3. Understanding Exception Types

Contents

[1. Introduction 1](#_Toc19946)

[2. Exceptions Can Be Handled by Type 1](#_Toc13947)

[3. Checked and Unchecked Exceptions 1](#_Toc22700)

[4. Handling Multiple Exceptions 1](#_Toc20895)

[5. Handling Unchecked Exceptions 1](#_Toc11936)

[6. Exceptions and Methods 1](#_Toc19643)

[7. Handling Exceptions Across Methods 2](#_Toc27905)

[8. Summary 2](#_Toc3741)

# 1. Introduction

=>slides: Pg. 1-2

Welcome to our next module, understanding exception types. So now throughout this module, we're looking how the type system represents exceptions. And as we'll see, exceptions are actually just represented by classes, which enables us to handle exceptions by type, allowing us to handle different errors differently. Now exceptions are divided into two broad categories, checked exceptions and unchecked exceptions. And we'll see how understanding the difference between these two categories of exceptions helps us to write our exception handling more effectively. And then we'll finish up, we'll look at exceptions and methods. And as we'll see, exceptions can actually span method boundaries. So there is some important things we have to consider when dealing with exceptions within methods.

=>slides: Pg. 3

Now, as we mentioned, exceptions are represented by classes and it turns out that all the classes that represent exceptions have a common base class, which is the Exception class. Now from there, the classes actually represent different levels of exceptions. Some of these classes represent very broad categories of exceptions. Other classes represent very specific type of exceptions. So every error doesn't necessarily have its own individual Exception class and that's really a good thing because it allows us to decide just what granularity of error handling we want to put inside of our code.

=>slides: Pg. 4

Now, as I mentioned, all classes representing exceptions inherit from the Exception class. Now since this is actually a class, it's going to have the characteristics of any class because as you may recall from the course, Working with Classes and Interfaces in Java, every class has a common base class, which is the Object class. So the Exception class descends from the Object class, but it turns out it doesn't directly inherit from the Object class. There is one class in between known as the Throwable class, and the Throwable class provides the capabilities to allow the language to throw exceptions. Now there are some other classes that inherit from throwable, but in general, we don't work with those a whole lot. The main class we're interested in is the Exception class. So the Exception classes is what provides the basic capabilities to be an exception. And then, from there we'll have our other classes. For example, we have the IOException class. The IOException class is a class that represents a broad category of errors. Pretty much any error related to input or output is represented by the IOException class. But you want to handle errors more specifically, there are then exception classes that inherit from IOException. For example, there is a FileNotFoundException. That's a pretty specific exception, it's an exception that's thrown when you try to access a file and that file isn't found. Now another class that inherits from IOException is SSLException and this is kind of an interesting case because it's more specific than IOException, but still not a super specific class because it represents any exception related to secure socket layers. There are actually other classes that inherit from SSL exception that represent more specific SSL related errors. So, as you can see, there can actually be multiple layers to our Exception class hierarchy, and beyond these, there is even other classes that inherit directly from IOException. Now another important class that inherits from the Exception class is the Runtime Exception class. Now, the RuntimeException class represents the kind of errors that could be thrown directly by the runtime. For example, the ArithmeticException class inherits from RuntimeException. ArithmeticException represents errors related to arithmetic. For example, divide by 0 throws an arithmetic exception, and there are a number of other classes that inherit from runtime exception. Another example might be a null exception. That gets thrown when you try to use a null reference. And there are actually a number of different classes that inherit from the Exception class and this is a really powerful concept because it gives us a lot of control over just what level of granularity we want to use in our error handling. We'll talk more about that next .

# Exceptions Can Be Handled by Type

=>slides: Pg. 5

So with our exceptions being represented by exception classes, that allows us to be more specific about what exceptions we'll want to handle because it turns out that a try can actually have multiple catches. And each catch will indicate what kind of exception that catch wants to handle or the type of the Exception class that it catches. Now if a try contains multiple catches, Java will test those catches from top to bottom. And the first assignable catch that it finds is the one that it will use. And by assignable, what we mean is, the first catch that catches the Exception class as either the class representing the specific exception that was thrown or a base class of that exception, the first one of those it encounters is the one it's going to use. And because of that, it's important that we put our most specific exceptions first and our less specific exceptions further down the list. So to understand what we mean here, let's take a look at some code,

