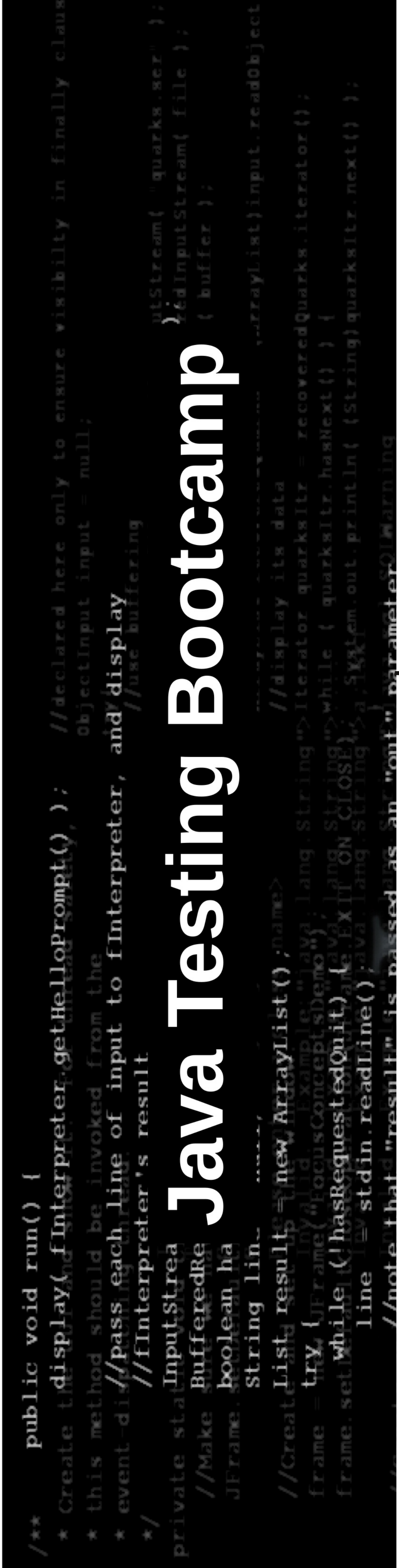


Java Testing Bootcamp

Module Nine

Exceptions

A rule without exceptions is an instrument capable of doing mischief to the innocent and bringing grief -- as well as injustice -- to those who should gain exemptions from the rule's functioning.
Derrick A. Bell

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.
Douglas Adams

Every rule has an exception, including this one.
Anonymous

Java Bootcamp for Testers

9.1 Introduction: Exceptions, Errors and Faults

The following are standard quality engineering definitions:

1. Failures: Where a program does not perform to spec.
2. Faults: Mistakes (bugs) in code that cause failures.
3. Error: The reason the programmer made the mistake.
4. Exceptions: Failures in the operating environment that cause failures in the program.

Errors and exceptions both cause failures, however the critical difference is that we can eliminate errors by changing the code that contains the fault, but we cannot eliminate exceptions – the only thing we can do is respond to them to try and recover from the situation in some manner. Often, we are not able to recover form the exception, and the only option available to us to exit the program in a graceful manner.

Consider the following non-programming example. I've decided to fly from Toronto to Montreal for a business meeting. Consider the following possible scenarios.

1. Error: I miss my flight because the schedule says the plane departs at 19:00, which I think is 9 pm when in fact it is really 7 pm. I show up 90 minutes early at 7:30 pm to see that my plane left 30 minutes prior.
2. Fault: I am fully aware that I need to be at the airport at 5:30 pm, but when the taxi pulls up to the terminal, I suddenly realize I have left my wallet and travel documents on my dresser.
3. Exception: I get to the airport at 5:30 pm, ready to fly and as I am waiting to check in, the airline announces that all flights are canceled due to extreme weather at the destination.

The error and fault can be corrected my changing my behavior. The error can be avoided in the past by my understanding how to use the 24 hour clock, and the fault can be avoided by having a pre-travel check list to make sure I don't make that same mistake again.

However, no amount of preparation, change or improvement on my part can prevent the flight from being canceled. It is an exceptional situation, and the only thing I can do is to try and handle it. In this case I might consider:

1. Taking a train.
2. Taking a bus.
3. Re-booking on a later flight.
4. Just not show up for the meeting.
5. Canceling my meeting in Montreal and go home to binge watch old episodes of X-Files.

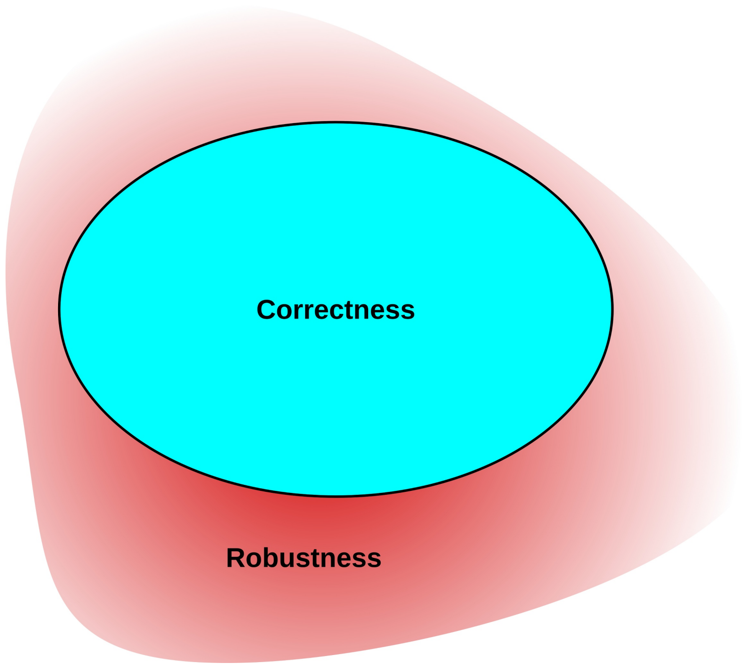
We might say that the last option is a way of exiting gracefully from the exception. While the fourth option is still exiting from the situation, but in a non-graceful way.

This is exactly the same model that we use in most object oriented languages to handle exceptions. Remember that the distinguishing feature of an exception is that no amount of programming can prevent the exception from occurring, we can only manage or handle the exception when it does occur.

Think very carefully about this next statement: not handling an exception is a programming error. Failure to handle exceptions properly affects the robustness of our code. Or as Bertrand Meyer puts it

Robustness is the ability of software systems to react appropriately to abnormal conditions. Robustness complements correctness. Correctness addresses the behavior of a system in cases covered by its specification; robustness characterizes what happens outside of that specification.

