4. Creating Custom Exceptions

Contents

[1. Introduction 1](#_Toc6924)

[2. Create and Throw an Exception 1](#_Toc10787)

[3. Throwing an Exception 2](#_Toc14712)

[4. Custom Exceptions 3](#_Toc21790)

[5. Declaring a Custom Exception 4](#_Toc26350)

[6. Throwing a Custom Exception 4](#_Toc32295)

[7. Chained Exceptions 5](#_Toc8621)

[8. Chaining Exceptions Together 6](#_Toc19959)

[9. Accessing a Chained Exception 7](#_Toc19768)

[10. Summary 7](#_Toc10500)

# 1. Introduction

=>slides: Pg. 1

Welcome to our next module, Creating Custom Exceptions.

=>slides: Pg. 2

Now throughout this course, we've been focused on the issue of handling exceptions that are thrown by other parts of our code. Well, in this module, we're going to turn things around a little bit. We'll start out by seeing how we can actually throw an exception from our code. In other words, how do we actually indicate from within our code that something has happened and use an exception to do so? Now as part of that discussion, we'll also see how to create a new exception instance. Because it turns out, before you can throw an exception, you need to create an instance of that exception. Now Java has a really rich set of built‑in exception types, and those built‑in exception types handle many scenarios, but they don't necessarily handle every situation you might want to represent from within your code. So Java allows us to define our own custom exceptions. In other words, we can create our own exception types and then throw those exception types just as you can throw the built‑in exception types, and this gives us a lot of flexibility in how we represent exceptions from within our code. Now it also turns out that exceptions can be chained together, and what that means is you can have one exception that wraps another exception, and this allows us to have higher‑level exceptions that may represent our application code or the application work we're trying to do while preserving any underlying exceptions that might occur.

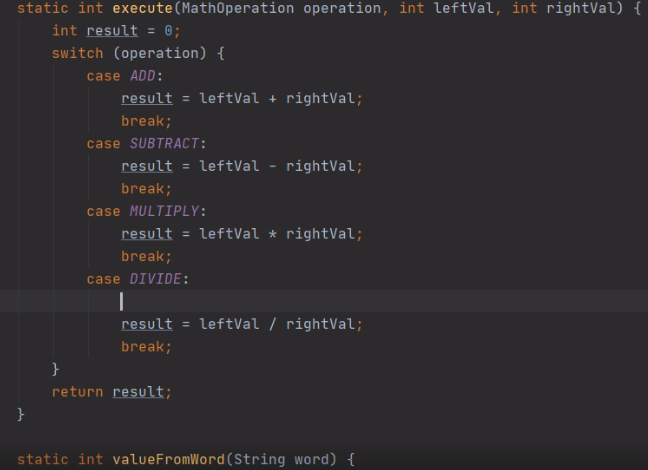
# Create and Throw an Exception

=>slides: Pg. 3

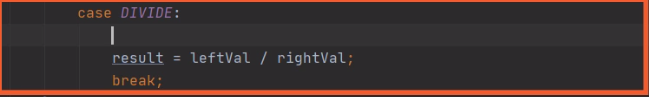
Now as we know, exceptions allow us to deal with errors. And up until now, we've been focused on one side of that issue, which is responding to errors that occur. Now as we know, the way we do that is by using try catch blocks. We actually catch any exceptions that might occur. But now what about the other side of the issue, though? What if our code needs to indicate that an error has occurred? How do we do that with an exception? Well, the way we do that actually makes a lot of sense. If the way we handle an exception is by catching it, the way we indicate that an exception has occurred is by throwing it. So there's a keyword, throw, that allows us to throw an exception to indicate that some issue has occurred. =>slides: Pg. 4

Now before we throw an exception, we first have to create the exception. The way we create the exception is by using the new operator. And, of course, the reason we use the new operator is because we know that exceptions are actually represented by classes. So whenever we want to throw an exception, we're going to create an instance of the class that represents that exception. Now as part of creating an exception, we need to provide some information about that exception. In general, the way we do that is by using the classes constructor, so we'll pass any necessary information to that constructor. At a minimum, we'll normally pass in a string value that describes the problem that occurred. And then, in some cases, there may be some additional information that we also want to include as part of that exception. But fundamentally and most commonly what it comes down to is we need to create an instance of the class that represents the exception, pass in at least a string description of the problem that occurred, and then once we create that exception, we can throw it. So to see what all that looks like, in our next section, let's jump into our CalcEngine project, and we'll throw an exception.

