CS 240: Exceptions – Exceptions in Java Transcript

This video shows a split screen of Professor Rodham on the right and a PowerPoint screen on the left. The left screen will often switch to IntelliJ. Any text displayed or action performed that is not verbalized will be included in italics as visual descriptions.

[00:00:00] **KEN RODHAM:** In this video, we're going to talk about exception handling in Java.

[00:00:04] Hopefully you've been exposed to exceptions and exception handling before.

[00:00:10] If you learned C++ or some of the language in a previous class, you should have had been exposed to this idea.

[00:00:16] What we're really talking about here is error handling.

[00:00:19] Generically speaking, error handling is a big part of computer programming.

[00:00:23] The idea here is that while programs are running, things will go wrong, and when things go wrong, we need to handle that appropriately in our code.

[00:00:34] A big part of writing good software is contemplating what things could go wrong while the program is running and then writing code that handles those possible error conditions so that if they happen, something reasonable happens.

[00:00:47] What we don't want to do is crash.

[00:00:50] We don't want programs that crash.

[00:00:51] Really, crashing is never acceptable, and so we want to avoid crashing.

[00:00:57] At a minimum, if an error occurs, we would at least like to do what's called a graceful exit, which is to give the user some feedback and tell them that something has gone wrong and that we can’t continue, and then shut down the program.

[00:01:12] Much better than that would be to actually recover from the error.

[00:01:16] If something goes wrong, we can actually take action to recover from the error and have the program keep running.

[00:01:23] That would be ideal, but both of those are better than just crashing, and that’s the idea here.

[00:01:31] Now if you think about the errors that can occur while a program is running, I’m going to talk about two large classes of errors.

[00:01:40] The first is errors that are caused by bugs in the code.

[00:01:45] Now ideally, we would never have these kinds of errors, but in fact, programs do have bugs and so these errors do occur.

[00:01:53] One class of errors is caused by bugs in your code.

[00:01:56] The other class of errors is errors that occur even if your program is perfectly written.

[00:02:03] Even for a perfectly written program, bad things can still happen while it’s running.

[00:02:09] For example, maybe it tries to open a file and the file is not there. That’s an error.

[00:02:16] Where did the file go, who deleted it, who moved it? I don’t know. All I know is that right now, that file is not there and I can’t open it.

[00:02:24] Maybe the program’s perfectly written, but that can still happen.

[00:02:28] Another example would be maybe my program uses the internet, which many programs do these days, and I tried to access maybe a web resource or something on the internet, and for whatever reason, the internet's down right now.

[00:02:42] Well, that can happen and it often does happen, but it's not the program's fault. Doesn't mean the program is poorly written, it just means the internet's down.

[00:02:49] You need to write your code so that you can handle errors like that that aren't actually caused by bugs in your code.

[00:02:56] Another example would be running out of memory.

[00:02:59] Maybe you call 'new' to create an object and they throw an out-of-memory exception.

[00:03:03] The Java Virtual Machine throws an out-of-memory exception and you're like, "Wow, I'm out of memory.

[00:03:08] It's not my fault. I didn't do it, but I still need to do something reasonable in that case." We're going to distinguish between errors that are caused by bugs and errors that are not caused by bugs.

[00:03:25] All right, the way I want to do this is, I want to use some code examples to talk through how we do exception handling in Java.

[00:03:35] The idea here is that when a program starts running, it starts executing in its main method.

[00:03:43] Then main will call another function, which will call another function, which will call another function, which will call another function, and every time a new function gets called, the runtime stack gets deeper and deeper and deeper.

[00:03:54] Maybe you’re down 30 or 40 levels in your runtime stack and that’s usually the code where you would detect that an error occurred.

[00:04:03] For example, where would the code be that detects that the internet’s down? Well, the code that actually detects that the internet’s down would actually be down in the bowels of the operating system somewhere, Linux or Windows or whatever it may be, and of course, that code in the operating system was called by some class in Java’s library.

[00:04:23] The kind of code that actually detects a lot of errors is very low-level.

[00:04:27] It’s down in the bowels of the Java library or in the operating system.