=>slides: Pg. 6

and what we'll do is use the same code we used in our slides in a previous module. So we'll start out by declaring two variables and setting them to have initial values. Then we'll have our try that does some arithmetic and displays the result. Now because the arithmetic here contains division, we know there's a possibility of an exception, so we're going to put a catch in place that catches Exception. So what this catch is saying by catching the Exception class itself, it says, I want this catch to handle any exception that is represented directly by the Exception class or any class it inherits from the Exception class. And since we know that every class representing exceptions inherits from the Exception class, that means that this catch will catch any exception of any type that's thrown. But now looking at our code, we know that the most common exception that would occur here is an ArithmeticException if we try to divide by 0. So let's go and add a catch to catch ArithmeticException. So we're saying with this second catch here is that if an ArithmeticException is thrown, we want that block to handle ArithmeticException. But the problem with this code is since our catch blocks are tested from top to bottom, an ArithmeticException would never reach our second catch block because ArithmeticException inherits from Exception.

=>slides: Pg. 7

So if we want to have two catch blocks, one handling ArithmeticException and then a second catch block that actually handles everything that's not an ArithmeticException, we need to switch the order of these catch blocks. So we'll start out by removing our catch for Exception, then we'll take the catch for ArithmeticException, and we'll move that up, and now we can go ahead and back the catch for Exception, we'll just add it after ArithmeticException. So with our code written out this way, the catch for ArithmeticException will handle any exception that's an ArithmeticException, and all other exceptions will be handled by the catch that handles the Exception class itself. Okay, so that's our exception classes, but it turns out that exceptions are also divided into categories, and we'll take a look at that r next .

# Checked and Unchecked Exceptions

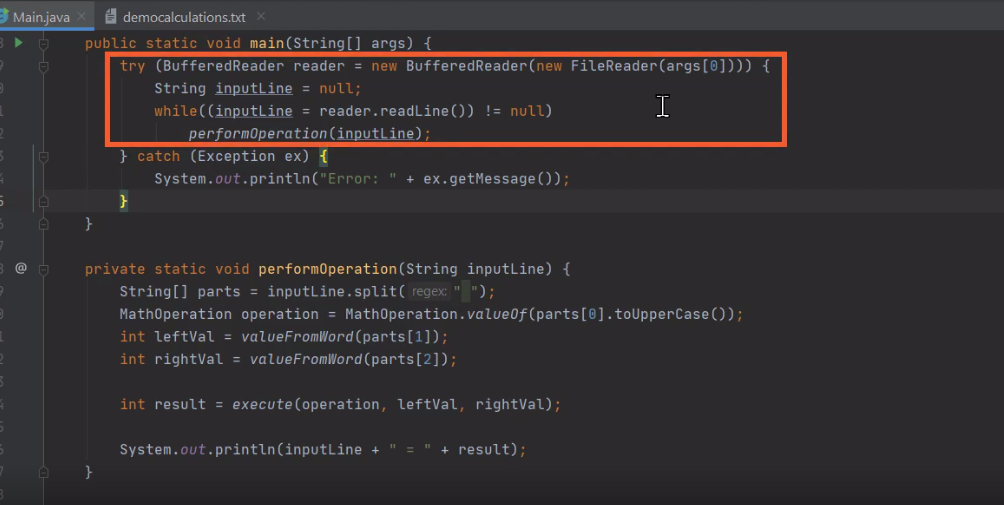
=>slides: Pg. 8

As we've seen, exceptions are represented by classes, and different kinds of exceptions are represented by different classes. Well, it turns out that these accepted classes are actually divided into two broad categories. One category is what is known as checked exceptions, and checked exceptions are exceptions where the compiler will actually raise an error if you don't handle that exception. In other words, as we're writing our code and we compile the application, the code we write has the possibility of raising an exception that's considered a checked exception. If we don't have a catch block for either that Exception class itself or one of the base classes of that exception, our code won't compile. The compiler will actually generate an error. Now the other category of exception is what's known as an unchecked exception. And as you might've guessed, an unchecked exception is an exception where the compiler does not force error handling. So in the case of unchecked exceptions, if we write some code that has the possibility of generating an unchecked exception, the compiler really doesn't care. The compiler is not going to raise an error if you don't catch that exception on one of its base classes. But here's a really important thing to remember, whether we're talking about checked exceptions or unchecked exceptions, if at runtime that exception gets thrown and we don't catch it, then our program is going to crash. The only difference between checked and unchecked exceptions is that compile time. At runtime, any unhandled exception can crash your application. So, of course, this immediately raises the question, how do I know if a particular exception is a checked exception or an unchecked exception?

