* - [Instructor] The type script techniques that I've shown in the course up to this point have been pretty foundational, things that you'd use in just about any type script file you'll edit.
* From this video forward, however, I'm going to be getting into more **advanced** techniques, techniques that you may not use in your everyday type-script editing, but ones that will allow to be far more exact and expressive with your typings to increase the number of issues type script can detect and help you avoid.
* In this video, I'm going to be showing you **how to reference the fields of a given type using the keyof operator.**
* Using the keyof operator looks a lot like using the native **instanceof** operator.
* This line defines a type alias consisting of all of the properties on the contact type.

Graphical user interface, text, application

Description automatically generated

* In other words, **a variable of this type may only contain values matching the names of the properties on the contact, ID, name, birth date and status.**

Graphical user interface

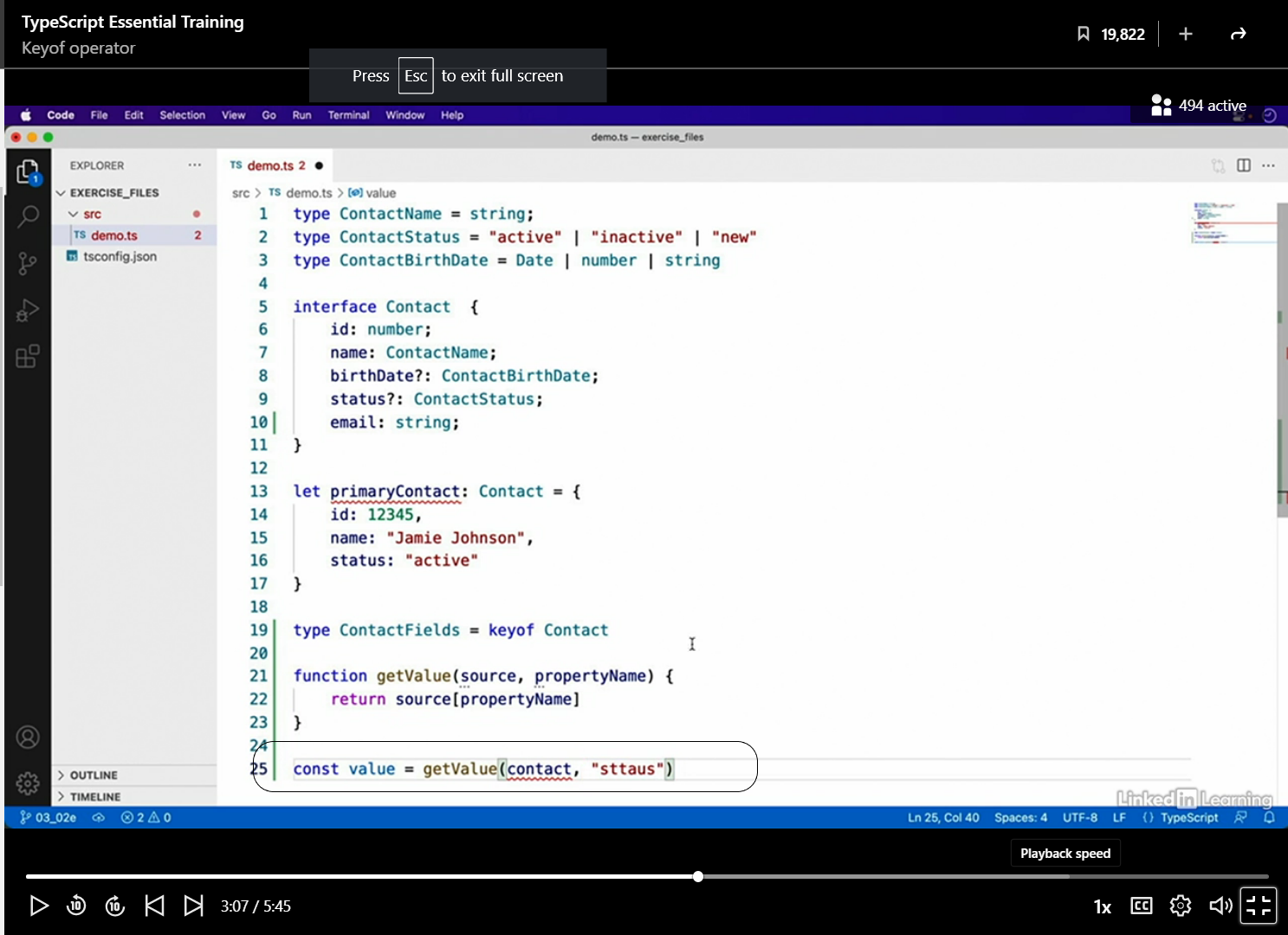
Description automatically generated with medium confidence