One of common problems in program design is that it is rather straightforward to identify the factors that contribute to correctness, since these are clearly defined in the specification, however, to identify the factors that may impact the robustness of the program is more difficult and requires a higher degree of analysis.



In the above diagram we can see the dilemma. Assuming we have a good specification, the boundaries of correctness we need to test are clear, however robustness is looking at rather more fuzzy sort of domain, often dependent on factors like configuration, hardware, operating system and the like.

9.2 General Exception Handling

All OO programming languages have very similar models for handling exceptions although there are differences in how this model is implemented in Java versus C++ or any other OO language. After we look at this general conceptual model, we will look at how Java implements this model.

Regardless of implementation, there are some best design practices that should be followed when implementing exceptions which Meyer calls disciplined exception handing. There are only three acceptable responses to an exception being raised.

1. Ignore: the exception is identified as a false alarm. This means that an exception was signaled in the environment but it has no effect on the success or failure of the program under execution. In this case the exception is ignored.
2. Retry: in this case the program attempts to change the conditions that led to the exception and then restart execution from a specified prior check point.
3. Failure: also knows as organized panic; the program cleans up the execution environment, reports the execution to the calling program, and then terminates execution.

There are two very common errors made in the design of exceptions:

1. Exceptions are thrown for non-exceptional conditions. In this case the exception handling mechanism is being treated as a programming control statement, usually as if it were a form of GO TO statement.
2. Exceptions tend to be ignored or routinely treated as false alarms.

For example, if we are defining a bank account class with a withdraw method, and the amount requested is greater than the balance, a lot or programmers will raise a “Not Sufficient Funds” exception and then terminate the method.

This is bad design because the NSF case should be handled in the specification and is part of the normal logic flow of the application. The usual sorts of reasons why this happens is that either the programmer did not implement the specification correctly, or the specification was incomplete and left this case out.

In this case, the programmer may have written code which initially implemented only the valid cases described in the specification, and then later, when the invalid case was identified, rather than rewriting the code, handled the invalid case as an exception.

The general model is often referred to as try-throw-catch.

First we identify a block of code in which an exception might occur. For example, if we are trying to open a file, then we could get all kinds of exceptions: the file might not exist, we might not have read access to the file or the file might be locked by the operating system. It is the programmer's responsibility in the design of their code to identify the potential

9.2.1 The “Fail-Safe” State

Part of every program design should be a fail-safe state. This is a state that the application goes into when an exception occurs that is characterized by the following properties.:

1. It can be reached from anywhere in the appropriate program environment.
2. If an exception was raised that has a retry option, the fail-safe state ensures we can transition to a resume point with the proper resetting of the environment. In other words we can think of this as making it as if the exception never occurred.
3. If an exception has a failure option, then the fail-safe state performs whatever clean up is necessary to shut the application down cleanly.

9.2.2 The Try-Throw-Catch Model

This is the manner in which exception handling is implemented in OO languages. The basic mechanism in the same in Java, C++ and C# for example, but the details do differ from language to language. The basic mechanism works like this:

```
try {
      code where where and exception could be raised
      throw new TestException();
      code note exectuted ifan exception is raised
}
catch (Exception e) {
      code to execute if an exception is thrown
}
```

First we identify a block of code where an exception may occur. This code is enclosed in what is called a try-block. What the try block does is set up the underlying exception handling mechanism that is used to handle thrown exceptions. The reason we use a try block is that there is a lot of overhead involved in throwing exceptions which has a significant impact on both code size and performance. By restricting our exception handing to a try bock, we reduce the amount of overhead needed overall in the program.

At some point in the try block, we may detect an exception. In this case we create an exception object and “throw” it. The exception object is usually packed with all kinds of information about the exception, including where it was thrown from. Once an exception is thrown, no further code is executed in the try block.

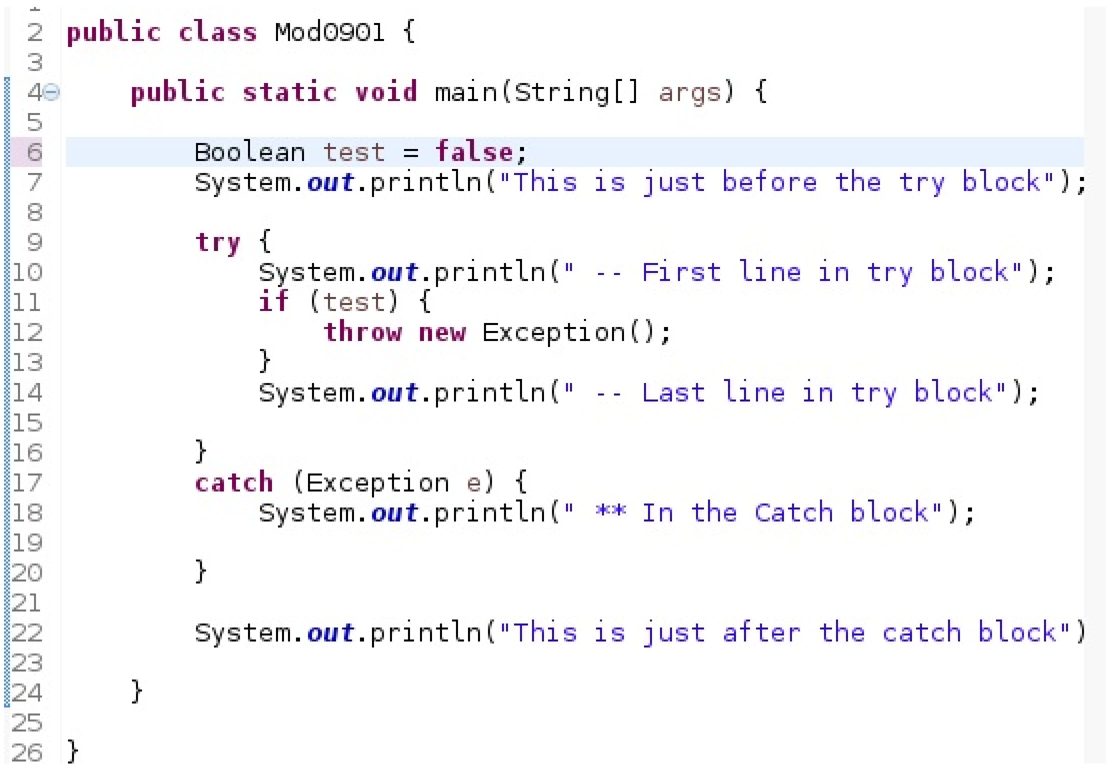
Instead, the try block passes the exception to the exception handler which is the the code contained in the catch block. This code is executed, and then control passes to the first statement after the catch block.