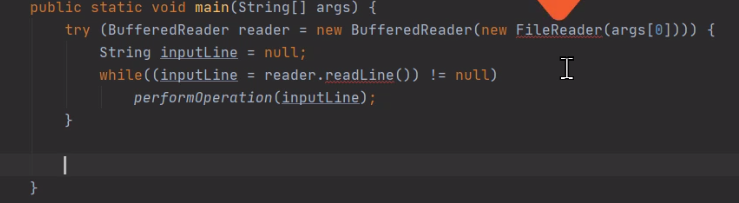
=>slides: Pg. 9

Well, it turns out that the categorization is actually pretty simple. In the case of an unchecked exception, that's any exception that inherits from RuntimeException. Unchecked exceptions are all the rest. And remember that RuntimeException represents a group of exceptions to kind of tie the things that can happen from the language, things like ArithmeticExceptions or null‑related exceptions or ArrayIndexOutOfBoundsExceptions. So what's happening here with the unchecked exceptions is that the compiler just doesn't want to burden us with having to handle all those exceptions that may never occur. But again, this is a pretty small category of our exceptions. Most of the exceptions we're going to be interested in are all the rest of them, and those are our checked exceptions. Remember, a key take away from here, though, whether an exception is a checked exception or an unchecked exception, if it happens at runtime and you don't catch it, your program will crash. Alright, so now in our next section, let's jump back into our Java code, and we'll work on the application we started working on in our previous module to enhance our exception handling to take advantage of the new information we know about exception classes and exception categories.

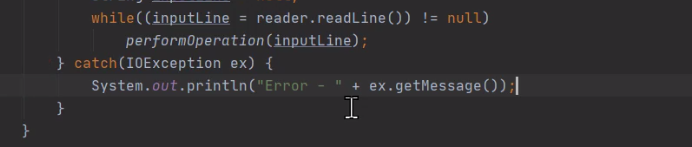
# Handling Multiple Exceptions



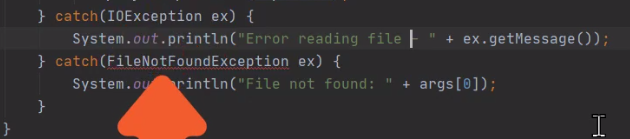
Here we are back in our STS IDE, we're looking at the same project we were working on in our previous module, and in fact, our code is in the exact same state we left it in at the end of that module. Now, as you recall, we have our try with resources that uses a buffered reader to read through a file that allows us to process the equations that are in that file. Because it's a try with resources, it will take care of closing the reader for us when we're done, and for dealing with exceptions, we have this one catch, which catches the Exception class. And since the Exception class is the base class of all exceptions, this one catch will handle any exception that might occur. And that's fine for making sure that we take care of any error that might come up, but the one challenge with having just this one catch that handles the Exception class is that we don't have the chance to handle individual errors that might come up in more specific ways. So let's remove this catch that just handles Exception.



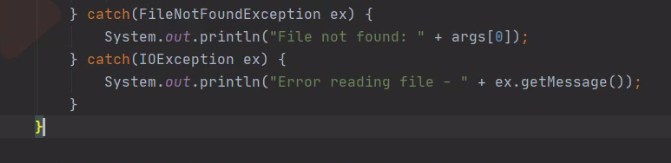
Now you notice that once we remove that catch, some of our code is now underlined in red, indicating that there are errors. So now one of the errors we have is where we create the FileReader. So if I move my cursor up there and hover, you'll notice that it says, Unhandled exception, FileNotFoundException. Well, the issue is that FileNotFoundException is a checked exception, so we're responsible to handle that exception. And there's where we call reader.readLine is also underlined in red. If i head down there and I hover over top of that, again it says Unhandled exception, IOException. Again, IOException is a checked exception so the compiler is requiring us to handle that exception. Now as we talked about, some exception classes inherit from other exception classes, and we know that FileNotFoundException inherits from IOException. So something we could do here is simply put a catch in place for IOException. So let's go ahead and do that.