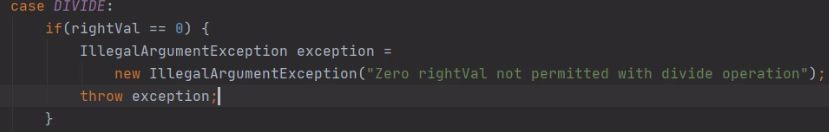
# Throwing an Exception



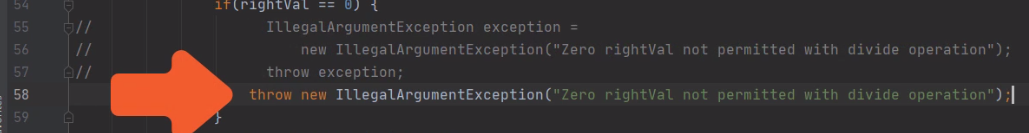
Here we are in our in our CalcEngine project and what we want to do now is see how it can throw an exception from within our code. Now, I'm currently looking at our Main classes execute method, and as you recall, the execute method takes care of the details of performing one of the four basic math operations. To do that, it accepts three arguments, an operation, a leftVal, and a rightVal.



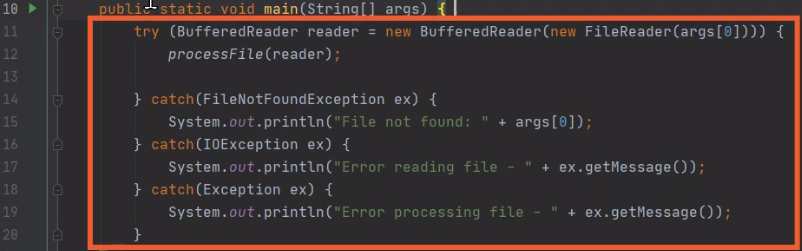
Now, if we look here where we do the divide operation, notice we simply take the leftVal and divide it by the rightVal. Now, of course, if rightVal is 0, that's going to throw a divide by 0 exception and that would certainly indicate that some kind of problem has occurred, but it might be better to provide some more information than a simple divide by 0 because, in effect, what's happened here is an illegal argument has been passed. A rightVal of 0 is an illegal argument on a divide operation. So what we can do here is check to see if rightVal is 0, if it is, we could throw an IllegalArgumentException and provide a description that actually indicates what happened. So let's do that.



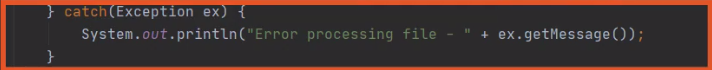
So here, in our divide case, let's start out by checking to see if rightVal is 0. Now if rightVal has a value of 0, what we want to do is create an instance of our IllegalArgumentException. So let's create a variable named exception of type IllegalArgumentException. Then we'll create an instance of IllegalArgumentException passing in an appropriate description. Now that takes care of creating the exception and we provided the description that says 0 rightVal not permitted with divide operation. So now to cause the exception, we need to throw it. So we're going to throw the variable we created, so we'll say throw exception and that easily, we've created and thrown an exception. Now, this technique we're using here where we create an instance of the exception, assign it to a variable, and then throw the variable will certainly work just fine, and in some cases you might need to do it that way if you have other things, you want to set on the exception after you create it, but in most cases we simply create the exception and throw it. So because of that, normally what we'll do is simply called throw and create the exception directly.

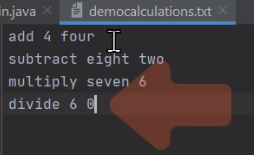


So what I'll do here is comment out these lines I've just added and then replace that with a throw new IllegalArgumentException. So now with that change, we've done all the work we need to do in just one line. We simply create the exception providing any necessary information to the constructor, and then we throw that exception directly. So now that we're throwing the exception, let's scroll up to our main method and make sure we have the appropriate catch blocks to catch this exception.