[00:04:33] That code knows exactly what went wrong.

[00:04:36] It knows the internet is down right now or it knows the file does not exist.

[00:04:40] But that low-level code has no idea what to do about it because it doesn’t have enough context, it doesn’t even know what application it’s part of, it doesn’t know how to respond to the error, it doesn’t know how to recover from it.

[00:04:51] The challenge we have is the code that detects the error is very low level and can’t really respond to the error.

[00:04:57] The code that can respond to the error is probably much higher up the runtime stack, closer to main, could be main itself.

[00:05:03] What we do with exceptions is we take all the information about the error that occurred and we package it up in an object, an Exception object.

[00:05:13] Then we take the object and we throw it up the runtime stack.

[00:05:20] While the exception is traversing the runtime stack, a process called stack unwinding occurs.

[00:05:28] What Java does in this case, when we throw an exception, is it looks for a try-catch block that would handle the exception.

[00:05:37] If the method that throws the exception doesn’t have a try-catch block, then that method would be terminated immediately and then Java would go to the next method that called the first one, the parent method, if you will.

[00:05:52] It’ll check to see if there’s a try-catch block in that method that handles the exception.

[00:05:57] If yes, it’ll handle the exception and continue processing.

[00:06:01] If not, the parent method also gets terminated and then it goes to the grandparent method call and checks for a try-catch there to see if there’s a try-catch to handle the error.

[00:06:13] If yes, it runs the try-catch and then it continues.

[00:06:17] If not, that grandparent method gets terminated and so on and so forth.

[00:06:21] What Java does is it kind of searches the runtime stack, looking for a method that wants to handle the error.

[00:06:30] That’s typically how things go when an error occurs.

[00:06:33] Create an object with all kinds of good information about the error, throw it up the runtime stack until somebody wants to handle it.

[00:06:42] That’s the way exception handling works at a high level.

[00:06:46] Another thing that’s good about exceptions is that it actually lets us separate the error handling code from the regular code in our program.

[00:06:56] It gets confusing when the error handling code is intermixed with the regular code.

[00:07:01] The nice thing about a try-catch block is it lets us put the regular code in the try block and then let's us put the error handling code in a catch block, and that physically separates the two kinds of code; it makes the code more readable.

[00:07:13] Exception handling is a very elegant way to handle errors in code.

[00:07:18] Now let's look and see how this works in Java.

[00:07:27] In Java, when you throw an exception, you have to throw an object.

[00:07:32] In C++, you can throw anything.

[00:07:35] You can throw an integer, you can throw a character, you can throw a pointer, you can throw a float.

[00:07:40] But in Java, you can only throw an object, which makes sense.

[00:07:44] In fact, you can only throw objects that inherit from this base class called Throwable.

[00:07:50] If an object doesn't inherit from Throwable, you can't actually throw it as an exception.

[00:07:55] Then underneath throwable, there's two subclasses.

[00:08:02] One's called Exception and one's called Error.

[00:08:05] It's important to understand the difference between exceptions and errors.

[00:08:09] An error is something that represents a catastrophic condition that's occurred down in the Java Virtual Machine.

[00:08:21] For example, maybe we ran out of memory.

[00:08:24] Well, that's pretty catastrophic.

[00:08:27] You’re probably not going to recover from that very well.

[00:08:30] Maybe a stack overflow exception, maybe you’ve exceeded the size of your runtime stack and you get a stack overflow exception; you’re probably not going to recover from that very well.

[00:08:39] Another common error would be NoClassDefFoundError.

[00:08:44] If your program is trying to use a class that doesn’t actually exist, and the virtual machine can’t find the class file for that class, it’ll throw a NoClassDefFoundError.

[00:08:54] Again, these errors are thrown by the Java Virtual Machine.

[00:08:59] Typically, they’re catastrophic errors you can’t really recover from.

[00:09:04] Typically, you don’t do try-catches on errors.

[00:09:07] They just happen. You want to write your code so that hopefully they don’t happen, but if they do happen, there’s nothing you can do about it.

[00:09:17] You just live with it. Typically, don’t catch an error in your code.

