2. Handling Exceptions

Contents

[1. Introduction 1](#_Toc17986)

[2. Working with try/catch 1](#_Toc9440)

[3. Handling Cleanup 1](#_Toc7475)

[4. Implementing try/catch/finally 1](#_Toc4836)

[5. Automating Cleanup 1](#_Toc1093)

[6. Transitioning from Manual to Automatic Cleanup 1](#_Toc11633)

[7. Summary 2](#_Toc7150)

# 1. Introduction

=>slides: Pg. 1

Welcome to Handling Exceptions in Java.

In this first module, take a look at how we get started with exception handling. Throughout this module, we'll take a look at this idea of exception handling and how we deal with exceptions within our code.

=>slides: Pg. 2

So we'll start out and look at the role of exceptions. Just why do exceptions exist in the first place? Next we'll take a look at how we deal with exceptions in our code, and we'll do that using something known as try/catch blocks. Now a key part of doing any work often involves some kind of cleanup, so we'll see how we can actually incorporate cleanup into our try/catch blocks by adding a finally block. Then we'll finish up, we'll look at how we can automate cleanup, which often allows us to simplify the code within our applications. So let's just jump right in.

=>slides: Pg. 3

Now as an application developer, a reality that we face is that programs are going to encounter errors, and our applications need to be notified of those errors, and our applications have to deal with those errors. And over the years a number of mechanisms have been tried to make working with errors a little bit simpler. In the earliest days, methods often returned back an error code that we had to constantly check explicitly within our code. Over time, there were these ideas of global error codes and even thread‑level error codes. Now although these provided the information that we needed, dealing with the errors themselves was often very cumbersome. So what we really need is some kind of mechanism that allows us to effectively deal with errors, have the code that we want to run, and recover from those errors in a structured, meaningful way. And that's where exceptions come in.

=>slides: Pg. 4

Exceptions provide a non‑intrusive way to signal errors. The language we use when talking about exceptions is that when the exception occurs, the exception is thrown. And our application can deal with that exception by then catching the exception. And then this mechanism of the exception being thrown and then caught by our code allows us to have structured error handling in our applications. We can have a clearly distinct area of code that represents the work that we're trying to perform, and then another clear, distinct area of code that can handle any errors that occur during that work To deal with these exceptions, in Java, we use what are known as try/catch blocks, and these try/catch blocks divide our code into the two distinct areas, the work that we want to perform and then our error handling code. Now the try portion, or what we call the try block, contains our normal code. Other words, this is the code that we're actually trying to execute. This is the work that we want to perform. And when everything goes well, the code within the try block will run to completion. It's just a block of code, and it executes. But if something happens, if an exception gets thrown, the code within the try block is then immediately exited. So at the point that the exception gets thrown, we jump out of the try block, and control is transferred to what's called the catch block. The catch block contains our error handling code. Now under normal execution, the catch block is skipped, but when an exception occurs, that's when the code within the catch block is then run. And the catch block not only has a chance to run, the catch block actually receives information about the exception. So with these try/catch blocks, we have a distinct area of code that contains the work that we're trying to perform. That's our try block. Then we have a separate area of code that's distinctly responsible for dealing with the error, and that's our catch block. So next , let's take a look at some code that uses a try/catch block.

# Working with try/catch

=>slides: Pg. 5

Let's take a look now at how try/catch blocks behave within an application. Let's just do something really simple. So we'll start out by declaring 2 variables, i and j, and we'll initialize each of those variables with a value. And then we'll perform a simple math operation. We'll subtract 2 from the value of j, divide that into i, and store it into a variable named result. And then finally, we'll just print out the value of result. Now looking at code like this, it seems so simple and might appear that nothing could possibly go wrong here. But it turns out that's not actually the case, because notice that our math equation includes a divide operation. And anytime your application involves division, there's an opportunity for an exception to occur. Because remember that mathematically, you can never divide by 0. So if there's any chance the value dividing by could be 0, there's a possibility that an exception might be thrown. So because of that, we want to be sure to protect this code, and we'll do that by using a try block. So we'll wrap the code we want to run using the keyword try, and the code itself is enclosed in opening and closing curly braces. And then normally what we'll do is take the code that's inside the block and indent it. There's no special meaning for that; it simply makes our code a bit more readable. So now we have the work that we actually want to do contained within the try block. So now the next thing we need to do is add the code the handle any exception that might occur, so we'll then use the catch block.