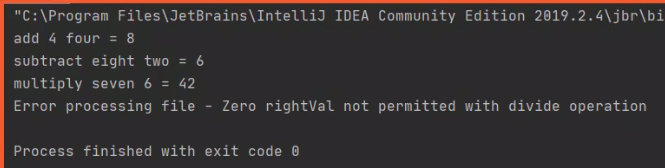


So now here we are in our main method, and as you recall, our main method already has a try that contains multiple catches. We have a catch for FileNotFoundException, IOException, and then just Exception. Now we could go ahead and add a catch for IllegalArgumentException as well. And if we want to do any kind of special handling in that case, that would be a great idea. But in this case, we simply want to report that the IllegalArgumentException has occurred.

So we'll just go ahead and let the catch block for Exception take care of our IllegalArgumentException. Now one last thing we'll want to do is take a look at our democalculations file and see what's inside there.



So now here in our democalculations file, notice that the fourth line is divide 6 by 0. So since our rightVal is 0, this will allow us to test that new code that we've added. So let me go ahead and run the application and let's see what happens.



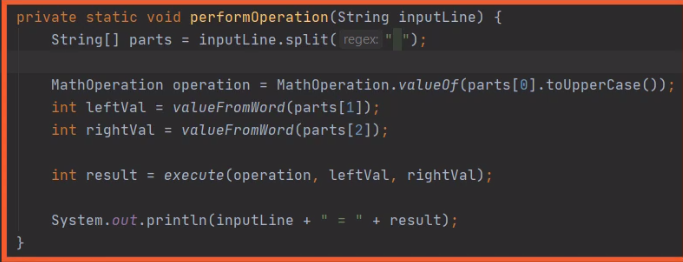
So you'll notice now when we run the code, when we go to do the divine operation, now, rather than getting a divide by 0, we get a more meaningful message. We now get Zero rightVal not permitted with divide operation. So we didn't just simply report that something went wrong, we've given the user an indication of just what happened, and that's the value of being able to throw exceptions from our code, we can report better information. So now in our next section, we'll take a look at how we can create our own custom exception types.

# Custom Exceptions

=>slides: Pg. 5

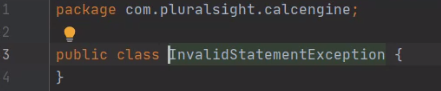
Now Java has a really rich set of built‑in exception types. And in general, those built‑in exception types will meet most of our needs, but there are times where we may need a type of exception that's not part of the built‑in exception types. Well, in those scenarios, we can go ahead and create our own custom exception, and, of course, the way we're going to do that is by defining our own custom exception class. Now, as you recall, an exception is a class, but not just any class can be an exception. Remember that all classes that represent exceptions have to inherit from the Exception class, they can either directly inherit from the Exception class or indirectly inherit from the Exception class, as long as at some point in the inheritance hierarchy, the Exception class appears. Now in my experience, in most cases, when I'm creating a custom Exception class, I'm going to directly inherit from the Exception class itself. And of course, what that means, if we're inheriting from the Exception class, then our custom exception is considered a checked exception, and that's what we would want. We're not a runtime exception, so we should be treated as a checked exception, which means that all the requirements of a checked exception that we talked about earlier in the course must be met when throwing one of our custom exceptions. Now the good news is most of the required features that we need to have as an exception are provided for us by simply inheriting from the Exception class. There is not a lot of additional work that we need to do when creating our custom exception, but there are some members we have to add. Now, in general, the only members we have to add are the appropriate constructors. We want to make it so we can create an instance of our custom exception and provide the necessary information. Now if your custom exception does require additional members such as methods, getters, centers, whatever, you're completely free to add those, but in general, the only thing we need to provide are the appropriate constructors. So to see what all this looks like, in our next section, let's create our own custom exception.

# Declaring a Custom Exception

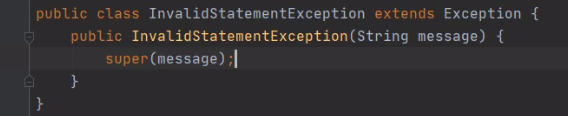


Here we are back in our CalcEngine project, and what we want to do now is see how we can create our own custom exception. So now we're here in our Main class, and we're looking at our performOperation method.