[00:09:21] Now, on the other side, this is usually what we focus on, with exception handling.

[00:09:25] There is a class named Exception.

[00:09:27] Most of the Exception objects that we throw and catch would be subclasses of Exception.

[00:09:35] Just realize that most Exception classes are going to inherit directly or indirectly from the Exception class. All right.

[00:09:45] Let’s look at some code where we can actually see what exception handling looks like in Java. *(Rodham switches to IntelliJ.)*

[00:09:55] This sample actually doesn’t compile at first.

[00:09:58] There’s a problem with it.

[00:10:00] Java is interesting in the sense that it’s the only language, that at least I’ve ever used, that forces you to handle exceptions properly.

[00:10:11] In a lot of languages, you can handle exceptions if you want to, but you don’t have to, but in Java, they actually look at your code and try to make sure that you’re catching exceptions that could be thrown, and they try to make sure that you’re handling exceptions properly.

[00:10:30] Your code actually won’t compile unless you’re doing it correctly in their opinion.

[00:10:34] This is why this program doesn’t compile.

[00:10:38] The reason is that in this sample code, I’ve got this method called processFile, which is called by the main method.

The following code is processFile:

```
private static void processFile(File file) {

        Scanner scan = null;

        scan = new Scanner(file);

        //…use Scanner for something useful

        scan.close();

}
```

End of code.

[00:10:49] In this method, what it does is it creates a Scanner.

[00:10:53] It wants to parse the text out of this file.

[00:10:55] It creates a Scanner object so it can parse the file.

[00:10:59] The problem is, is that if you look at the documentation for the Scanner class, which we will.

[00:11:06] Let's see, Scanner is in the java.util package. *(Rodham opens Javadocs in a browser.)*

[00:11:11] We'll jump to there.

[00:11:15] We look for Scanner.

[00:11:21] We can see that the Scanner class has a bunch of constructors on it.

[00:11:26] Let's just pick one. I think this is the one we're using.

[00:11:29] If you go to the documentation for this Scanner constructor, it says that it can throw a FileNotFoundException.

[00:11:36] If I try to open a file on a Scanner, and if the file doesn't exist, it's going to throw a FileNotFoundException.

[00:11:42] Now that makes a ton of sense.

[00:11:44] If I go back to my code *(in Intellij)*, what Java does is it looks at this code and says, oh, okay, you're creating a Scanner, and you're calling a constructor that could throw a FileNotFoundException.

[00:11:57] Then what Java does is it looks to see if I have a try-catch around that code.

[00:12:02] It checks to see if I’m handling that FileNotFoundException.

[00:12:06] In this case, I’m not handling it.

[00:12:09] It says, oh, that’s not right.

[00:12:12] You’re not handling exceptions properly.

[00:12:13] One way to fix that is to actually put a try-catch around it.

[00:12:17] I’m assuming here you’ve learned about try-catch before.

[00:12:21] This is not entirely new to you.

[00:12:23] I put the code inside the try block.

[00:12:25] This is the normal code, or the regular code.

[00:12:28] Then I put the error handling code inside a catch block.

[00:12:31] From the catch block, I’ll catch the FileNotFoundException.

[00:12:40] Now that I’ve caught the exception that could be thrown, the Java compiler is happy.

[00:12:45] It says, okay, now you’re handling the exception properly.

[00:12:48] Now my code is worthy to compile.

[00:12:51] Now what I do in the catch block is up to me.

[00:12:53] Of course, this catch block will only execute if a FileNotFoundException is thrown by the Scanner constructor.

[00:12:59] At a minimum, I would want to go ahead and print the stack trace.

The following code adds try-catch blocks to processFile:

```
private static void processFile(File file) {
        Scanner scan = null;
        try {
                scan = new Scanner(file);
        }
        catch (FileNotFoundException ex) {
                ex.printStackTrace();
        }
        scan.close();
}
```

End of code.

[00:13:04] At a minimum, when an exception gets thrown, you want to print out the error, you want to log the error somehow.

[00:13:10] Ideally, I would actually recover from the error in some way.