=>slides: Pg. 6

So we do that by using the catch keyword. Then notice after the word catch in parentheses, we have a variable here ex whose type is Exception. So if an exception gets thrown, that variable ex will receive the information about that exception. Now the code for the catch block again goes inside of opening and closing curly braces. And inside the catch block is the work we want to do if an exception occurs. One of the most common things we'll want to do is just go ahead and print out information about the exception. Because remember we said that the exception actually tells us a bit about what happened. One of things we most commonly want to do is get the message that's associated with the exception. So if we receive an exception, we'll print out the string Error, followed by the message that's contained in the exception. Now exceptions can also contain other information.

For example, exceptions also have information about the stack trace that led to the exception itself. In other words, the exception can tell you the list of method calls that occurred that ultimately led to that exception. And if you compiled your application in debug mode, the stack trace will not just have method names; it will also have the line numbers where each of the calls occurred. Now, in general, you wouldn't want to print the stack trace out to the end user. But during application development, bringing out that stack trace can be hugely helpful because it makes it much easier to debug our applications. So now we have our try block, which contains the work that we're trying to do. We have our catch block, which will deal with the exceptions associated with that work. But this block is just a small part of our overall application. Because our application probably wants to do more things beyond just this little bit of code. And that's just fine, because after the try/catch block, we can do any other work that we want to do. The try/catch block is just part of the overall flow of the application. So now if we run this code, first thing we would do is initialize i to 12. Then we initialize j to 5, and then from there, we now move into the protected work, the equation. So we started out subtracting 2 from j, so 5 ‑ 2 would give us 3, and divide 3 into i, so 3 into 12. Store that outcome in and our variable named result. Then we would print out the value of result, which would print out the value 4. So in this case, our try block ran to completion. So the flow of our application would simply jump to the next line of code after the catch block. Then our application would just run normally from that point on.

=>slides: Pg. 7

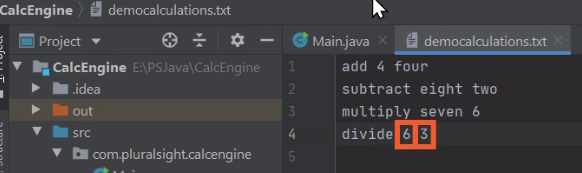
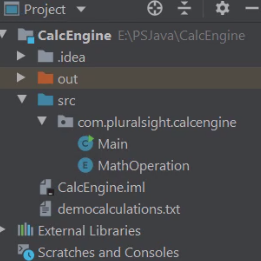
But now let's make a small change to our application. Here where we initialize j to 5, let's go ahead and initialize it to a value of 2. So now as we run our application, we set i to 12, j to 2, move into our try block, and we subtract 2 from j. Well 2 ‑ 2 is going to be a result of 0. So we'll then attempt to divide 0 into i, and dividing by 0 is illegal mathematically, so that's going to throw an exception. So as soon as that exception is thrown, we immediately leave the try block. That means that any other code in the try block doesn't run. A value is never assigned a result, and the code to print out the result never runs. So instead, the flow of control is here in the catch block. We receive the information about the exception in our variable ex. We then run the code in the catch block, which will print out the text we have here, so it'd print out error and whatever message is associated with the exception, which is divide by 0. We then run the next line of code in the catch block, which will print out the stack trace. And then once we finish the catch block, and this is a really key part of exceptions, the program flow continues normally. So we would then do any other work that occurred after the catch block, and that code would keep running throughout the rest of our application. And that's the key value of try/catch. It allows us to protect an area of work by placing it inside of a try block, associating the error handling with that work by placing it in the catch block. And that work and its error handling are then coupled together as part of the overall flow of the application. So whether the work succeeds or the exception occurs, all the work after the try/catch block continues on normally, and it's allowed us to protect our application code in a meaningful and structured way. So let's take a look at how we handle cleanup when working with try/catch blocks.