* To show it in action, I can try to create a new variable with this type, and see that the auto-complete options only include the names of the contact properties I just mentioned.
* This, of course, includes any fields that exist right now as well as any I might choose to add in the future.
* So if I add another field to contact, I can see this new field in the list of possible values as well.
* Note that ***just like the a string union type-aliases I showed in the previous video, the contact fields and its values are only available at compile time and not actually compiled into your running code.***
* So they're not available at run time.
* Now, this is pretty impressive, but it might not be obvious at first glance just how useful these kinds of time types can be.
* So let me show you a more real world example, one that I use all the time.
* This function accepts two parameters, an ***object*** and the **name of a property** on that object.
* The function then returns the ***value of that object's property dynamically*** using JavaScript's object index syntax.

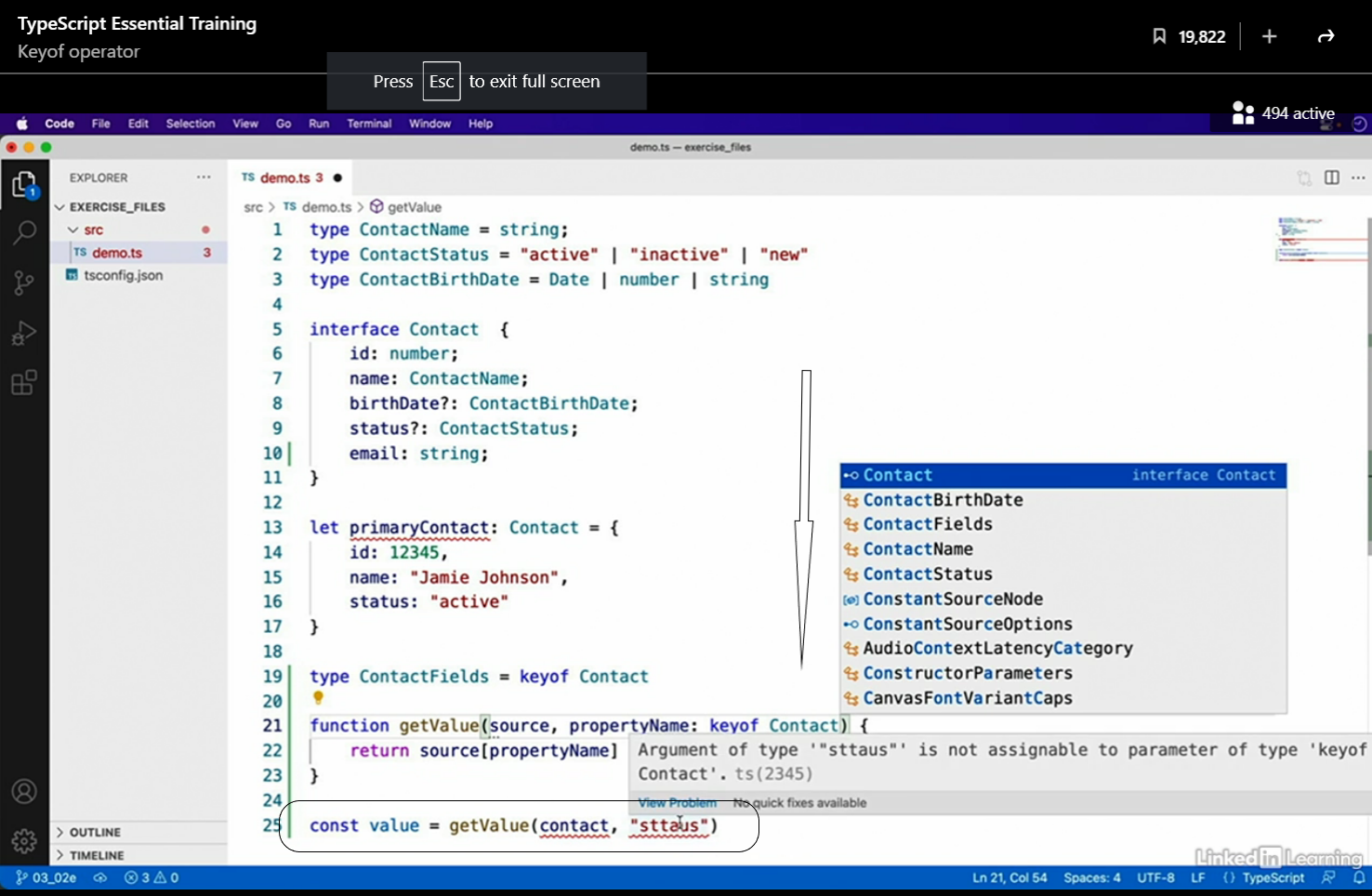
Graphical user interface, text

Description automatically generated

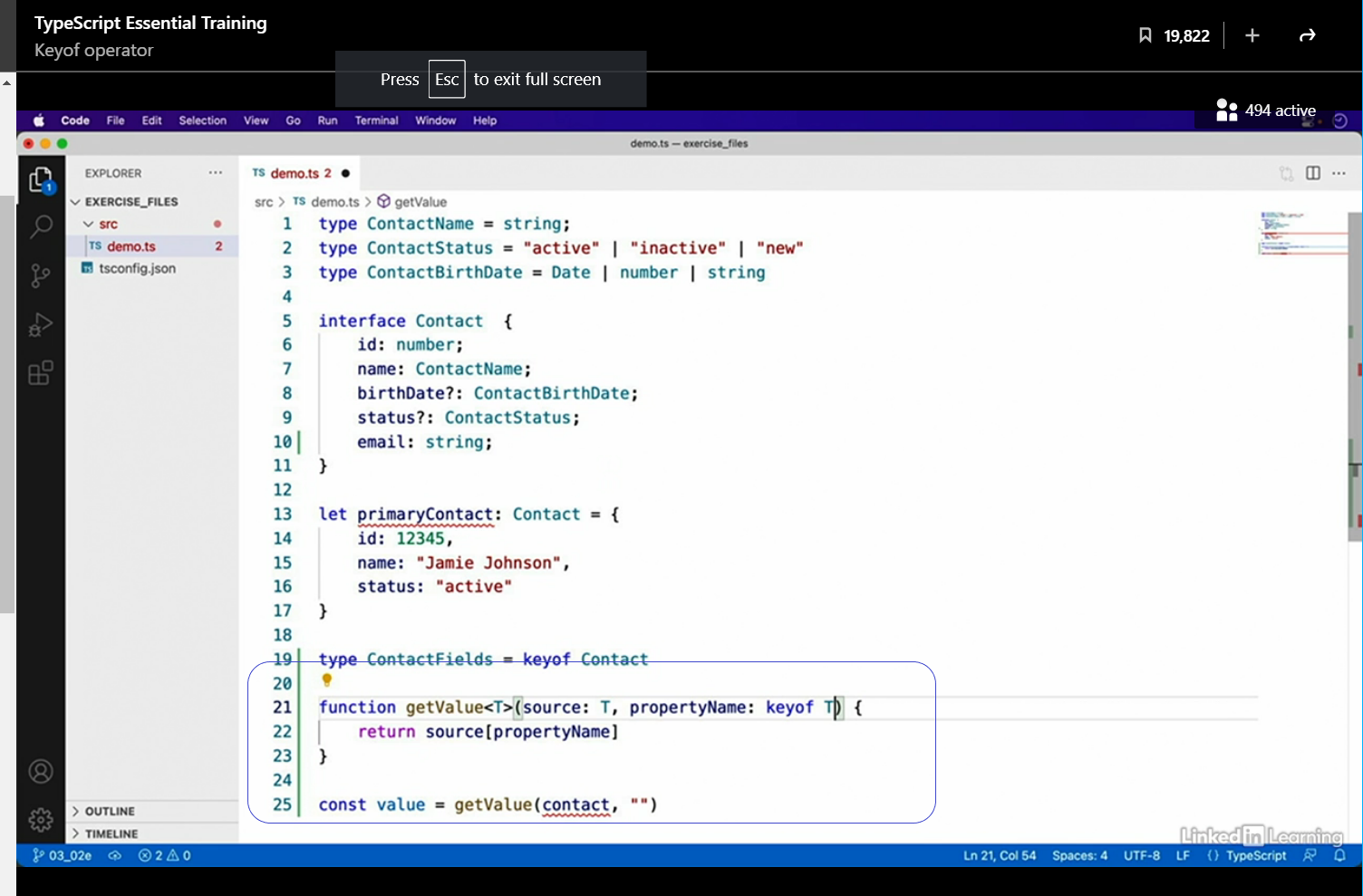
* This code may be simple, but it's a great example of JavaScript's powerful dynamic typing.
* Without strong guardrails, however, this dynamic approach can quickly get out of hand and result in easy to create and hard to diagnose bugs.
* For example, misspelling the name of a field.