[00:13:13] Maybe this program can proceed without the file that is trying to open or whatever.

[00:13:20] Try to recover if you can.

[00:13:21] At a minimum, you need to log or print out the stack trace for the exception.

[00:13:27] What you don't want to do is nothing.

[00:13:31] If I catch an exception and I don’t do anything, I don’t log it, I don’t print it out, I don’t do anything, I just swallow it or I eat it, I didn’t do anything with it.

[00:13:44] This is bad programming practice, to catch an exception and then not do something with it.

[00:13:50] Because that means if that error actually ever occurs, that no one will know.

[00:13:56] It’ll be very hard to diagnose and debug what’s going on.

[00:13:59] If an exception occurs, you do want to do something useful with it.

[00:14:05] That’s one way to fix this program and make it compile, is to handle the exception in that version.

[00:14:11] Now you would only want to do that if this method is actually the right place or the best place to handle the error.

[00:14:17] But we said earlier that a lot of times, the code that detects an error is not the right place to actually handle the error because it doesn’t have enough knowledge or context to do that well.

[00:14:29] A lot of times, the method that actually throws an exception is not the right place to handle the exception.

[00:14:36] Let’s assume that’s the case here.

[00:14:38] Let’s just get rid of that try-catch that we just put in.

[00:14:41] Now it’s back to the state where it doesn’t compile.

[00:14:44] The other choice you have, if you don’t want to do a try-catch in that method, is you can put a throws clause on the method and declare that this method throws a FileNotFoundException.

Rodham removes the try-catch blocks and changes code back to original. The following code makes a change to the processFile declaration:

private static void processFile(File file) throws FileNotFoundException{

End of code.

[00:14:58] What that means is, is that this method called another method that could potentially throw a FileNotFoundException, and this method chose not to do a try-catch.

[00:15:10] That means that this processFile method now could potentially throw a FileNotFoundException.

[00:15:16] What we need to do is declare that as part of the method signature.

[00:15:20] What we're doing is we're advertising to anybody who might call the processFile method, hey, if you call this method, it's possible that you're going to get a FileNotFoundException thrown at you.

[00:15:31] That's your other option on making this code compile.

[00:15:35] You either put a try-catch around the code that can throw the exception or you put a throws clause on your method that says, I might throw it.

[00:15:45] Really, you're declaring that you didn't handle it and so whoever called you might want to think about handling it as well.

[00:15:51] Now if we look at this program, we go back to the main method that calls processFile(file).

[00:15:57] Now it's got a problem.

[00:15:59] Java is complaining about that code, saying this code doesn't handle the exception.

[00:16:04] It says, unhandled exception, FileNotFoundException.

[00:16:08] Well, this is just the same problem we had in processFile, except we've moved it up a level.

[00:16:13] In the main method, we have the same two options.

[00:16:16] We can either put a try-catch around the code

The following code puts a try-catch block in main:

```
public static void main(String [] args) {

        File file = new File(args[0]);

        try {

                processFile(file);

        }

        catch (FileNotFoundException ex) {

                ex.printStackTrace();

        }

}
```

End of code.

—you can see that that made the compiler happy—or we can declare that we didn't handle it.

Rodham removes the try-catch block and changes main back to original code. The following code shows the second option of updating the main declation:

public static void main(String [] args) throws FileNotFoundException {

End of code.

[00:16:47] Now the program compiles.

[00:16:48] The problem with this approach is that if you don't catch the exception in the main method, then this is your last chance to catch it.

[00:16:57] If you don't catch it at this point, that means the exception is going to propagate out of the main method.

[00:17:04] If an exception propagates out of the main method, that causes your program to crash.

[00:17:09] Actually, declaring or putting a throws clause on the main method is generally a very bad idea because it means if that exception occurs, the program's going to crash.

[00:17:19] Earlier, we said writing programs that crash is typically not acceptable.

[00:17:25] You'd only really do this thing if you're writing a little program that you're going to probably just use once and throw away or something.

[00:17:32] You would really never want to do this in production code. I'm going to get rid of that.