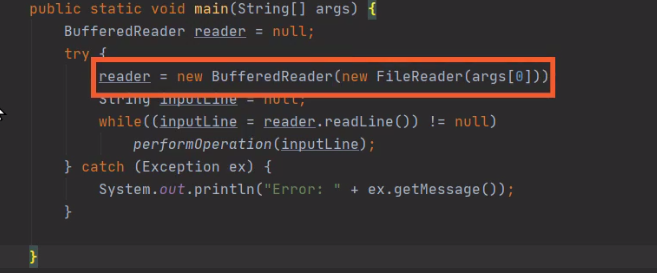
# Handling Cleanup

=>slides: Pg. 8

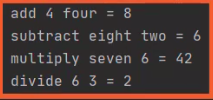
As we're doing the work within our applications, at any given time the task that we're trying to perform may require some cleanup, and this often comes up when dealing with external resources. So if you're reading from a file, you need to close that file. If you're working with a database, you want to close the database. If you're communicating over a network, you want to shut down that network connection. So you want to be sure that this cleanup work actually occurs if it's necessary for the task you're performing. And it's important to understand, when we're working with try/catch blocks, you may need to do your cleanup not only when the work succeeds, but you also might need to do that cleanup in the event that an exception occurs. So we need an easy way to be sure that cleanup occurs whether the try block runs to completion or when an exception is thrown and we transfer control over to a catch block. And that's where the finally block comes in. The finally block can be added to a try/catch, and the code we place in the finally block runs in all cases. So if the try block runs to completion, after the try block is done, control is transferred to the finally block, which can do its cleanup work. But also in the case, if during the running of the try block an exception occurs and control is transferred to the catch block, when the catch block finishes, then again the finally block will have a chance to run. So using this mechanism, we have three distinct areas of work. We have the try, which is the work we're trying to perform, the catch, which is our error handling, and then the finally, which is our cleanup. So to get a better understanding of all this, in our next section we'll jump into our Java IDE, and let's see how we can use, try, catch, and finally to protect some application code and assure that our cleanup properly occurs.

# Implementing try/catch/finally





Here we are inside of our STS IDE. It's a fun little application that we've built that allows us to specify a math operation and two numeric values, and those numeric values can either be literal numbers, or they can be words that represent those numbers.



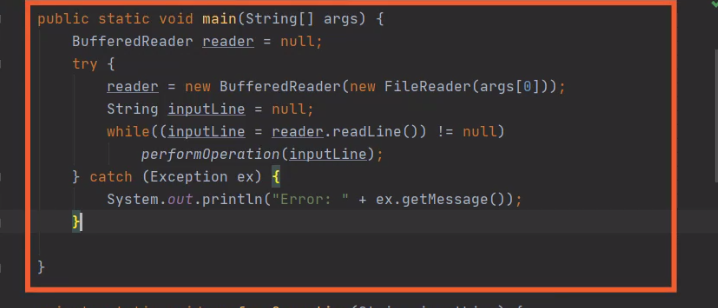
When we run the code, it does a calculation. You can see here in the Run window that it will actually display the results of that calculation.

That sets everything up and shows you how to receive command lines and do the basic work.

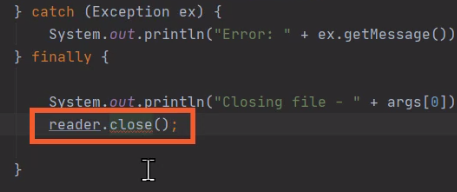
But fundamentally, what we do is receive the operation, the two values, and then do the operation displaying the result. So let's see how we're using that capability in this application. So I'm going to go ahead and close he Run window, and I'll switch over to our Main class. So here we are in our Main class, and what we're doing in this class is actually reading the list of operations from a file. And the file name is actually passed in on the application command line. So we basically use a file reader to open up the file, wrap that in a BufferedReader, and just read the equations one by one. So we read those equations in, we translate them with a method called performOperation, and then finally we execute them in a method called execute.



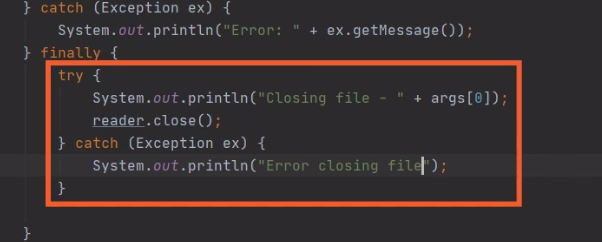
Let let me scroll down to this method named execute. So now we're down here in this method execute, and in this method, this is where we actually do the operation. And you can see what fundamentally has happened. We've taken all those string values, done all the translation work, and then ultimately we just do an equation. We either add two values together, subtract one value from another, multiply, and then even just do a simple divide. So ultimately, we're just doing a simple math operation, just using words to describe that operation. But since we're reading these operations from a file, there's one special thing we have to consider. So let's scroll back up to our main method.