The usual case is that the try block executes without raising an exception (remember, exceptions are exceptional conditions). In this case all the code is executed in the try block and the the flow of control skips over the catch block and resumes with the first statement after the catch block.

9.3 Exceptions in Java

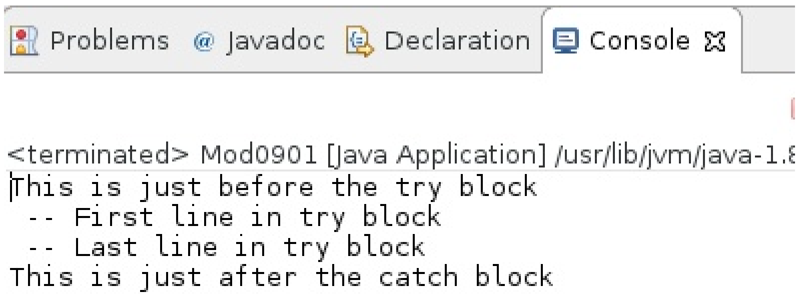
The previous section described exceptions generally. Now we will turn to the way Java implements exceptions. The actual implementation of excepts can get fairly complicated fairly quickly, so we will proceed incrementally.

Consider the following demo code.



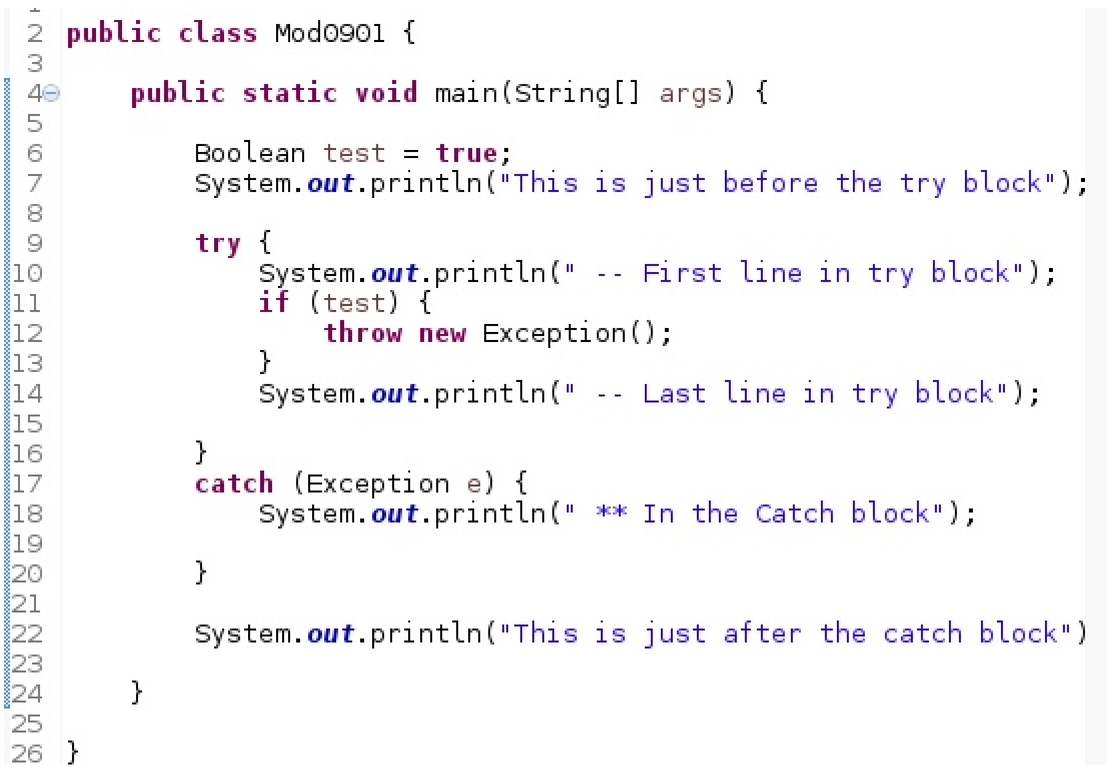
```java
public class Mod0901 {

    public static void main(String[] args) {

        Boolean test = false;
        System.out.println("This is just before the try block");

        try {
            System.out.println(" -- First line in try block");
            if (test) {
                throw new Exception();
            }
            System.out.println(" -- Last line in try block");

        }
        catch (Exception e) {
            System.out.println(" ** In the Catch block");

        }

        System.out.println("This is just after the catch block")

    }

}
```

When executed it produces the following output showing the normal flow of control when no exception is thrown.



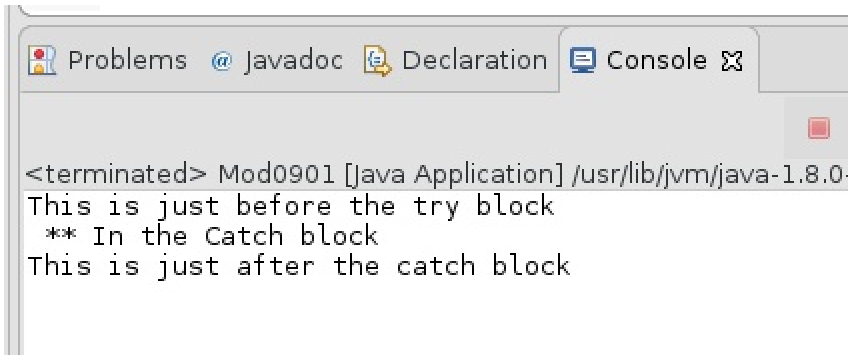
```
<terminated> Mod0901 [Java Application] /usr/lib/jvm/java-1.8
This is just before the try block
 -- First line in try block
 -- Last line in try block
This is just after the catch block
```

Now consider what happens when we throw an exception.



```
public class Mod0901 {

    public static void main(String[] args) {

        Boolean test = true;
        System.out.println("This is just before the try block");

        try {
            System.out.println(" -- First line in try block");
            if (test) {
                throw new Exception();
            }
            System.out.println(" -- Last line in try block");

        }
        catch (Exception e) {
            System.out.println(" ** In the Catch block");

        }

        System.out.println("This is just after the catch block")

    }

}
```