[00:17:37] I'm going to say that main should actually have a try-catch that handles the exception in this case.

[00:17:52] As we just said, one way to make the compiler happy is to simply put throws clauses on all the methods in your code.

[00:18:03] Not a good idea to do that on main though.

[00:18:05] Eventually, somebody needs to handle the errors, the idea.

[00:18:11] The other solution that we mentioned is put a try-catch around the code or around the code that can throw the exception, which is what I did in this example here.

[00:18:28] Now the try-catch, you can actually have multiple catch blocks on a try-catch.

The following code shows processFile with multiple catch blocks:

```
private static void processFile(File file) {
        Scanner scan = null;
        try {
                Scanner scan = new Scanner(File);
                scan.close();
        }
        catch (FileNotFoundException ex) {
                System.out.println("Could not find file: " + file);
                ex.printStackTrace();
        }
        catch (IOException ex) {
                ex.printStackTrace();
        }
}
```

End of code.

[00:18:35] There could be different kinds of exceptions that could be thrown by the code in the try block, and so you can have as many catch blocks as you want that would handle different kinds of exceptions in different ways.

[00:18:47] What Java does is when an exception is thrown and it’s looking for a catch block to handle that exception, it just searches the catch blocks in order, and it uses the first catch block that matches the type of the exception that was thrown.

[00:19:05] If it was a FileNotFoundException, it would use this catch block, and after running that catch block, the program would just continue running down here after the try-catch.

[00:19:18] But if the exception was not a FileNotFound, but it was some kind of IOException, then it would use this catch block and execute that.

[00:19:28] The method keeps running.

[00:19:30] The advantage of catching an exception and handling it is that method gets to keep going.

[00:19:34] It doesn’t get terminated.

[00:19:36] Now if the exception that got thrown was neither a FileNotFound nor IOException, then this method would terminate, and it would go to the caller to look for a try-catch that will handle the exception as we discussed.

[00:19:52] But you can have as many catch blocks as you want, and because it uses the first catch block that matches, you want to be careful how you order these things.

[00:20:01] Because what this really means is any exception that is a FileNotFoundException or a subclass of FileNotFoundException.

[00:20:09] Catch blocks are inheritance aware, so any subclass of FileNotFoundException would also match this catch block.

[00:20:15] The same for IOException.

[00:20:17] This would catch an IOException or any subclass of IOException.

[00:20:22] Because FileNotFound is an IOException, actually, you'd need to put that one first because if you put IOException first, you would never get to the file not found.

[00:20:31] Put the more specific exception types first and then put the more generic ones down at the bottom.

[00:20:38] In the most generic case, you would say catch exception, and that would catch any kind of exception.

[00:20:51] Try-catch is the way we handle the exceptions, and we would only do that in the appropriate place in our code that knows how to handle it.

[00:21:00] Now, there's actually an error in this code that I'm looking at right here.

[00:21:05] It does compile, but it has a problem.

[00:21:08] The problem is this, that if you go into this try-block or the program goes into the try-block, it opens a Scanner.

[00:21:16] Now, that could throw a FileNotFoundException as we've discussed but let's suppose in this case, that it doesn't do that and it opened successfully, then we're going to run some code that uses a Scanner to do whatever the program does.

[00:21:29] Then after it finishes using the Scanner, it's going to close the Scanner.

[00:21:34] Now, this is a pattern we use in programming all the time.

[00:21:37] We open a resource or allocate a resource, we use the resource, and then we close the resource or deallocate it.

[00:21:45] That’s a pattern we follow over and over and over again in code.

[00:21:49] It is very important to make sure we do close resources or deallocate them, and in this case, we’re opening a file, so we need to make sure that file gets closed.

[00:22:00] The reason that’s important is that there’s a limit to the number of files a program can open.

[00:22:04] If I don’t close the files that I open, eventually, the operating system’s going to say you can’t open anymore files.

[00:22:10] You’ve maxed out. You’ve got to make sure you close your files.

[00:22:14] Similarly, the same would go for a database connection or a network connection or any other resource.