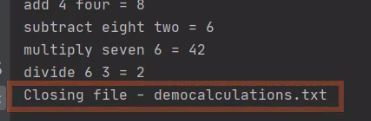
So here we are back in the main method, and remember that we're using a BufferedReader to read these operations from a file. So that means that when we're all done with that BufferedReader, we need to be sure to close it. And this is a perfect opportunity to use the finally here within our try catch. Because remember we want to make sure the file's closed, even if an exception occurs, and that's what the finally block will do for us.



So here at the end of our try catch we'll add the keyword finally. Then we'll add our opening and closing brackets. Now the main thing we want to do in this finally block is close the reader. So let's print out a message saying that we're closing the reader so we'll know when this code runs, and then we'll call close on our reader reference. So now here in our finally block we'll print out that message saying we're closing a file, identifying what the file name is, and then we call reader.close. But notice that close is underlined in red. Let me go up here and hover my mouse over the close method. Now you notice here the reason we have this red line under close is it's indicating that an exception might occur. So that means that we actually need to wrap this call to close in its own try/catch, because even though this finally is associated with a try/catch block, the catch that's part of this try will not handle any exceptions that happened in the finally block. So that means what we need to do is actually wrap this call to reader.close in its own try/catch block. So let me go ahead and add that code.



So now here inside the finally block, we have a separate try/catch to handle any exception that might occur as we try to close the reader. Let's go ahead and run our application, and let's see what happens here.



So now you know as we run the application, we go through and perform all the operations that are in the input file. Then when that work finishes, we close the file. So that shows us when we ran our try block to completion, the code and our finally block ran to close the file.



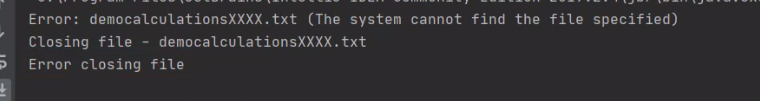
Let me go up here to our democalculations file where we have those operations, so we'll switch over to that file. And here in this file, let's change this last operation from divide 6 by 3 to divide 6 by 0. So now that we're dividing by 0, we know that's going to throw an exception. So let's run our code again and let's see what happens.



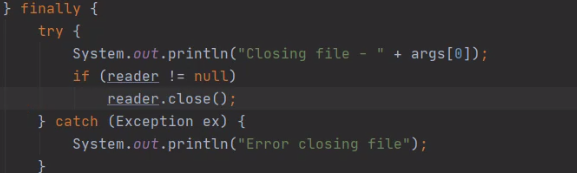
So now you can see in our Run window that when we get to the operation where we divide by 0, we throw the exception indicating that we divided by 0. But then notice after that we still closed the file. So our finally block is doing exactly what we want it to do. But there's still one more situation we have to consider. Let me switch back over to our main class.



So now we're back over in our main class looking at our main method. Remember, we're using his BufferedReader to read the file, and then we close that reader. Well, what would happen if we tried to open a file that didn't exist? Now remember, we're receiving that file name from the command line, and we can configure our command line arguments here inside of IntelliJ. Remember, we talked about how to do this back in the course, Getting Started with Programming in Java. To do that, we'll head up here to our Run menu. Then we'll head down here to Edit Configurations. We'll choose that. And then one of the options we have here are program arguments. So the values we place here are passed into our program, as if the user provided them on the command line. Now currently, the program argument is a valid file name, so let's change that to be an invalid file name. So now I've added a bunch of Xs to the file name, which is not a valid file name. So let me go ahead and accept that. So now when I run the application its going to try to open a file that doesn't exist. Let's see what happens.



So notice now when I run the application, we first get an exception indicating that we tried to open a file that doesn't exist. But then, even though we didn't open it, we then try to close it. But that in turn throws an exception because our reader reference is actually null. So we need to address that so we don't try to close a reader that was never opened. Let me go ahead and collapse our Run window down here at the bottom. So now we're back here in our main method. Then we have our finally block here inside the main method, and what we want to do is, before we actually call close on the reader, we want to make sure the reader is non‑null.



So let me add an if here that checks to see if reader is non‑null. And now with that change, our finally block can now operate properly. We have the close protected by a try/catch block in case an exception happens during the close. We also make sure our reader is non‑null to verify whether we should call close or not. Now as you can see, this is a fair bit of work, but this is what we need to do to make sure that we properly close that reader when we're all done with it. But it turns out, in cases like this, Java provides another option that makes cleanup much easier, and we'll see how to do that next.