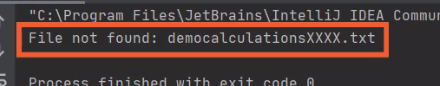
So now we have our catch in place for IOException. You notice that once we do that, the code that was previously underlined in red, the creation of FileReader, and the call to reader.readLine are no longer underlined in red. So we've taken care of the requirement by the compiler to actually handle those exceptions. But having just this one catch for IOException still doesn't really solve the problem we were trying to solve. We wanted to be able to provide more specific error handling. Right now, we're just catching IOException. It might have happened because we created the FileReader, it might have happened because we were reading from the file itself. So in this case, even though handling IOException met the compiler requirements, it might still be a good idea to actually handle FileNotFoundExceptions separately from IOException. So let's go now to catch for FileNotFoundException.



So now we have our two separate catches, one for IOException, and one for FileNotFoundException. And by doing that, we're able to provide more specific error information. In the case of FileNotFound, we can actually print out the words FileNotFound, and show the file name. In the case of IOException, we know that's related to reading the file. So again we could print out a more specific message related to reading from the file. But there's still one problem with our code. Notice that FileNotFoundException is underlined in red. And if I go and hover over that, notice that I have the error message that says this exception has already been caught. In other words, the compiler is telling me that I already have a catch in place that handles a base class of the FileNotFoundException. So there will never be a chance for this catch to ever actually run. So what we need to do is actually switch the order of these two catches.

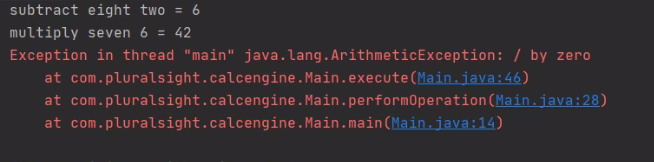


So let's move our FileNotFoundException before our IOException. And now we have our two separate catch blocks placed in correct order. So let's go ahead and run our program now, and we'll verify that it behaves as we expect. Now I've actually already configured the project to have an invalid file name.

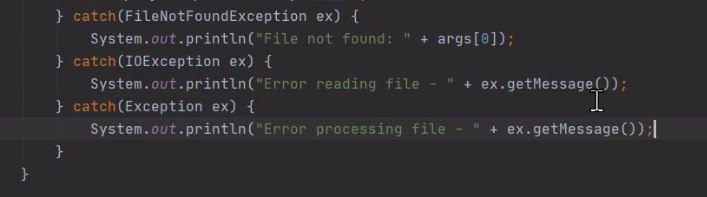


So the code for our FileNotFoundException should run for us. And as you can see, when our code runs, we get that message, File not found, showing the file name. So we've made some real improvements to our exception handling, but there's still one other issue we need to address, and we'll do that in our next section.

# Handling Unchecked Exceptions



Here we are back in our application, and what we want to do now is make sure we have all the necessary exception handling in place. Now as you recall in our previous section, we put in catch blocks for FileNotFoundException and IOException, and that allowed us to compile and run our program because we were sure to handle any exceptions that the compiler indicated that we had to handle. But the question is, does that assure that we've handled every exception that we need to handle? Well, the easiest way to know is try running our program. Now I've already changed the configuration of our project to now point to our democalculation file, which is the file that has the calculations that we want to run. So let me go ahead and execute the program, and let's see what happens. And you'll notice when the program runs, the program actually crashes. It crashes because of an arithmetic exception for a divide by zero. And that issue came up because if we look at our democalculations file, one of the operations in there causes a divide by zero, and this is the issue of an unchecked exception. In other words, even though there was a possibility in our code for a divide by zero to happen, because the arithmetic exception is an unchecked exception, the compiler did not force us to handle it. But that doesn't mean we're not responsible to handle it. If we know there's a possibility of this exception occurring, we've got to be sure to have a catch block for it. So let me go ahead and collapse the run window here at the bottom and move over to our Main.java file. So now we're over in the main method of our Main.java file, and we want to be sure that any exceptions that occur don't crash our program. Now one except we encountered was our arithmetic exception. So one thing we could do is put a catch in for that specific exception. And if we did that, that would take care of our divide by zero. But then, what if some other unchecked exception eventually reared its head as our program was running with other data?