We see the following output. You should step through the code to make sure you understand what hap-pened.



```
<terminated> Mod0901 [Java Application] /usr/lib/jvm/java-1.8.0-
This is just before the try block
 ** In the Catch block
This is just after the catch block
```

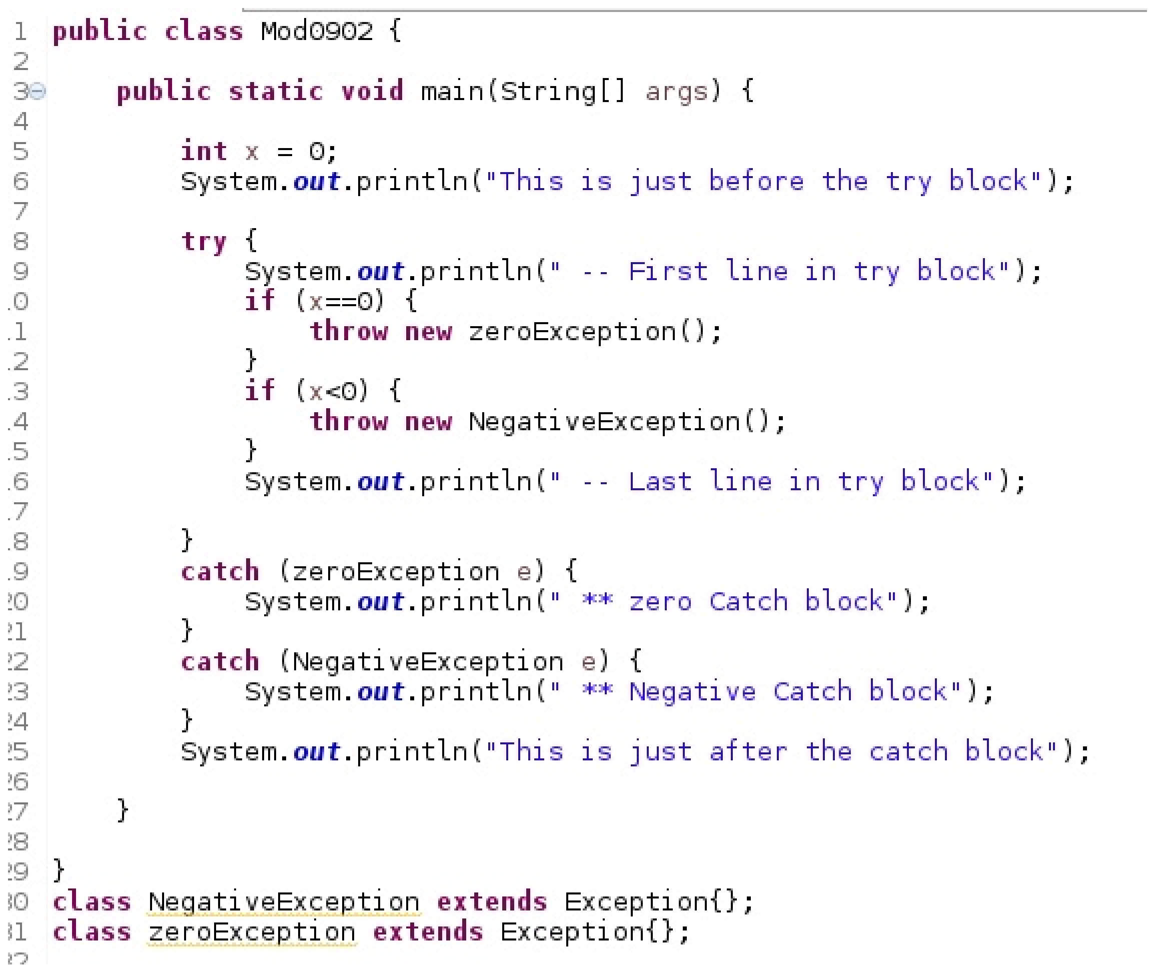
9.3.1 Multiple Exception Types

In the previous example, we used the built in Java class Exception, which is a really bad practice. We will look into types of exceptions a bit later, but for now, we can formulate the following principle.

> *For each type of exception that can be raised, there should be a corresponding Exception Class defined in Java that can be instantiated and throw.*

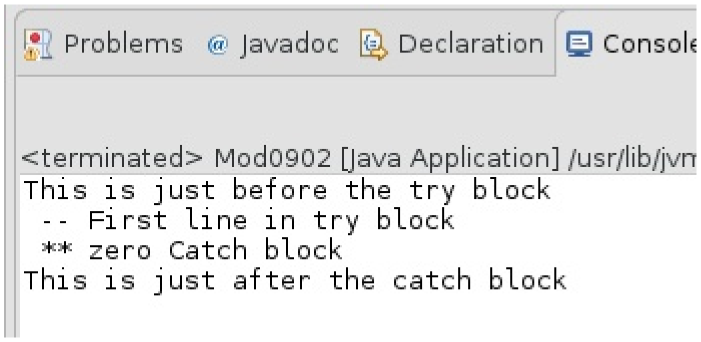
We will go into more detail later, but the usual approach is to sub-class an existing Java exception class to create our own exceptions. This allows us to have multiple exception types handled by a single try block by adding multiple catch block, each of which handles a specific type of exception.

Consider the following code:



```java
public class Mod0902 {

    public static void main(String[] args) {

        int x = 0;
        System.out.println("This is just before the try block");

        try {
            System.out.println(" -- First line in try block");
            if (x==0) {
                throw new zeroException();
            }
            if (x<0) {
                throw new NegativeException();
            }
            System.out.println(" -- Last line in try block");

        }
        catch (zeroException e) {
            System.out.println(" ** zero Catch block");
        }
        catch (NegativeException e) {
            System.out.println(" ** Negative Catch block");
        }
        System.out.println("This is just after the catch block");

    }

}
class NegativeException extends Exception{};
class zeroException extends Exception{};
```

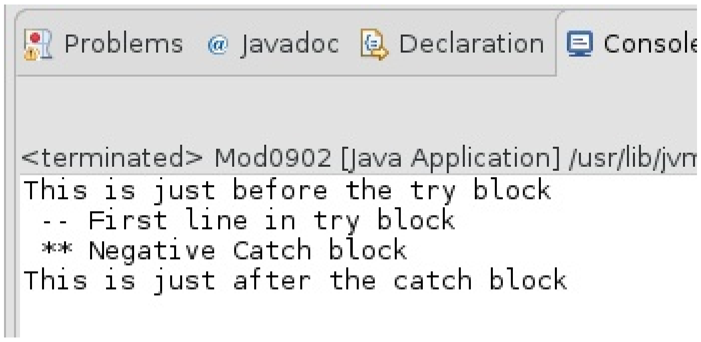
The last two lines just above define two new exception types. We have to subclass them from Exception here because they have to inherit the throwable property or they can not be used as exceptions. We also have two catch blocks. In the first case above, the try block will pass the exception to the first catch to see if that block handles a zeroException, which it does, so we get the following output.



```
<terminated> Mod0902 [Java Application] /usr/lib/jvm
This is just before the try block
 -- First line in try block
 ** zero Catch block
This is just after the catch block
```

And the zeroException handler does in fact execute. If we change the line int x= 0; to int x = -1; and execute, we should see the negativeException catch block execute. The try block will test each catch block in turn until it finds a match.