# Automating Cleanup

=>slides: Pg. 9

Incorporating the appropriate cleanup code as part of our try/catch blocks is of course very important. Well, as we saw on our Java code, manually performing cleanup can actually be quite cumbersome in some cases. Because we often have to have to include null checks to make sure the resource we're trying to clean up was actually created. And also, we may need to exploit a try/catch within the finally block itself because the cleanup work can also throw an exception. And these sort of things are very common. This need for null checks and exception handling as part of our cleanup comes up over and over again. So it would be nice if we could automate the cleanup process a bit, and it turns out that we can.

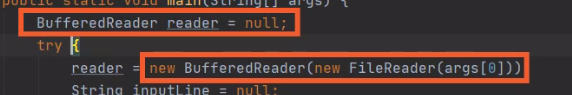
=>slides: Pg. 10

Now there's two parts to automating clean up. First of all, a type has to indicate that it supports automated clean up, and then when working with those types, in our code, we have to code our applications in a way that will leverage that automated cleanup. So first, let's take a look at the types. Any type that supports automated cleanup needs to implement the interface AutoCloeseable. And the role of this interface is really simple. It's simply to indicate the type supports automated cleanup. Now the AutoCloeseable interface only has one method, which is the close method. So any type that implements this interface is responsible, to put all the necessary cleanup code inside of its close method. So if we call close on that type, it assures that the resource is properly cleaned up. Now it turns out that most types do not directly implement AutoCloeseable. They generally implement other interfaces that inherit from AutoCloeseable. The most common of those is the Closeable interface. Now as I mentioned, it inherits from AutoCloeseable, and it actually doesn't even add any other methods. It simply has that close method. So any time you're working with a type that inherits from the Closeable interface or any other interface that inherits from AutoCloeseable, that means that type supports automated cleanup. =>slides: Pg. 10

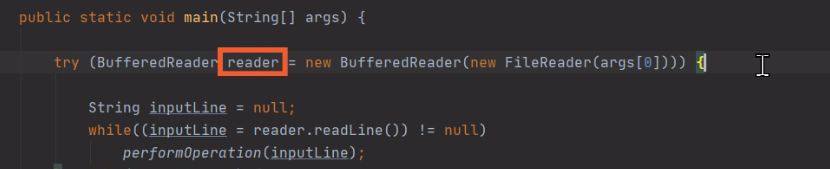
Now when we're working with one of these types, we need to code our applications in a way that will leverage that automated cleanup, and we do that by using the try‑with‑resources statement. And try‑with‑resources allows us to create an instance of a type, and when we're finished with that type, Java will generate the appropriate code to automatically call that type's close method. Now to use try‑with‑resources, we have to be using a type that implements AutoCloeseable or any other interface that inherits from AutoCloeseable, and Java will generate the code necessary to automatically call that type's close method. And in addition to calling the close method, it will also automatically incorporate the necessary null checks. It will only call the close method if an instance of that type was successfully created. Now as we work with try‑with‑resources, the syntax is going to look really familiar. It's going to look a whole lot like a regular try block, but with one key difference. The type that we want to clean up has to have a lifetime that's scoped within that try block itself. So when we're using try‑with‑resources, the type we want to automatically have cleaned up we'll need to create as part of the try block itself. So we'll create a new instance of that type as part of the try block, and when we reach the end of the try block, the close method will automatically be called on that instance. Now try‑with‑resources, just like a regular try block, can handle exceptions. So we can optionally include catch blocks. And the really cool thing about try‑with‑resources is the catch blocks that you put on the try‑with‑resources will handle not only the code that you explicitly put inside the try block, but they'll also handle any exceptions that are thrown as part of the automatic call to the close method. So rather than having to duplicate try/catch blocks within a finally, the catch blocks you put on try‑with‑resources will automatically take care of handling the exceptions that might occur as part of the automated call to the close method. So to get a better understanding of all this, in our next section we'll get back into our IDE, and let's modify our code to use try‑with‑resources.

# Transitioning from Manual to Automatic Cleanup

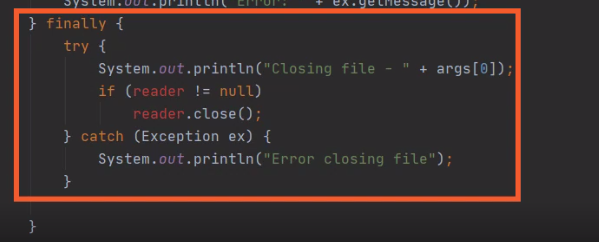
Here we are back in our application code. What I want to do now is change the application so that rather than using explicit finally block to close our reader, we're instead going to use a try with resources. So to convert this to a try with resources, one the key things that has to happen with our reader is there needs to be scope within a try block.