Remember that this method is responsible to actually take the user's input statement and break it up into the appropriate parts to identify the operation and the corresponding values. So, of course, the first thing we need here is to make sure we have the right number of parts. If the user doesn't provide exactly three parts, then we know we have a problem. A great way to indicate that problem would be to use an exception. And there really isn't an existing exception that would work well for us here. So this is a great chance for us to add our own custom exception to represent problems related to our application's processing of a particular statement. So let's go and add a class to represent our exception.

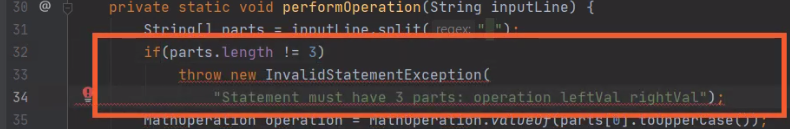


So to create our Exception class, we'll go over here to our project window, we'll right‑click on the package name, we'll head up to New, and then we'll go and choose Java Class. Now, of course, we need to give our class a name. And by convention, the name of any class representing an exception ends with the word exception. So let's name our class InvalidStatementException. So now our class has a name, so we'll go ahead and create it. So now here we have our class, but at this point, it's just a regular old class. It's not really a valid exception class. Remember that in order for a class to represent an exception, it needs to inherit from the Exception class at some level. And most commonly, we go ahead and just extend the Exception class itself.



So let's go ahead and do that. And then once we extend the Exception class, we need to add whatever members we might need for our custom exception class. And as we mentioned earlier, in most cases, the only thing we need to add are the appropriate constructors. And oftentimes, the only constructor we need is a constructor that accepts a string to receive a message that describes the problem. So let's go and add a constructor that accepts a string named message. So now we have a constructor receiving a string message. The only thing left for us to do is call the super class' constructor, passing in that message. And believe it or not, that's it. All we had to do was declare the class, extend the Exception class, and provide the appropriate constructors. So in our next section, let's see how we can use our new custom exception class.

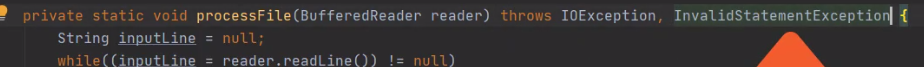
# Throwing a Custom Exception



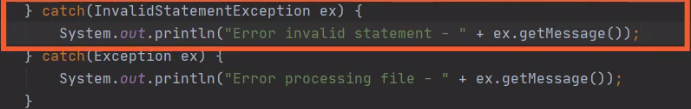
Here we are back in our CalcEngine project, and we're now ready to use our custom exception. So again, we're here in our are performOperation method within our Main class. Remember, the thing we want to do is verify that our parts array has exactly three members in it, because if it doesn't have three members, we're going to go ahead and throw our new InvalidStatementException. So let's go ahead and add an if to check to see if the parts.length is not equal to 3. And if the length of our parts array is not equal to 3, that means we have a problem. So we'll go ahead and throw our InvalidStatementException, passing in an appropriate message. So now we have the code in place so that if our parts.length is not equal to 3, we throw our InvalidStatementException, passing in a message saying the statement has to have three parts; operation, leftVal. and rightVal. But now notice our throw statement is all underlined in red, and the reason for that is that our new custom exception is actually a checked exception, which means that any method that doesn't handle this exception needs to indicate that the exception might be thrown.



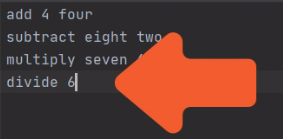
So up here at the top of our performOperation method, we need to add a throws clause for InvalidStatementException. So now performOperation indicates that it might throw an InvalidStatementException, but now look at the method above it, processFile. Well, that method calls performOperation. So since performOperation can throw an invalid statement exception, processFile also might throw an InvalidStatementException.



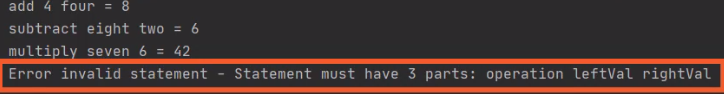
So we need to go to the throws calls here on processFile, and in addition to IOException, we also need to indicate InvalidStatementException. So now both performOperation and Process file indicate that they might throw an InvalidStatementException. So the next thing we want to do is head up to our main method and make sure we handle this exception. So let's scroll up to our Main method. So now we're up here in our Main method, and we have this existing try with several catches.