 So a better choice for us rather than just handling that one exception, arithmetic exception, is instead to add a catch for exception as the last catch on our try block. So let's go ahead and do that. So with the addition of that catch block, our application now does a solid job of handling the exceptions. We've got catches for typed specific exceptions that we know might come up, but then we also have the catch for the Exception class to be sure that we handle any other exceptions that might occur during the execution of our program. All right, so now in our next section, let's take a look at the relationship between exceptions and methods.

# Exceptions and Methods

=>slides: Pg. 10

Now when it comes to dealing with exceptions within methods, it turns out that an exception thrown in one method can actually cross the method boundaries. So if an exception is thrown within a method and that method doesn't handle the exception, the exception will actually travel up the call stack. And this has some really important applications on how we organize our exception handling because it turns out because these exceptions can travel up the call stack, it allows an exception thrown in one method to be caught by a different method that called into the method that threw the exception. So let's see visually what this looks like.

=>slides: Pg. 11

Let's add a method here, methodA. Remember that within the system, each method has space on the call stack, and the call stack is where we actually have information related to the method, things like local variable allocation and that sort of thing. So here in methodA, we'll put in a try catch block to handle an exception. And then here in methodA, we'll call methodB from within methodA's try block. So then we have methodB. MethodB allocates space on the call stack. MethodB does some work. Then after it does that work, it calls methodC. So again, we repeat the process again. We have methodC. MethodC gets space on the call stack. But now, as methodC does its work, methodC does some work that throws an exception. So on our call stack, it's down to the methodC area where the exception actually occurs. But now methodC doesn't handle the exception. So that means the exception is going to start traveling up the call stack. And if no one on the call stack handles the exception, then our program would ultimately crash.

=>slides: Pg. 12

But in this case, as we travel up the call stack, as we reach methodA, we actually have our catch block. So that means that methodC and methodB were both totally unwound, and we're now up in the catch block in methodA, and that gives methodA the opportunity to now do the work related to handling that exception. So now thinking through that example, there's something really important we need to understand.

=>slides: Pg. 13

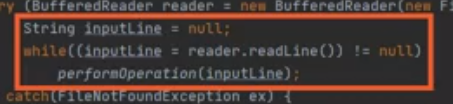
MethodA was responsible to handle an exception that methodA didn't generate. And not only was the exception not generated by A, it wasn't even directly generated by a methodA call. It was actually generated a few methods down in the call stack. So how does methodA know it needs to handle that exception? Well, it turns out that exceptions are actually part of a method's contract. So each method is responsible to deal with any checked exceptions that might occur. Now one way to deal with a checked exception is to actually have a catch in place to handle that exception. Well in the example we just looked at, that was in the case. So in the case of checked exceptions that might occur within a method, if the method is not going to catch the exception, then the method is responsible to document that the exception might occur. And the way a method does that is by using the throws clause. A method is responsible to list using its throws clause any checked exceptions that might occur that the method doesn't actually handle So now to get a better understanding of all this, in our next section, let's jump back into our Java project. We'll see how to deal with all this in our code.

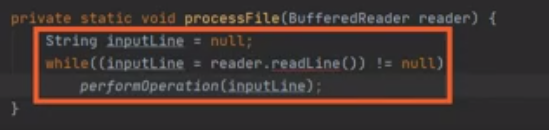
# Handling Exceptions Across Methods

Here we are back in STS, and what we want to do now is see the effect on our exception handling of moving some of our code into a separate method. Now the way our application is currently set up, all the code related to opening the file and reading the file is all up here in our main method. But now let's say we want to split part of this code out. Let's say we want to take the code that's related to actually processing the file and move that into a separate method. So what that means is these three lines here where we declare inputLine, read from the file, and then call performOperation, we want to move into a separate method. So to do that, let's go ahead and declare a new method named processFile.