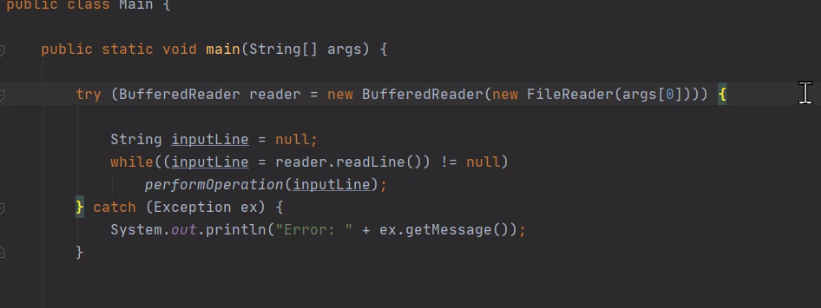


So that means that our declaration of our BufferedReader and also the code where we actually create the instance of the BufferedReader all has to be tied to the try block, and we do that by going up here to our keyword try and test if the try will add opening and closing parentheses, and then inside these parentheses, we'll first declare our reader variable.



Then once we have our variable, we'll take all the code where we create the BufferedReader over the FileReader and do that creation right here within the parentheses. So now our BufferedReader is scoped against that try and that makes this a try with resources, and as a try with resources, Java will take care of all the cleanup work of the reader when we're done.



So that finally block we have down here where we do all that work manually that closed a reader, checking for null, and handle any exceptions, we don't need any of that code, so we can simply remove that finally block. And that's it. We're all done. 

Java will take care of all the details of closing up our reader. It'll verify that the reader is not null before it tries to call close on it, and in addition, this one catch block we have here will now handle any exceptions that happened here in our try block, as well as any exceptions that might occur when trying to close the reader. So, as you can see, using try with resources greatly simplifies our cleanup work because Java takes care of those details for us.

# Summary

=>slides: Pg. 12

To wrap up., here's some of the key things we want to remember from this module. Remember that in this module we got our first look at exceptions. Remember that exceptions are used to signal errors, and the cool thing about exceptions is they get rid of this need for checking for error codes or looking at error flags. Instead, they signal that an error occurs, and we have a structured mechanism for handling those errors. And the way we handle those errors is by relying on a language construct known as try/catch blocks. So first we have our try block.  
=>slides: Pg. 13  
 The try block contains the normal code. In other words, the try block contains the actual work that we're trying to perform. And as long as everything goes well, that try block will run to completion. Other words, nothing special will have to happen. But in the event of an exception, our code will immediately exit the try block. So once that exception occurs, no other code within the try block will run. Instead, control is passed to our catch block, and that catch block contains our error handling code, the code we want to run in case something goes wrong. Now remember, the catch block only runs if a matching exception is thrown. As long as everything goes well in the try block, the code in the catch block never runs. But if an exception does occur, the first thing that happens is the exception. Information is actually passed to the catch block. So the catch block will receive that exception information and then run the code contained within the catch block. As we saw, when working with our try/catch blocks, we also need to perform some kind of cleanup, and that's where the finally block comes in.  
=>slides: Pg. 14  
The finally block allows us to provide our own manual cleanup code. And the cool thing about the finally block is it runs in all cases. So if the try block runs to completion, once that's done, the code in the finally block runs. But even if an exception is thrown, control is transferred to a catch block. Once the catch block finishes, the finally block has a chance to run. But now as we saw, when implementing a finally block, there are common things we have to do that can be somewhat cumbersome. We have to incorporate things like null checks, and we often have to have to incorporate additional exception handling within the finally block itself. So to help us out with that Java supports automated cleanup. In order to use automated clean up, we have to use the try‑with‑resources statement. That indicates that we have a resource that we want to have close automatically called on once our try block finishes. But in order to use try‑with‑resources, a type needs to indicate that it supports try‑with‑resources. And it does that by implementing the AutoCloeseable interface or one of the interfaces it inherits from AutoCloeseable. As we talked about, the most common one of those interfaces is the Closeable interface. All right, that wraps up this module. In our next module, we'll see how we can actually add multiple catch blocks to a single try block to improve our error handling to provide different code for different types of errors.  
  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*