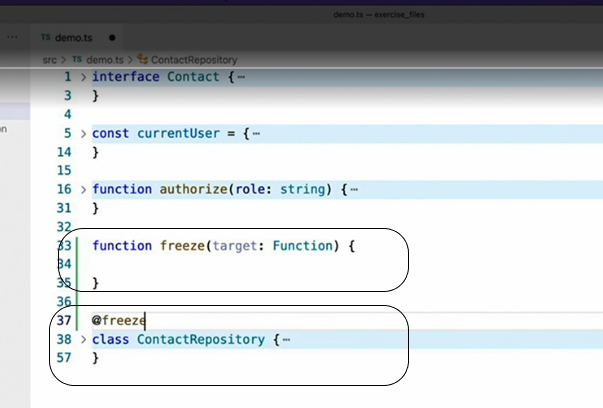
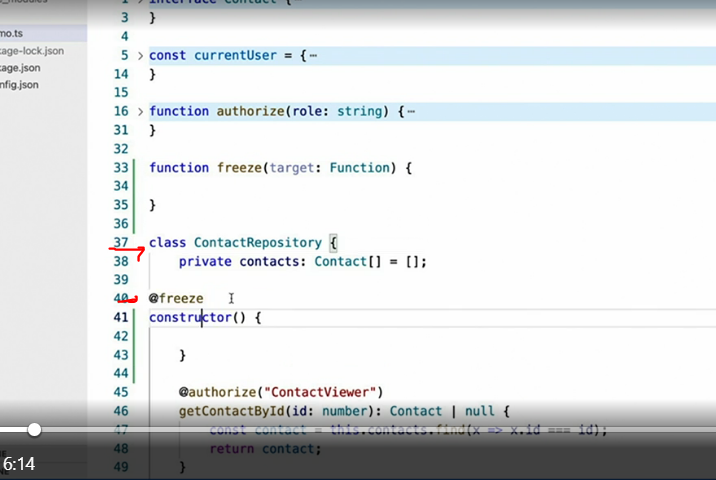
* - [Instructor] Now that I've shown the fundamentals of how decorators and decorator factories work with methods, I'll show you how to **apply decorators to classes.**
* At a high level, the approach is pretty much the same.
* For example*, if I wanted to create a decorator to freeze a class to prevent any changes, I can create a function that accepts a single parameter with the function type like this.*
* (keyboard keys tapping) And we apply the decorator to the class by placing it just before the class definition like this.



* So far, so good.
* It is the same familiar syntax that is used to create method decorators.
* The biggest difference, however, is ***what is actually being decorated.***
* In other words, what exactly is the target that this parameter references? As the function type implies, the target being passed to the class decorator is not the object being created when the class is instantiated.
* Instead, it's *the class's constructor function.*
* In other words, you can think of it more like this.
* (keyboard keys tapping) With that in mind, let's actually change the name of the parameter, just to be really clear what it is.



* (keyboard keys tapping) I can now implement this decorator by calling the object.
* freeze method to freeze the constructor so that it cannot be modified after it's defined.
* (keyboard keys tapping) We'll also want to freeze its prototype as well.
* And that's it.

Graphical user interface, text, application

Description automatically generated

* We've now created a class decorator to modify the class's constructor function.
* To be honest, however, I've never had the need to modify a constructor function with a class decorator.
* Most of the time, I use class decorators to **dynamically add properties or behaviors** to a class at runtime.
* To do that, we take a slightly different approach in our class decorator.
* And instead of modifying the constructor function as it's passed in, we actually return a brand new class.
* It looks like this.
* Let me create a new decorator named singleton.
* (keyboard keys tapping) And for the type of the constructor parameter, I'm going to use any for right now.
* I'll come back to that later.
* Then I'm going to simply use the standard class syntax to define a brand new class that derives from the one that was passed in.
* (keyboard keys tapping) And then I'll simply return that new class.
* And that's it.
* I've now dynamically *redefined the decorated class by creating and returning a whole new class behind the scenes.*

Graphical user interface, text, application

Description automatically generated

* While this new class is technically not exactly the same as the one that was passed in, it does extend the class that was passed in.
* So it still retains all of the original class's members and behaviors.
* Once this code is in place, I can do whatever I want, such as create a static property.
* (keyboard keys tapping) Or add behavior to the constructor, such as checking to make sure that only one instance of this class has been instantiated.
* To do this, I'll add the basic constructor, which calls the base class's constructor via the super method.
* (keyboard keys tapping) But because this decorator can be applied to any class, and we don't know what, if any, parameters that target constructor might have.
* So we'll use the spread operator to accept any and all parameters being passed into the original constructor.
* And pass them right on to the base constructor.
* With this in place, I can finally add my custom logic.
* *First, I'll check to see whether another instance of this class has been created.*
* *(keyboard keys tapping) If another instance has been created, I'll throw an exception.*
* *(keyboard keys tapping) Otherwise I'll save a reference to the newly constructed instance at the end of the constructor, so that I'll be able to check for it next time.*
* (keyboard keys tapping) And that's it, a fully implemented example of a class decorator.

Graphical user interface, text, application

Description automatically generated

* Before we end this video though, let's go back and fix the constructor parameter type, if only because I try to avoid the any type whenever possible.
* The reason I originally converted this to the any type was because the function type just wasn't going to work.
* TypeScript won't let you create a new class that derives from a regular function.
* Instead, we need to use a generic type definition that describes the special behavior of a constructor function, which looks like this.
* (keyboard keys tapping) I won't get into the detailed explanation of this generic type, except to scroll down to where we've used our singleton class decorator.

Graphical user interface, text, application

Description automatically generated

* (keyboard keys tapping) And see that the absence of compilation errors shows that it's exactly the type that TypeScript was looking for.