Error tracing with stacks

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] What else could you use a stack for? Well they are great for programs where you need to reverse things. For example, I am very good at losing things, and if I could just go back in time and rewind time and find that lost item, that would be great. Well, in programming we can. Inside of a program we can track each step and ultimately retrace our steps as needed. So when would you need to retrace your steps in programming? Well it would be nice if you could do that when you get an error, and you can. This is where something called the runtime stack, or call stack, comes in. The runtime stack keeps track of what variables you currently have access to and what subroutine or function you are in. Whenever you get an error, an error message coming from the runtime stack usually appears, and you can retrace your steps from where the error occurred. So let's add an error to a program that uses stacks in Java. The program doesn't have to use stacks, but since we're talking about them, let's try it out in Java. Here we have a Java program and we import the stack class in Java. We import it so that we can use its functionality which we see here in the body of the main function. On line four we create a stack, then we push the item high and then we pop the item off. If we execute this we get no errors and so we don't even see the runtime stack. We get nothing in our console, all is great. But let's say we add an error. If we pop an item off of the stack. We say myStack.pop. So popping an item that does not exist on this stack. The stack is empty and has no items on it. We're popping off an item that does not exist. If we execute this we're going to get an error right here. We get this exception which is essentially an error. It's an empty stack error or an empty stack exception. And then it traces where exactly it errored. We see in this one file on line 102 something here errored and then a pop errored, so basically what happens is the program tried to peek and see what that next item was, then it popped it off. All of that errored. Then we got back to our main class on line seven of our main class is where we have the error. So it's when we popped off this item. And it errored because when we peeked we got an item that did not exist. The item that we were trying to see, trying to reveal before we popped it off the stack did not exist. This whole process that we see here, this tracing back in time of where the error occurred, what it affected, and then what initially caused the error, which is our line on line seven. This is all a part of the runtime stack. And it's the runtime stack that keeps track of the state in time and checks out, like okay, this is where the error first occurred. Then it affected this. Then it affected this. That is all kept track of with the runtime stack. Now this error is actually pretty common with stacks. If the stack is empty, the only mutable operation that is valid is push. Popping an empty stack as we just saw here will cause an error because there's nothing to pop, there's nothing there. Now stacks work differently depending on which language you use them in. And Java, as you saw here, we can use the regular stack class. This stack class comes with the peek, push, pop, search, and empty methods. In C# and .NET we also have a stack class which has peek, pop, and push. In Python we do not have an explicit stack class. But we can use lists as stacks. Instead of pushing and popping we can append and pop. Ruby also doesn't have a specific stack class, but we can use an array to do this because the array is resizable and has push and pop methods. Similarly in JavaScript we can use its array that has push and pop methods since there is no explicit stack data structure.