And that is exactly what happens.



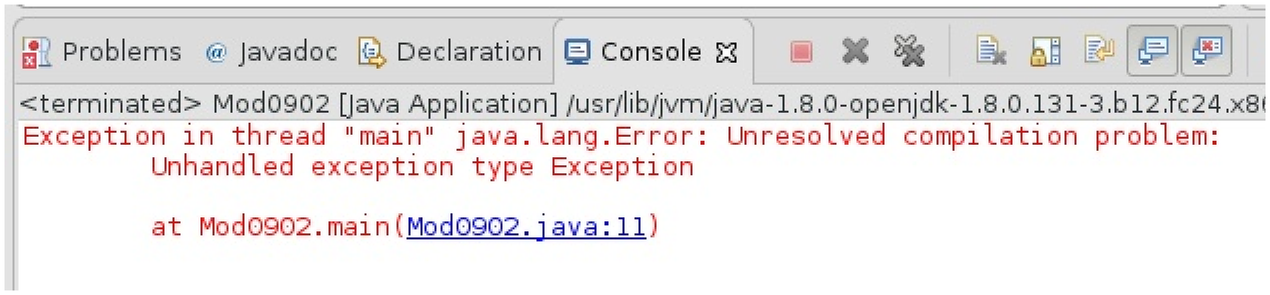
```
<terminated> Mod0902 [Java Application] /usr/lib/jvm
This is just before the try block
 -- First line in try block
 ** Negative Catch block
This is just after the catch block
```

9.3.1.1 No Matching Catch Block

But what if the thrown exception does not match any of the catch blocks? If we just change the type of exception thrown without putting in a catch block to handle it like so:

```
        try {
            System.out.println(" -- First line in try block");
            if (x==0) {
                throw new Exception();
            }
```

We get the following error:

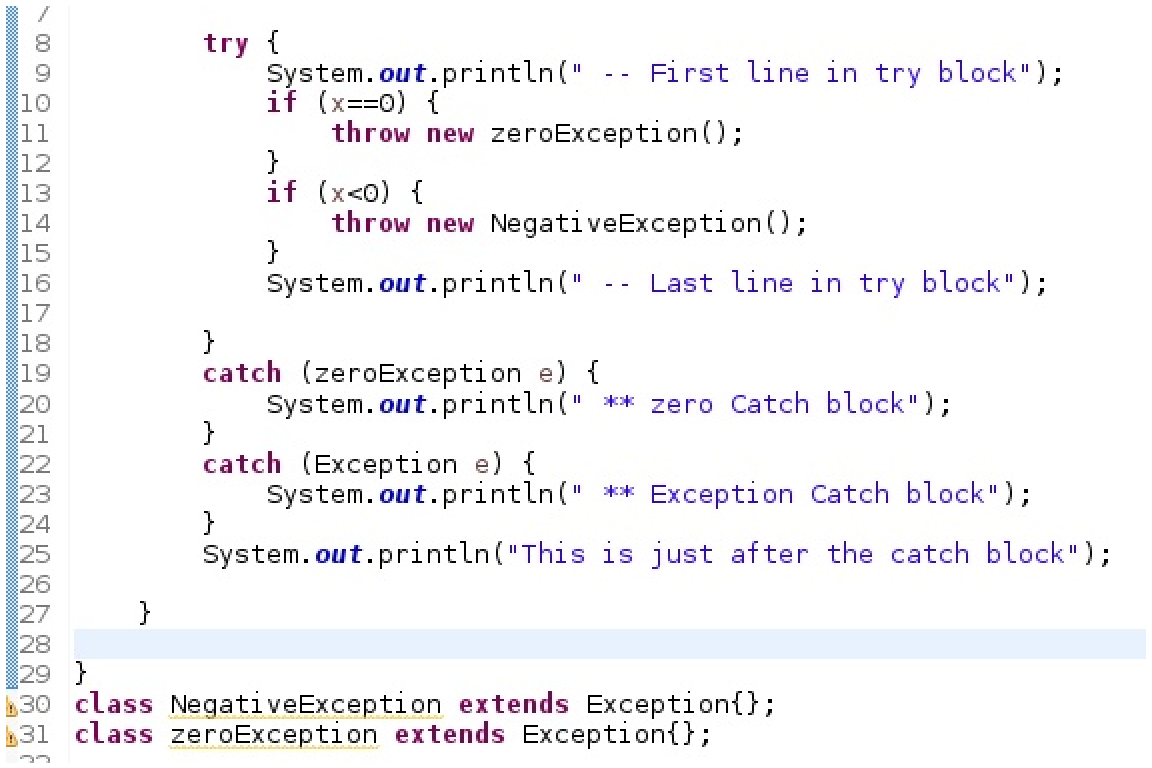


```
<terminated> Mod0902 [Java Application] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.131-3.b12.fc24.x8
Exception in thread "main" java.lang.Error: Unresolved compilation problem: 
	Unhandled exception type Exception

	at Mod0902.main(Mod0902.java:11)
```

We will return to this issue later when we talk about scoping, but for now, we will state that you have to handle all your exceptions somewhere.