[00:22:22] If you look at this code, let’s suppose that the Scanner opens the file successfully, and then it uses the Scanner.

[00:22:30] But let’s assume that while we’re using the Scanner, an exception happens.

[00:22:33] Maybe it’s an IOException of some sort.

[00:22:36] Something bad happens.

[00:22:38] Well, what’s going to happen when that exception gets thrown is Java’s going to jump to the catch blocks and try to find one that handles the exception.

[00:22:47] It’s going to look. Is it a FileNotFoundException? No, not in this case.

[00:22:52] Is it an IOException? Yes, it’s an IOException.

[00:22:55] Then the program would jump to the catch block here, it would execute this catch block, and then the program would continue execution right after the catch block.

[00:23:09] You’ll notice in this case that even though we opened the Scanner successfully, the Scanner would never get closed.

[00:23:16] That’s the problem with this code, is that if an exception actually does occur after the Scanner opens the file, the files are never going to get closed, and so we’re going to leave this file just hanging out there open.

[00:23:31] That’s the problem. There’s actually three possibilities in this case.

[00:23:38] One possibility is that the try-block runs without exception, everything goes well.

[00:23:46] In that case the Scanner gets closed.

[00:23:49] But the other possibility is an exception does get thrown, and in that case, the Scanner is not going to get closed.

[00:24:00] How do we fix that? That’s a big problem.

[00:24:03] In Java, they’ve got this thing called the finally clause.

[00:24:10] On a try-catch, you can also have a clause called finally.

The following code shows processFile with a finally clause:

```
private static void processFile(File file) {

        Scanner scan = null;

        try {

                Scanner scan = new Scanner(File);
        }
        catch (FileNotFoundException ex) {
            System.out.println(“Could not find file: “ + file);
            ex.printStackTrace();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
        finally {
            if (scan != null) {
                scan.close();
            }
        }
    }
```

End of code.

[00:24:18] You’d only have one of these.

[00:24:20] You can have as many catch blocks as you want, but you would only have one finally clause.

[00:24:23] What the finally clause does is it has what’s called no matter what execution semantics.

[00:24:31] Whatever code you put inside the finally clause it’s going to execute no matter what.

[00:24:35] Even if an exception gets thrown, it’s going to execute.

[00:24:39] Even if an exception doesn’t get thrown, it’s going to execute.

[00:24:43] No matter what, the code in the finally block is going to execute.

[00:24:46] What we did here is we open the Scanner on the file.

[00:24:53] We use the Scanner as before, but we took the code where we close the Scanner and we moved it down to the finally clause, the finally block.

[00:25:04] You can see here, if Scanner is not equal to null, close it.

[00:25:09] One thing you’ll see here also is that we took the Scanner variable and we lifted it outside the try-catch.

[00:25:16] The declaration of the Scanner is actually before the try now, and that’s so it’s visible in the finally block.

[00:25:23] We declare the Scanner outside the try.

[00:25:27] But otherwise, the code’s just like it was before, except we moved the code that closes the Scanner into the finally.

[00:25:34] What this means is that now the Scanner’s going to get closed no matter what.

[00:25:41] What are the possibilities? Well, one possibility is the try-block runs with no errors.

[00:25:47] But after the try-block finishes, the finally block is going to execute.

[00:25:53] Great. That’s what we hope happens most of the time.

[00:25:57] But the other possibilities we said is an exception does get thrown in the try-block, and in that case, the finally clause is still going to execute.

[00:26:12] Before Java handles the exception, it's going to make sure the finally block runs so the Scanner gets closed.

[00:26:20] The finally clause is really important in Java programming to make sure that your resources get cleaned up properly whether or not an exception gets thrown.

[00:26:31] The try-catch finally is actually a construct you'll see in many different programming languages like C#, JavaScript, etc.

[00:26:40] Most of the modern languages have a finally clause for this purpose to make sure things get cleaned up properly.

[00:26:48] It's interesting to note that C++ does not have a finally clause because it really doesn't need one.

[00:26:55] Because C++ has something else that most languages don't have, which is something called a destructor.

