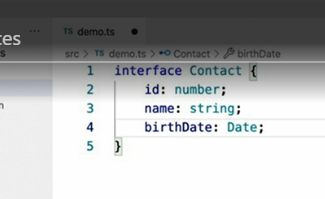
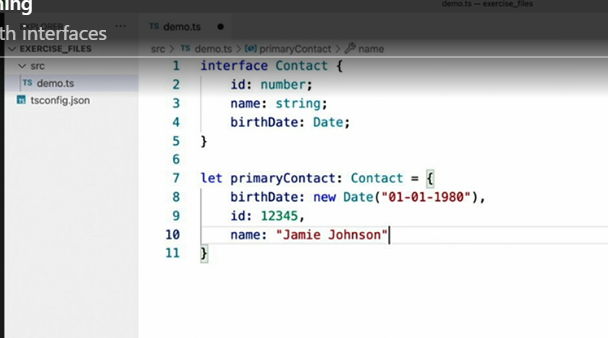
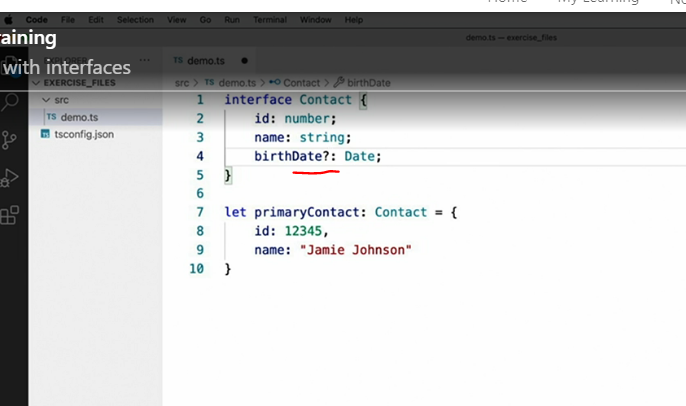
* - [Instructor] Now that I've shown you the syntax to tell TypeScript about primitive and built-in types it's time to **define our own custom type using what TypeScript calls an interface.**
* The syntax to define an interface is pretty straightforward, just the keyword interface, followed by the name of the interface, I'll call this one Contact, followed by open and closed braces.
* Then inside these braces we can begin adding the properties that we want this interface to have.
* For example, let's say a contact should have an ID field with a type of number, a name with a type of string, and a birth-date field with a type of date.



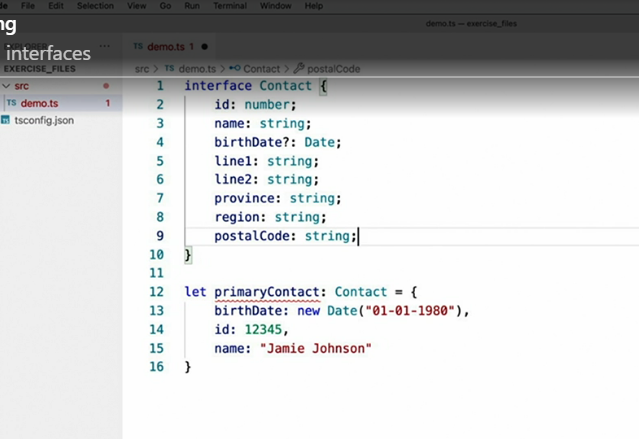
* Now, if you're familiar with the more recent versions of JavaScript and the class syntax this might look incredibly familiar.
* In fact, **any class definitions created using the JavaScript syntax can also be used as interfaces.**
* The biggest difference between an interface and a class however, is that **interfaces strictly exist as a way for you to provide type information to TypeScript.**
* They are *only ever used at compile time* and are never available, and will never even appear in your runtime code.
* Now, once you've defined an interface it can be used just like any other built-in type like this.
* Here I've created a variable named primaryContact with the type of our new interface Contact, but things don't get very exciting until we try to assign a value to this variable.
* For example, I'll start by just creating an empty JavaScript object and see what happens.
* Because I've defined this variable as a contact type and TypeScript knows the fields that a contact type has to have TypeScript is able to provide me with much better autosuggest capability, showing me the fields that I can add to this type.
* What's more, TypeScript is complaining that the value I'm trying to assign is not a valid variable because it's currently missing all of those fields.
* So, let me go ahead and add them.



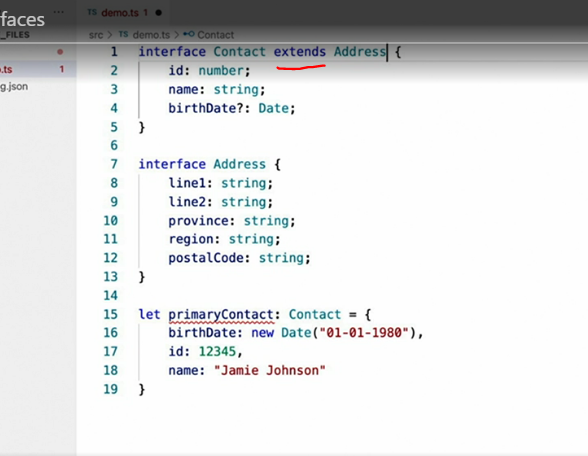
* Now that I've provided fields with values that match the types described by the interface all my errors go away, and it should come as no surprise that if I set one of these fields to a value of the wrong type TypeScript warns me of my mistake.
* ***What if you define an interface with many fields, but don't always expect every field to be populated?*** For example, in the case of the Contact interface what if we don't always need to have the contact's date of birth? Luckily, TypeScript allows us to define any one of an interface's fields as **optional by appending a question mark** to the field's name in the interface definition like this.



* Notice that once I make the birth-date field optional my compilation error immediately goes away.
* What this does is effectively tell TypeScript that in this case not every contact will always have a birth-date field, so don't require this field in order to consider something a valid contact.
* However, the interface does still enforce typing on this field when it does exist.
* *In other words, not every contact must have a birth-date field, but when a contact does have this field it must be a date.*
* So, even though I don't currently have any errors in this code, if I try to readd the birth-date field but assign it to a string value TypeScript does not allow it.
* However, if I go back to using the proper date type TypeScript is happy once again.
* As if that weren't powerful enough, **multiple interfaces can be combined to create brand new ones.**
* For example, let's say that I want my Contact interface to also include the contact's address.
* I could do that by adding those fields to the contact like this.



* Now, this would work fine until I wanted to use those address fields somewhere else.
* So, instead I will move them to their own interface called Address.
* And then I can merge them back into the contact type using the **extends** keyword.



* Doing this merges the Address interface into the Contact interface to create one big interface.
* With this code in place I can now assign those address fields to my contact objects.
* With this code in place I can now assign those address fields to my contact objects while still being able to reuse the address type anywhere else in the application that I need to.
* **Interfaces are a fundamental building block of creating a fully-typed application with TypeScript,** and I hope these examples show how simple and how powerful interfaces are as you'll be seeing quite a lot of them throughout this course.