Now our processFile method will actually need to accept a parameter, which is the buffered reader we want to read from. So now with that, we have our processFile method and we can receive a reference to the reader that we want to read from. So now let's take these three lines related to reading the file and let's move them down into our processFile method.





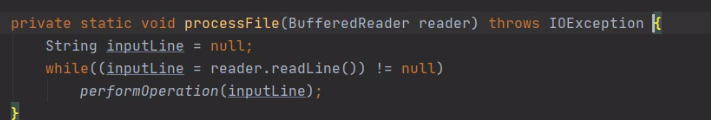
So now with that, we have all the code related to processing the file down here in our processFile method. So what we'll do then is head back up to our main method, and just after we opened the reader, let's call processFile passing in the reader.



So now we have our call to our processFile method in place, and then down on our processFile method, we do the actual work. But looking at our processFile, notice that the call to reader.readLine is underlined in red.



And as we know, reader.readLine can actually throw an IOException. And remember that any exceptions that incur within a method are the responsibility of that method. So we have a couple of choices here, one choice is to actually put a try/catch in place here inside of processFile that handles IOException, and in some scenarios, we might want to do that. But in this case, instead of handling it right here in processFile, we want to allow the method that calls processFile to handle the exception. So what we have to do is document that processFile might throw an IOException.



So the way we do that is that as part of the method declaration right after our closing parentheses, we're going to add a throws clause. So the throws clause is responsible to document any exceptions that might come out of this method. So after the word throws, we provide a comma‑separated list of each exception that might come from this method. Now, in our case, there is only one exception, IOException, so we'll simply say, throws IOException, and notice that once we do that, reader.readLine is no longer underlined in red because our method processFile is dealing with this responsibility to document that the exception might come out of this method. Now what that does mean is any time the processFile is called, the method that calls processFile has to deal with that exception, and in our case, up here in our main method, we already have a catch block in place that will deal with that exception, so we're all set. So our code is easily able to separate the work into the appropriate methods, but still centralize our exception handling up here in our main method.

# Summary

=>slides: Pg. 14

To wrap up, here are some of the key things we want to remember from this module. Remember that exceptions are represented by classes, and they're actually organized in a class hierarchy with all exception‑related classes inheriting from the Exception class. So it's the Exception class that provides this fundamental capability of being an exception. Now because exception classes are organized in a hierarchy, some exception classes represent broad sets of errors, other exception classes represent more specific errors, and this is helpful because it allows us to organize our error handling in broad chunks or more specific chunks. And what makes that possible is the fact that exceptions can actually be handled by type. Remember that one try can actually have multiple catch blocks. And when we have multiple catch blocks, those catch blocks are tested in order from top to bottom.

=>slides: Pg. 15

But a key thing to understand here is that when we put a catch block in place and we specify the exception type for that catch block, that catch block will now handle that specific exception type, as well as the types of any exceptions that inherit from that class. So for that reason, we organize our catch blocks with the most specific exceptions first and the less specific exceptions further down the list. =>slides: Pg. 16

Remember that Java divides exceptions into two broad categories. One category is checked exceptions. Checked exceptions are exceptions where the compiler will raise an error if you do not handle the exception. So in the case of checked exceptions, the compiler requires you to deal with those exceptions. But remember, we also have unchecked exceptions. Unchecked exceptions are exceptions where the compiler does not enforce error handling. Again, these are generally exceptions related to the runtime, things like ArithmeticExceptions or null‑related exceptions or ArrayIndexOutOfBoundsExceptions, those sort of things. So a compiler does not require you to handle these kinds of exceptions. But keep in mind, if one of these exceptions does get thrown while your application is running and you don't handle the exception, your application will crash. So even in the case of unchecked exceptions, you want to make sure you have the code in place to deal with these exceptions if anything might occur during your application's execution. So we finished up with a look at exceptions and methods.

=>slides: Pg. 17

Remember that exceptions can actually cross method boundaries, and this is a really useful capability because it allows us to centralize our exception handling. We can have code that calls a method, and if the exception occurs within that method, we can actually catch it in the code that called the method. But that does put a bit of a burden on the method itself because remember that any checked exceptions are part of that method's contract. So in the case of checked exceptions, the method has the option to either catch the exception, but if it doesn't catch the exception, then it has to document that exception might occur, and it does that by using the throws clause.

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