[00:27:00] As you know, a destructor gets called on an object when it goes out of scope, and so you can achieve no matter what semantics using destructors in C++.

[00:27:12] They actually don't need the finally clause.

[00:27:16] But since most newer languages are garbage collected and don't have destructors, then they do need the finally clause to make sure we clean up our resources correctly.

[00:27:29] Now a couple other points on exceptions.

[00:27:35] We just talked about the finally clause that you can put on a try-catch.

[00:27:43] Try, catch, finally.

[00:27:45] You can imagine in code you’d have a lot of try-catch finally stuff.

[00:27:49] It becomes very repetitive in some sense.

[00:27:52] Creating these finally clauses to close your resources gets pretty repetitive and cumbersome.

[00:27:59] Java added a nice feature called try with resource.

[00:28:05] This example shows the same code, but it uses a try with a resource.

The following code shows processFile with try with resource:

```
private static void processFile(File file) {
        try (Scanner scan = new Scanner(File)) {
                //use Scanner for something useful
        }
        catch (FileNotFoundException ex) {
                System.out.println(“Could not find file: “ + file);
                ex.printStackTrace();
        }
        catch (IOException ex) {
                ex.printStackTrace();
        }
}
```

End of code.

[00:28:09] What you'll notice here is in the try statement here, it actually has some code inside parentheses.

[00:28:19] Inside this code here, we open the resource that we're wanting to make sure gets closed.

[00:28:24] In this case, it's a Scanner.

[00:28:26] We open the Scanner and the code that is in the try.

[00:28:34] We open the Scanner and then we use the Scanner.

[00:28:40] Rather than writing a finally clause to close the Scanner, what this will do is Java will automatically close the Scanner for us.

[00:28:49] In effect, if you use a try with resource, whatever resource you open inside the parentheses there, Java will make sure that the close method gets called on it.

[00:28:59] In effect, what Java does is it writes the finally clause for you automatically so you don't have to write it yourself.

[00:29:05] That's a much more concise and simple way to make sure your resources get closed.

[00:29:11] Use the try with resource, that's the recommended way to do it rather than writing your own finally clause.

[00:29:18] You may have other reasons to write finally clauses that don't have anything to do with closing resources.

[00:29:24] In those cases, go ahead and write your finallys.

[00:29:26] But if you want to make sure your resource gets closed, then go ahead and just use the try with resource syntax and Java will do it for you.

[00:29:33] That's a nice feature.

[00:29:43] This next version of this code shows a couple more ideas that are important.

[00:29:51] I've modified this processFile method such that it can throw exceptions.

The following code shows processFile that can throw exceptions:

```
private static void processFile(File file) throws java.net.MalformedURLException,
SomethingBadHappenedException {
    Scanner scan = null;
    try {
        scan = new Scanner(File);
        //use Scanner for something useful
        boolean invalidURL = true;
        if (invalidURL) {
            throw new MalformedURLException();
        }
        boolean somethingBad = true;
        if (somethingBad) {
            throw new SomethingBadHappenedException("The Sun
went out!");
```

```
                        }
                    }
                    catch (FileNotFoundException ex) {
                        System.out.println("Could not find file: " + file);
                        ex.printStackTrace();
                    }
                    catch (IOException ex) {
                        ex.printStackTrace();
                    }
                }
```

End of code.

[00:29:59] It has a throws clause now.

[00:30:02] This method can throw a MalformedURLException, which is built into Java, that's just a type of error that is built into Java so MalformedURL.

[00:30:12] Another type of error it can throw is a SomethingBadHappenedException, and that's an exception class that I wrote myself.

[00:30:19] What this example demonstrates is that in your own code, when you're writing code, normally exceptions are being thrown at you usually by Java, the Java library, or maybe some other library that you're using.

[00:30:33] A lot of times, exceptions are being thrown at you and so you need to catch them and handle them.

[00:30:40] But also in your own code, you can throw exceptions as well.

[00:30:44] In your own code, when detect those errors have occurred, you can actually create an exception object of your own and then throw it.

[00:30:54] This example shows that you can catch exceptions, but you can also throw them, which you should probably get in the habit of doing.