Let's go ahead and add a catch for our InvalidStatementException. And if we catch that exception, we'll go ahead and display an appropriate error message. So now we have our catch block in place for our InvalidStatementException displaying an appropriate error message. So let's go ahead and test out our code.



Now to do that, we'll go up here to our demo calculations file, we'll switch over to that file, and let's go to our last statement here where we divide 6 by 0, and let's just delete off the 0 at the end. So now with that, that last statement doesn't have the appropriate number of parts. So when we run our code, we should see our InvalidStatementException get thrown. So I'll go ahead and launch the application.

So now when we run our code, we see we get that error message there at the end, saying we have an invalid statement, that our statement must have three parts. So that shows that we've successfully created an instance of our custom exception, thrown it, and handled that exception. So now in our next section, let's take a look at how we can chain exceptions together.

# Chained Exceptions

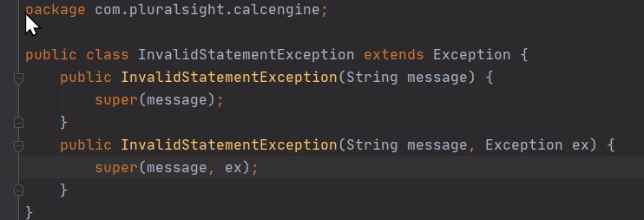
=>slides: Pg. 6

Now one of the options we have when working with exceptions is we have the ability to chain exceptions together, and what that means is we have the ability to create one exception that, in effect, wraps another exception. Now, of course, the question comes up, well, why would I want to do that? Why would I want to chain exceptions together? Well, what comes up is there may be situations where an underlying error occurs, which causes an exception to be thrown, but you don't want to directly throw that underlying exception. You'll actually throw a more meaningful exception, maybe something that's more representative of the application‑level work you're trying to do. So you want to throw that more meaningful exception, but you don't want to lose the underlying exception that represents the error that actually occurred. So a chaining exception allows us to throw that more meaningful exception, but maintain the information from the original exception.

=>slides: Pg. 7

Now chaining exceptions together is actually quite simple. We have a couple of options. One way we can chain one exception to another is to use the initCause method we inherit from the Exception class. So in this scenario, we create our new exception, then on that instance, we call initCause, passing in the original exception. And doing it that way is just fine, but in practice, we generally don't do that so much. More commonly what we do is we use a constructor to pass in the original exception. So in the process of creating the higher‑level exception, we can pass in the original exception to that constructor. Most exception types include just such a constructor. And in general, while we're creating custom exceptions, we'll also want include this sort of constructor. So once we chain these exceptions together, how do we access the chained exception? Well, the Exception class provides a getCause method. So calling getCause on the current exception instance will return back an instance of that underlying exception. So to see how this works and how we can use it, in our next section, let's jump back into CalcEngine, and we'll see how we can chain an underlying exception to our custom exception.

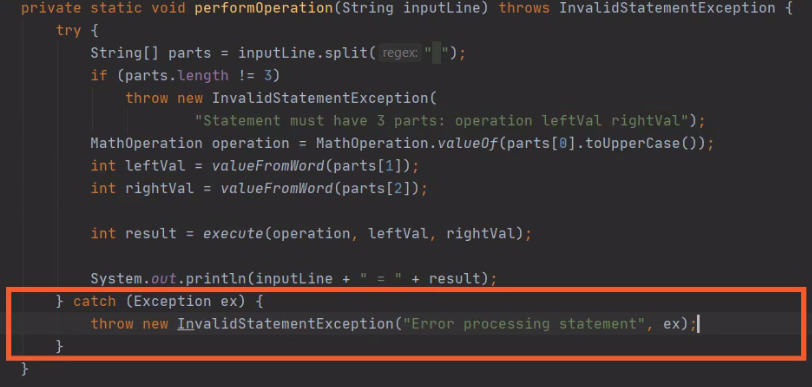
# Chaining Exceptions Together



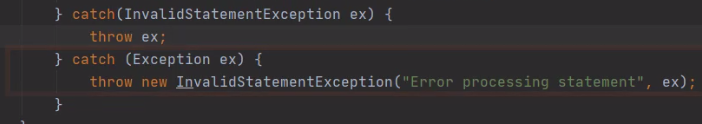
Here we are back in our CalcEngine project, and what we want to do now is see how we can use one exception to wrap another. And let's say we want to add that capability to our custom exception type here, InvalidStatementException. Well, the way we commonly do that is by starting out by adding another constructor that, in addition to accepting the string message, also accepts another exception. And then, within that constructor, all we're going to do is call our super class constructor, passing in the message and the exception. And that easily, our InvalidStatementException, now has the ability to wrap another exception. So let's see why we might want to do that. So let's head over here back to our Main class, and here in our Main class,



