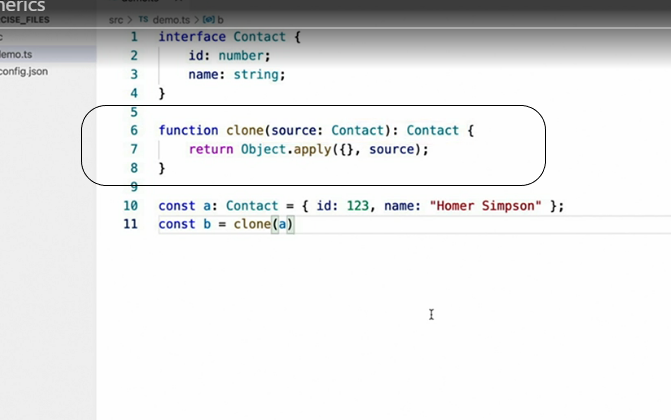
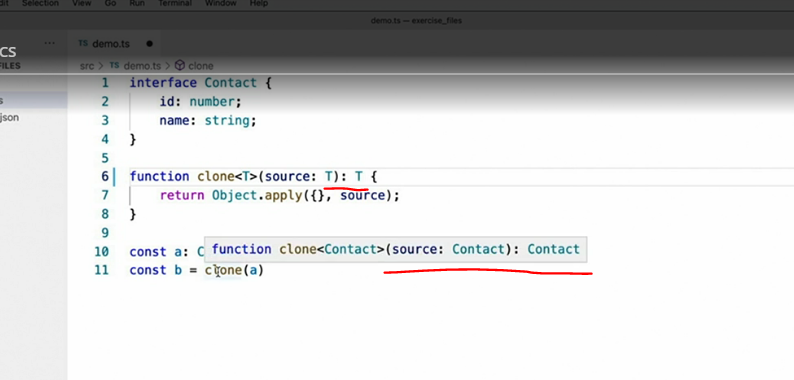
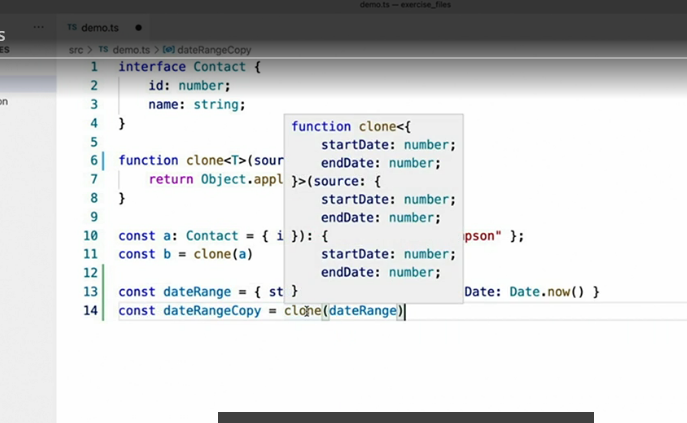
* - [Instructor] In the previous video, I showed you how to apply TypeScript typings to a functions parameter and return values.
* In this video, I'm going to show you how to take that syntax one step further with what might be considered TypeScripts most powerful feature, ***generics***.
* ***A generic type is simply a meta type, a type that represents any other type that you might want to substitute in.***
* It's easiest to understand the generic type and the power it brings by simply showing it in action.
* And that clone function from the previous video is a great candidate for generics.
* Let's take a look and see why.
* In the previous video, in order to give TypeScript enough information to be useful, I had to supply a specific input type and return type.



* In this case, the method accepts a contact parameter and returns a contact value.
* But if we take a look at the actual code in this function, there's nothing specific to a contact at all.
* In fact, this function can take almost any object and return a clone of that same type.
* Enter generics, which allow us to replace both of those references to the contact type with a placeholder type, something that can be substituted for a different type each time the method is used.
* **To convert this to a generic function**, I start by defining the generic type parameter right before the parameter list, like this.
* Here, I've used the generic type name T but this is just a convention that most people use.
* You can use any valid type name.
* Then once I've defined the generic type, I can begin using it to replace any other type in the functions definition, like this.
* *I've now told TypeScript to use the same type for the input parameter, as well as the return value.*
* In other words, whatever type is passed in will get passed right back out.