[00:31:03] For example, here in the processFile method, it checks to see if the URL is valid.

[00:31:14] If the URL is invalid, what it does is it throws a MalformedURLException.

[00:31:21] That just shows you the syntax for throwing an exception.

[00:31:24] If something bad happened, I can create my own exception object and throw it.

[00:31:28] Now in this case, MalformedURLException is built into Java.

[00:31:32] You can throw the exception classes that are built into Java, you don't have to write your own exception classes if the ones that are built into Java suffice for what you're doing.

[00:31:43] But a lot of times for your application you'll need to create exception classes that are specific to what you're doing.

[00:31:49] Maybe Java doesn't have an exception class that's right for what you're trying to do, you can create your own.

[00:31:55] In this case I created an exception class called SomethingBadHappenedException.

[00:32:02] I create one of those objects and I throw it.

[00:32:05] Because I'm throwing these exceptions, I had to declare that in my method signature.

[00:32:11] You can be a catcher of exceptions, you can also be a thrower of exceptions, which is nice.

[00:32:17] Now to create your own exception classes, it's quite simple.

[00:32:21] In this example, I've got my SomethingBadHappenedException class, which I wrote.

[00:32:27] Actually, I didn't write it, IntelliJ wrote it for me, but you'll notice that SomethingBadHappenedException, it extends Exception so it does inherit from exception.

[00:32:37] If you want to create exceptions of your own, all you have to do is create a new class.

[00:32:45] New, Java class, and I'll say myException, for example.

[00:32:53] I just create a new class, and then all I have to do is say extends Exception.

[00:33:02] Now I've created a minimal exception class that I can throw in my own code.

[00:33:07] Now you probably want to add some more stuff to it.

[00:33:10] You might want to put some variables on it that can contain information about the error that occurred.

[00:33:15] You want to probably put some good constructors on it so it can be initialized in different ways.

[00:33:21] In this example, I'd probably like to add some constructors to my exception class.

[00:33:26] I'm going to go to the code, let's say, I'm going to generate some constructors.

[00:33:34] Basically what it's going to do is it's going to say, "Hey, here's some constructors your superclass has.

[00:33:41] Would you like to have the same constructors on in this class?” In this case, I’m going to say, yeah, that’s exactly what I want.

[00:33:47] I want to have the same constructors that my superclass has because the exception superclass has a bunch of useful constructors that you want.

[00:33:57] I would typically add those to my exception class, then add whatever else you want to it.

[00:34:02] It’s really easy to create an exception class.

[00:34:09] Now the last thing that this example shows is it shows a multi-catch.

The following code shows main with a multi-catch:

```
public static void main(String [] args) {
        File file = new File(args[0]);
        try {
                processFile(file);
        }
        catch (MalformedURLException | SomethingBadHappenedException ex) {
                for (StackTraceElement ste : ex.getStackTrace() ) {
                        System.out.printf(“Class: %s, Method: %s, File: %s, Line: 
%d\n”, ste.getClassName(), ste.getMethodName(), ste.getFileName(), 
ste.getLineNumber());
        }
}
```

End of code.

[00:34:17] You can have as many catch blocks as you want on a try.

[00:34:21] That's great if you want to handle different errors differently, but sometimes you want to handle different errors in the same way, and so it's convenient if you could write a catch block that can handle multiple different kinds of exceptions, not just one kind.

[00:34:40] In this example, it shows a catch block that handles a MalformedURLException or a SomethingBadHappenedException.

[00:34:46] Because I want to handle those two kinds of exceptions identically, I don't want to duplicate that code, I'd rather just put both of those on the same catch block.

[00:34:55] Therefore, the syntax is, put the name of an exception class, then you can put an OR bar.

[00:35:00] You can have as many ORs as you want.

[00:35:04] That's how you handle multiple exceptions in the same catch.

[00:35:11] Let's see, I think that's pretty much it.

[00:35:17] Exceptions are obviously core to Java programming because the compiler doesn't force proper exception handling.

[00:35:24] Hopefully the things we've just talked about will help you properly use and handle exceptions in your code.