we're, again, looking at our performOperation method. Remember, the purpose of this method is to take the input statement, break it into its three parts, and then convert those parts into an operation, a leftVal, and a rightVal. Now as this processing is going on, some of the work we're doing has the potential to throw an exception as part of the value conversions. For example, if the user enters an invalid operation, the attempt to convert that to an enum will throw one of the built‑in exceptions. In addition, an attempt to convert the leftVal or rightVal will throw one of the built‑in exceptions that the value provided is not a valid numeric value. And if one of these built‑in exceptions were to occur, it would be nice to actually wrap that in an InvalidStatementException to provide more context for what has happened.

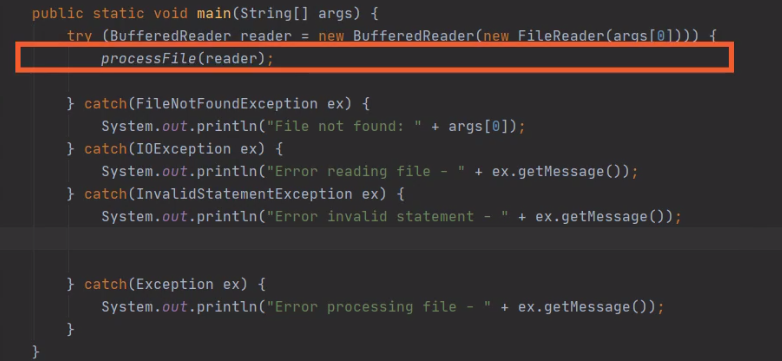


So to allow us to do that, let's take all the code here inside of performOperation, and let's put that inside of a try block. So now the entire body of the method is inside of our try block. So let's go ahead and catch the exception type itself. So now we have a catch block, so we'll catch any exception that might be thrown. And what we'll do here is go ahead and create a new InvalidStatementException, passing in an appropriate message, as well as that original exception. So now with any exception that occurs within performOperation, we'll catch that exception and wrap it in an InvalidStatementException. And that's going to work really well for most scenarios, but there's one situation that's not ideal. Remember that inside of performOperation, we already have this code that checks to see if the number of parts is not equal to 3. And if that's the case, we throw an InvalidStatementException. Well, as our code's currently written, when we throw that exception, we're going to catch it in this catch block, and then what will we do? Well, we create another InvalidStatementException that wraps the original InvalidStatementException, and that doesn't really make a lot of sense. if we're throwing an InvalidStatementException already, we should probably just go ahead and let that exception bubble up to the caller, and it turns out we can actually do that pretty easily.

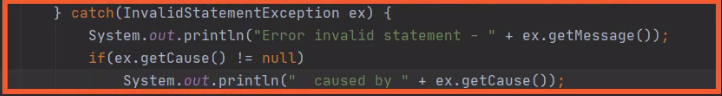


What we'll do is we'll go up here, and we'll add another catch block, and we'll catch InvalidStatementException. And when we catch the InvalidStatementException, all we're going to do is throw that exception again. And now that'll give us the behavior that we want. If we throw the InvalidStatementException, we catch it and simply rethrow it. In other words, we allow the code that calls performOperation to deal with that exception. But if any other exception occurs, we catch that exception, create our InvalidStatementException, and have it wrap the original exception. So now that we have the ability to wrap the original exception and our InvalidStatementException, in our next section, let's see how we can access that exception.