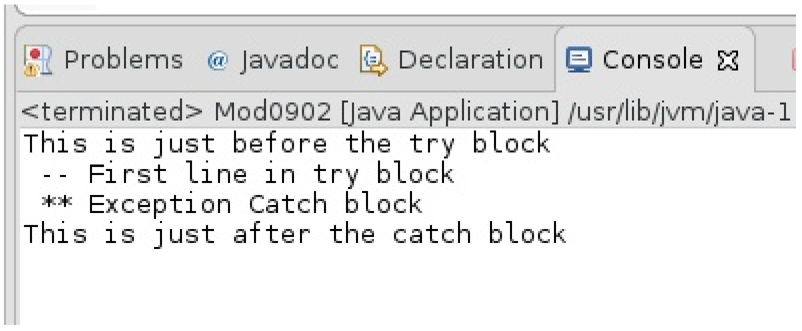
9.3.1.2 Multiple Exception Matches

But what if there are two matches? The first matching catch block is used and any other blocks are ignored. Lets rewrite our code to look like this:



```
        try {
            System.out.println(" -- First line in try block");
            if (x==0) {
                throw new zeroException();
            }
            if (x<0) {
                throw new NegativeException();
            }
            System.out.println(" -- Last line in try block");

        }
        catch (zeroException e) {
            System.out.println(" ** zero Catch block");
        }
        catch (Exception e) {
            System.out.println(" ** Exception Catch block");
        }
        System.out.println("This is just after the catch block");

    }

}
class NegativeException extends Exception{};
class zeroException extends Exception{};
```

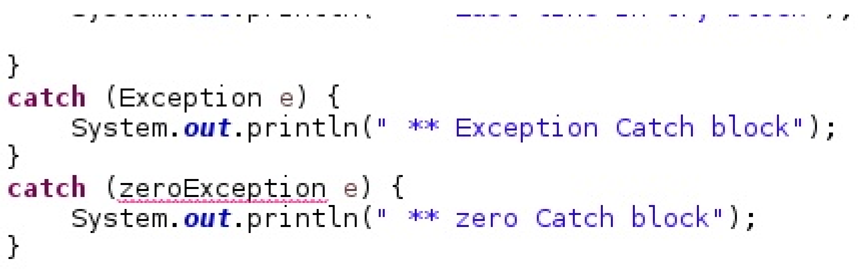
In this case we do not have a negativeException catch block but – remember inheritance – a negativeException is a type of Exception therefore it matches the Exception catch block, and we get this output:



Problems | Javadoc | Declaration | Console

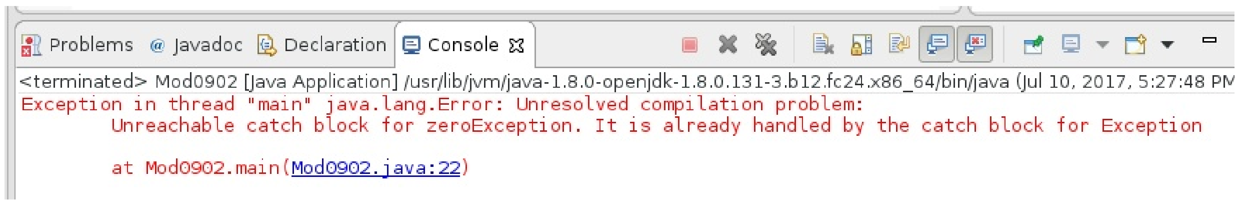
```
<terminated> Mod0902 [Java Application] /usr/lib/jvm/java-1
This is just before the try block
 -- First line in try block
 ** Exception Catch block
This is just after the catch block
```

When we have an exception that can match multiple catch blocks, then we have to remember that it will only be handled by the first catch block it matches. Consider the following code:



```
}
catch (Exception e) {
    System.out.println(" ** Exception Catch block");
}
catch (zeroException e) {
    System.out.println(" ** zero Catch block");
}
```

The problem is that the zeroException block can never execute because every zeroException is an Exception and therefore will be caught the catch block above it. In fact, a smart Java compiler will pick this up as unreachable code:



```
<terminated> Mod0902 [Java Application] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.131-3.b12.fc24.x86_64/bin/java (Jul 10, 2017, 5:27:48 PM
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Unreachable catch block for zeroException. It is already handled by the catch block for Exception

        at Mod0902.main(Mod0902.java:22)
```

Later in this chapter we will explore more complex implementations of exception handling, but it is important that you have a good understanding of this first section before we move on.

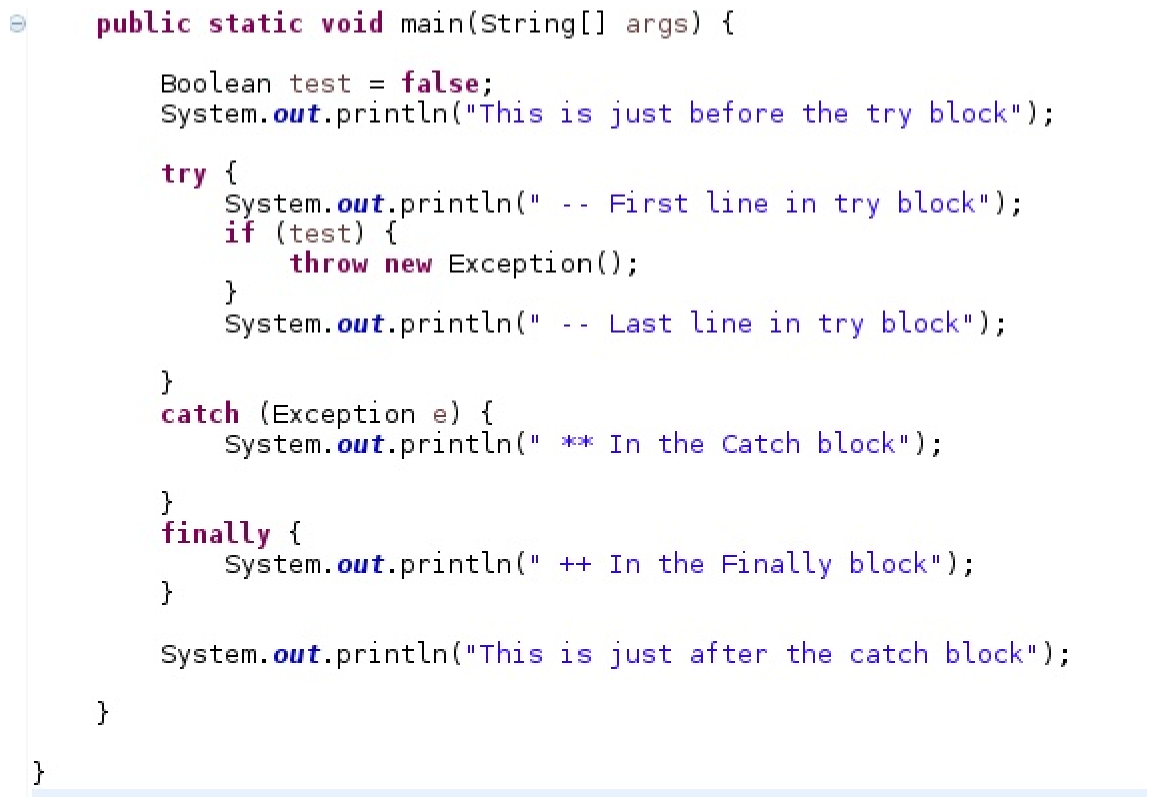
It is also important to note, that our action in all of these exceptions has been to not go into a fail-safe state but just carry on with the code after the try-catch blocks. We will look at alternatives later on in this chapter.