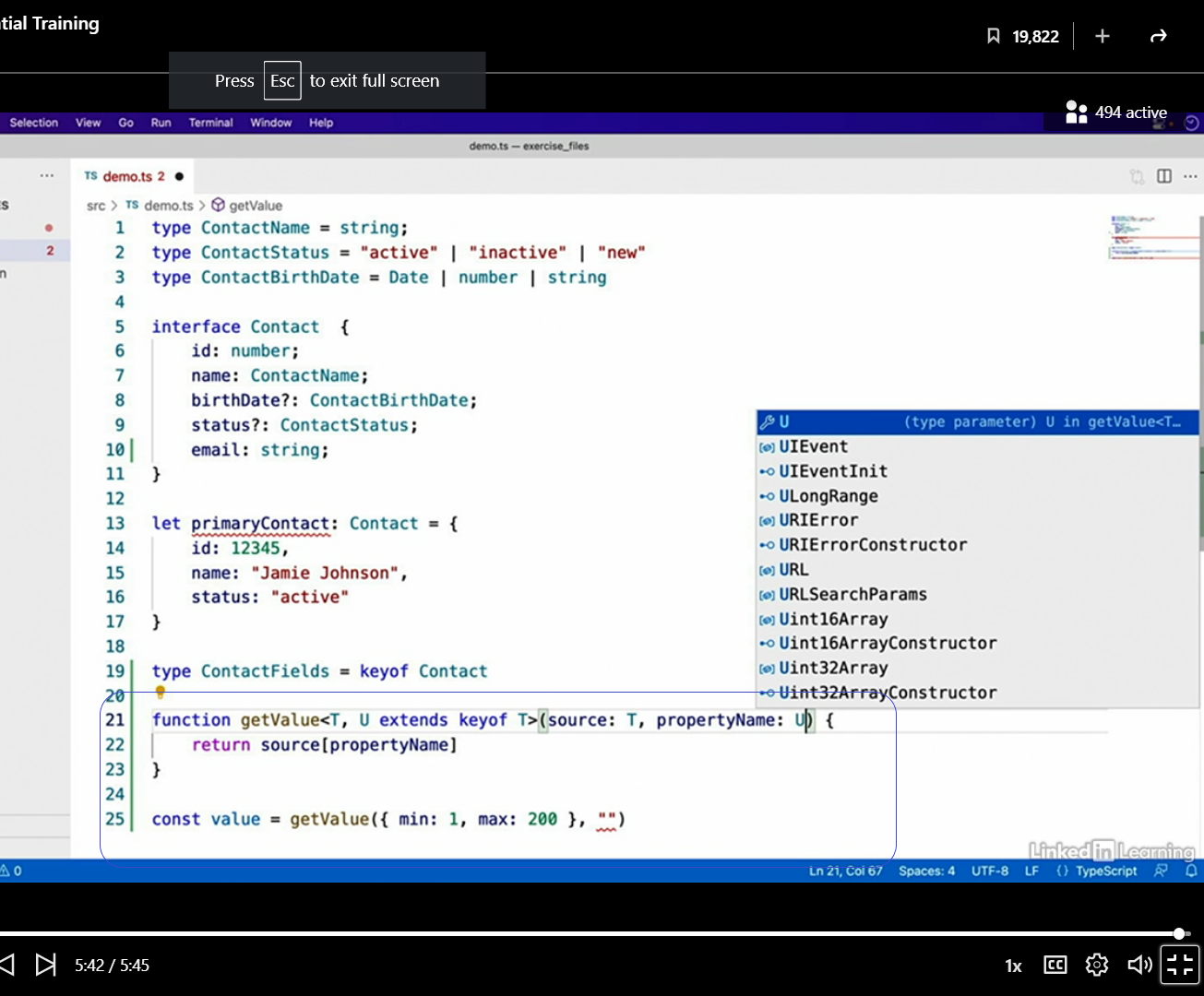
* Without any static typing to help out, JavaScript will happily execute this code and return the undefined value.
* However, if I **introduce a keyof** type to this second parameter, it limits the values of this second parameter to valid property names of the contact type, and type script, now, correctly highlights my mistake as an invalid value.



* What's more, as with my example function in the generic types video, the logic in this method has nothing to do with the values of the contact type specifically.
* So **refactoring it to a generic function** makes a whole lot of sense.
* Just take a second to appreciate just how powerful this is.



* Not only am I able to pass in any type as the target object to read from in the first parameter, type script readjusts to limit the second parameter, ensuring that it is a property name of the first parameter's type, no matter what that first parameter's type may be.
* For example, let me now try calling this function again, this time with an object for which I don't even define a type From this list of properties, I can see that type script was able to inspect the first parameter that I provided and extract its field names, all on the fly, but we're not done yet.
* Let's take this one step further.
* Back in the generics video, I showed you not only how to declare a generic type like T in this example, but also to constrain the types that may be used to satisfy that generic type.
* Well, the keyof operator is a great way to constrain your generic types.
* For instance, we can rewrite this function like this.
* Now, I haven't really changed the way that this works.
* The second parameter is still limited to the property names that appear on the generic T type, the first generic type.
* However, what I have done is introduced a generic type that can be referenced later on in this function.



* For now, I'll just present this as a cleaner version of this function, but in a few videos, I'll add a few more techniques to this function to make it even more powerful.
* So keep moving on to the next videos to see what I'm talking about.