# Accessing a Chained Exception



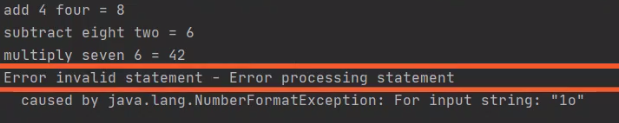
Here we are back in our CalcEngine project and what we want to do now to see how we can access an exception that's wrapped by another exception. Now we're current looking at our main method. Remember our main method calls code that actually can throw an InvalidStatementException, which in some cases will wrap another exception. So what we'll do is go to our catch block here that catches the InvalidStatementException. Now, of course, we'll continue to print out the message that's part of the InvalidStatementException, but the code that we want to add is to check to see if there is another exception wrapped by our InvalidStatementException.



The way we'll do that is by using the getCause method and checking to see if the value it returned is now equal to null. So if getCause returns a non‑null value, that tells us that there is another exception wrapped by our InvalidStatementException because getCause returns back a reference to that wrapped exception. So what we'll do is add another print statement that indicates that are InvalidStatementException was caused by something else and we'll print out the information from that contained exception. So now when we catch the InvalidStatementException, we print out our initial error message, we then check to see if another exception is wrapped by this InvalidStatementException, if it is, we print out information saying caused by and display the information for that contained exception. So let's see how this would affect our code when we run it. So let's head over to our democalculations file.



We have this last line here which currently is divide six, which we know, of course, would create an error because there aren't enough parts. So let's change this into divide 60 by 10, but you'll notice that for the value of 10 rather putting 1‑0, I put a 1 followed by the letter O. Now, when we try to convert that into a number, it's not going to work. That will cause an initial exception that we will then catch then wrap in our invalid statement exception. Let's go ahead and run the code and see what it looks like.



Now you notice that when we run the code, we first get the error message related to our InvalidStatementException, but then after that, we get a message that says caused by NumberFormatException related to the input string 1o. So that shows us that we successfully wrapped an exception with are InvalidStatementException, handled the InvalidStatementException, and then printed out the information not just about the InvalidStatementException, but also the exception that we had wrapped.

# Summary

=>slides: Pg. 8

To wrap up, here are some of the key things you'll want to remember about this module. Remember that although much of our discussion throughout this course about exceptions has been in handling exceptions, remember, we also have the ability to throw exceptions, which allows our code to indicate issues or errors that might occur during processing. The way we do that is by using the throw statement. Now remember that exceptions are ultimately classes. So before we can throw an exception, the first thing we need to do is create an instance of the class that represents the exception. Once we create that instance, we can simply throw it. Now Java has a really rich set of built‑in exception types. In most scenarios, those built‑in exception types work really well for the things we want to deal with, but there are times where we need an exception that's more specific to what we're trying to do.

=>slides: Pg. 9

And for that reason, Java allows us to define our own custom exception types. Now because exceptions are just classes, we're going to create a class to represent our custom exception, but remember that class has to inherit from the Exception class. You can inherit from exceptions either directly or indirectly, but in my experience, most commonly, we inherit directly from the Exception class itself. Then once we define the class representing our exception, we need to add the appropriate members. And most commonly, the only members we need to add are the constructors. Most of the behavior that we need to be an exception are provided by that exception base class. But if your exception type does require additional members, you are free to add those. There are no limits on what kinds or how many members you can add to a class that represents an exception.

=>slides: Pg. 10

And finally, remember that exceptions can be chained. In other words, one exception can wrap another. What that allows us to do is if, in the course of processing, an underlying exception occurs, rather than have that exception be thrown directly, we can actually wrap it in a more meaningful exception that indicates the actual kind of work that we're doing. But as we throw that more meaningful exception, we don't want to lose the underlying exception that actually created the error. So by chaining exceptions, we can provide this higher level, more meaningful exception, but contain within that exception the details of the exception that actually occurred while doing the work. Now the way we chain exceptions together normally is to simply pass the exception to the constructor of the new exception that we're creating. And then by passing it to our super class constructor, that takes care of wrapping the original exception. Remember, there is also in an initCause method that will allow us to set a contained exception, but in my experience, the overwhelming majority of the time, we simply pass the original exception to the constructor. But, of course, if one exception wraps another, in many cases, we want to be able to access that contained exception. And we do that by using the getCause method. So the getCause method returns a reference to the contained exception. If getCause returns null, that tells you that the current exception does not wrap another exception. But if it's not null, you know it does wrap another exception. You can get a reference of that wrapped exception by simply calling getCause. Alright, that wraps up this module.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*