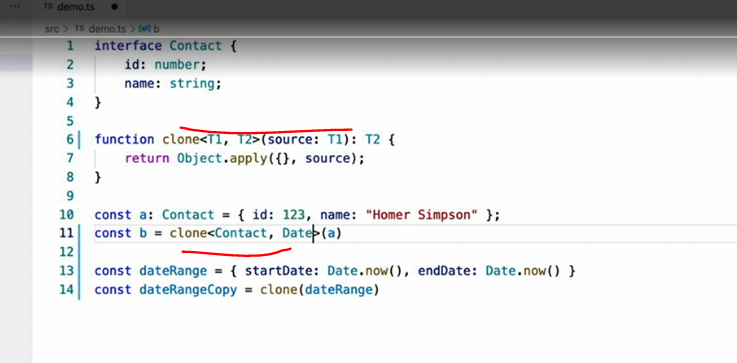
* For example, if I hover over the return value of the existing call that I made to the clone method, I can see that TypeScript continues to correctly determine that a clone of variable A, which is a contact type, should also be a contact but let's try another example just to really see how powerful this actually is.



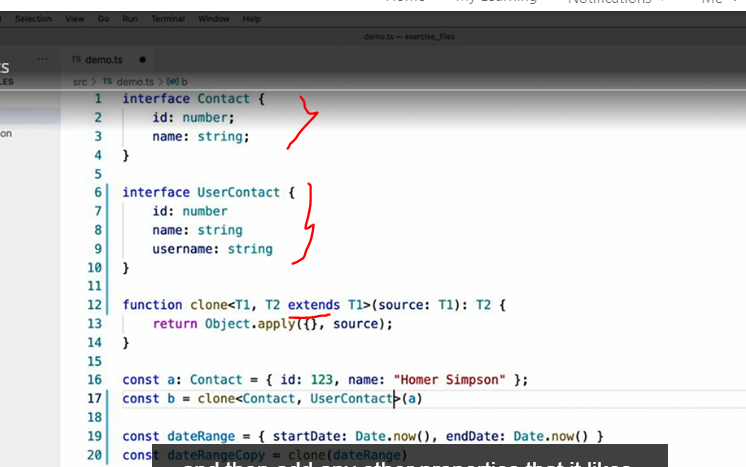
* Here, I've declared a variable without explicitly defining a type and then I've passed that variable through my generic clone method.
* In this example, TypeScript dynamically replaced the generic placeholder type T with the structure of this variable that I passed in and that type flows all the way through to the variable that I assign the return value to.
* Now, you don't have to stop at one generic type parameter either.
* You can define as many as you wish.



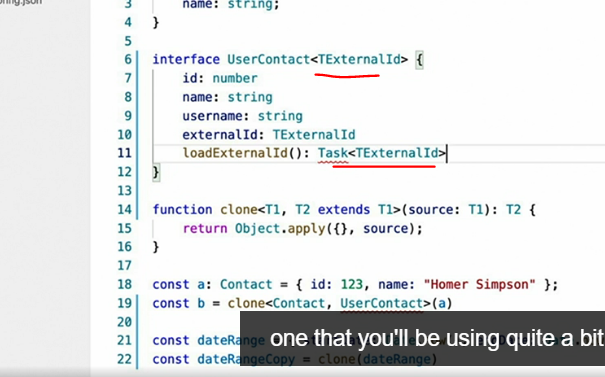
* For example, here, I've defined two generic type parameters that can each be used to refer to different types.
* When I use generics in this way however, a way in which TypeScript cannot easily infer what they should be, I have to set them explicitly when I call the method using this syntax.
* Here, I've used the contact type for both parameters but I could have just as easily done something like this.



* One final feature of this syntax is something called **generic constraints**, which allow you to place more restrictive rules around the types that may be used as generic type parameters in your functions.
* For example, if for some reason I wanted to allow this clone method to accept multiple generic type parameters but I wanted to make sure that the return type matched the input type, I could restrict the types that may be used for the second T2 parameter, using the extends keyword like this.
* This would allow me to supply any type I wanted as the second generic type parameter but it would have to at least match the properties of the first type parameter.
* For example, if I ***add another type that matches the signature of my contact type***, I can use that as the second type parameter.
* **Note that the extends keyword doesn't mean that the second type literally has to extend or derive from a given type.**
* It only **has to match it and then add any other properties** that it likes.



* In this example, I've created an entirely new interface, user contact, that has ***no direct relationship with the contact type.***
* It ***just happens to share the same properties that the contact type defines and that allows it to satisfy the extends constraint.***
* One last thing about generics, **they're not limited to functions.**
* They can ***be applied to interfaces in classes too***.
* Take this example.
* Here, I've used the generic syntax to define a generic type named T external ID on the user contact interface.
* Once I've defined the generic type, I can then refer to it whenever I want inside of the interface, whether it be the type of a property or even the generic parameter to another interface.
* As you might be able to tell, generics are an incredibly powerful tool in your TypeScript tool belt, one that you'll be using quite a bit.



* And guess what? This video only scratches the surface of what you can use generics for.
* Keep your eyes peeled because you'll be seeing a lot more of them and a lot more advanced usage of them throughout this course.