9.3.2 The Finally Block

Consider the following dilemma. At the start of a try block we allocate a resource, lets say a data base connection that will only be used in the try block but we want to free the resource irrespective of whether an exception is thrown or not.

Java allows us to add an extra block of code after the last catch block that will be executed no matter which catch block is executed or if no catch block is executed at all. The purpose of this block to provide a single place to put clean up or exit code instead of having to replicate it in every catch block and at the end of the try block.

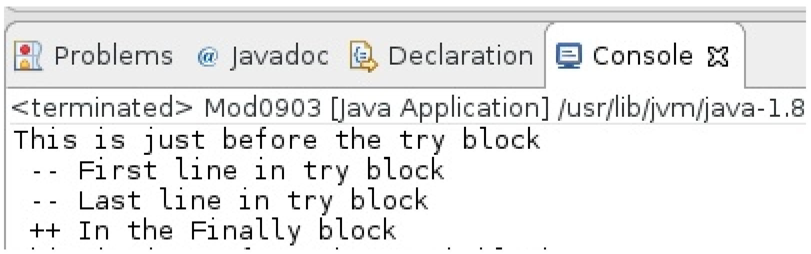
A finally block looks like this:



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

        Boolean test = false;
        System.out.println("This is just before the try block");

        try {
            System.out.println(" -- First line in try block");
            if (test) {
                throw new Exception();
            }
            System.out.println(" -- Last line in try block");

        }
        catch (Exception e) {
            System.out.println(" ** In the Catch block");

        }
        finally {
            System.out.println(" ++ In the Finally block");
        }

        System.out.println("This is just after the catch block");

    }

}
```

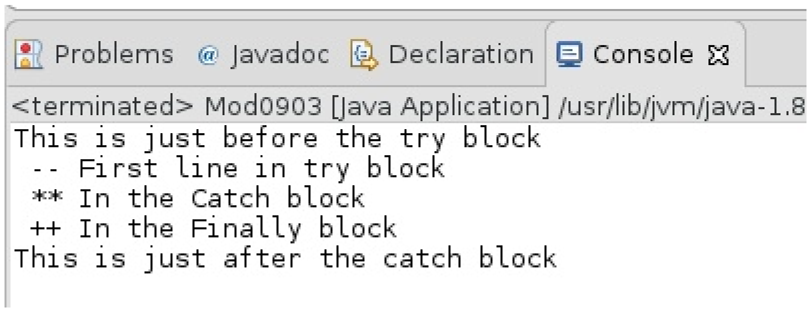
If you run this code both throwing and not throwing an exception, you get the following sets of output.

First without an exception:



```
<terminated> Mod0903 [Java Application] /usr/lib/jvm/java-1.8
This is just before the try block
 -- First line in try block
 -- Last line in try block
 ++ In the Finally block
```

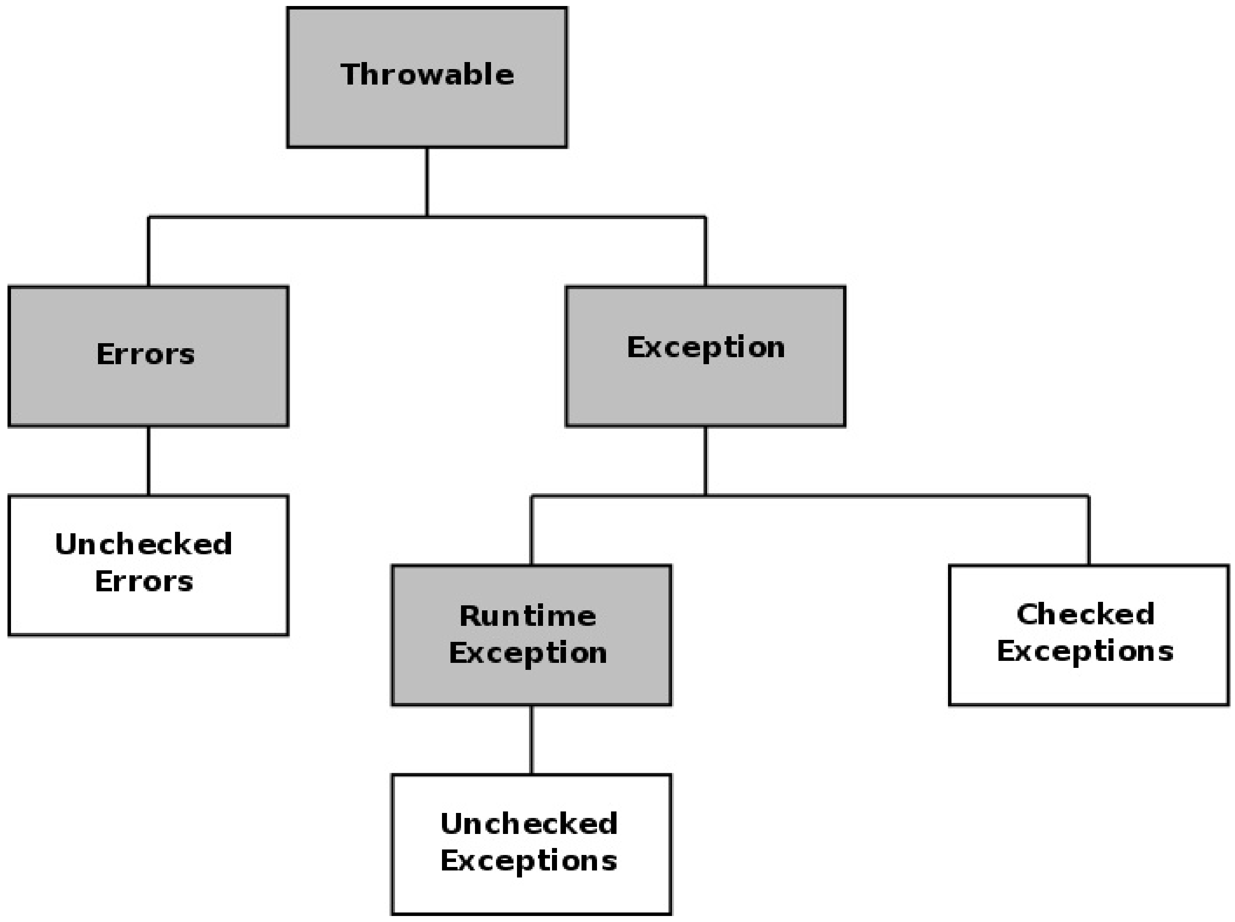
And with an exception:



```
<terminated> Mod0903 [Java Application] /usr/lib/jvm/java-1.8
This is just before the try block
 -- First line in try block
 ** In the Catch block
 ++ In the Finally block
This is just after the catch block
```

9.4 The Java Exception Hierarchy

Java provides a class hierarchy of exceptions.



The throwable class is the base of the hierarchy and only throwable objects can be used in exception handling. (NB: In this discussion remember that "error" and "exception" are used in a Java specific way and do not mean the same as we have been using them in a more general software testing context.)

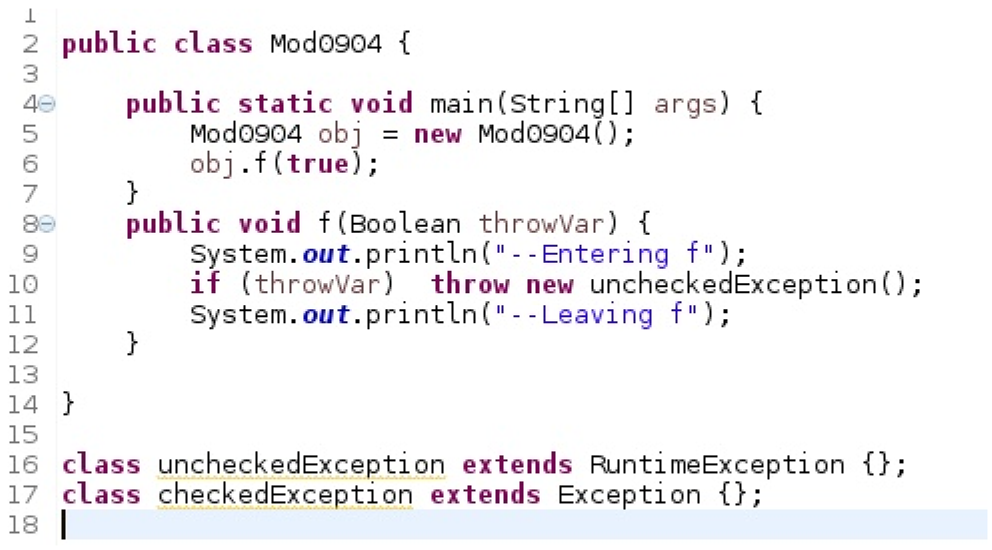
The Error subclass is reserved for abnormal situations that can occur in the JVM execution environment. Errors are rare and should not be caught by applications because there is generally no way to recover from the corruption of the underlying JVM.

The main difference in exceptions is whether they are checked or unchecked. The both work essentially the same way, and there is currently a ongoing theological debate in the Java community as to whether one should use checked or unchecked exceptions. Some Java inspired languages, like C#, have done away with checked exceptions and uses only unchecked exceptions.

A checked exception is one where the compiler checks each method call and method declaration to determine whether the method throws checked exceptions. If it does, the compiler verifies that the checked exception is caught or is declared in a throws clause. In other words, it verifies that you have written code somewhere to handle that exception if it is thrown.

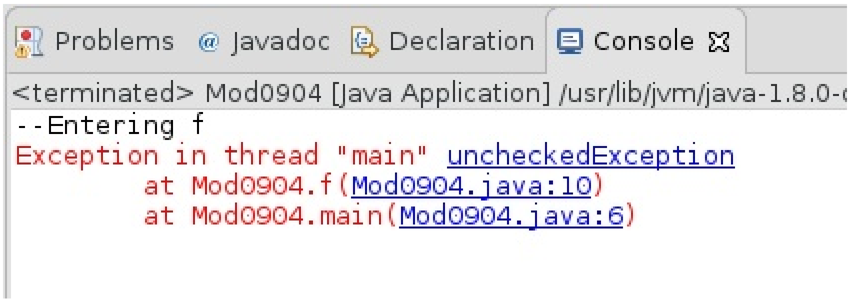
Some programmers are of the opinion that this is necessary, while others claim that using checked exceptions that forces the programmer to focus on robustness in the exception handling design.

For example, considering the following code where we define an unchecked and a checked exception.



```java

public class Mod0904 {

    public static void main(String[] args) {
        Mod0904 obj = new Mod0904();
        obj.f(true);
    }
    public void f(Boolean throwVar) {
        System.out.println("--Entering f");
        if (throwVar)  throw new uncheckedException();
        System.out.println("--Leaving f");
    }

}

class uncheckedException extends RuntimeException {};
class checkedException extends Exception {};

```

In this first version, we are throwing an unchecked exception, and get the following output. Looking at the output, it is clear that our exception was thrown.

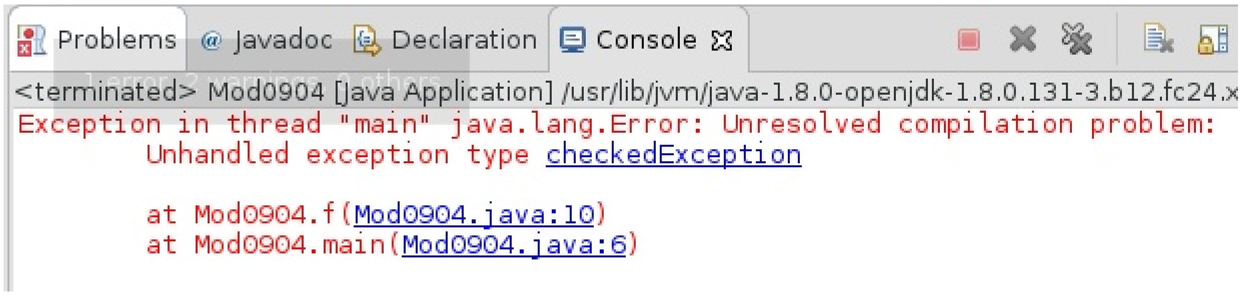


```
<terminated> Mod0904 [Java Application] /usr/lib/jvm/java-1.8.0-
--Entering f
Exception in thread "main" uncheckedException
        at Mod0904.f(Mod0904.java:10)
        at Mod0904.main(Mod0904.java:6)
```

Now changing the type of exception produces the following.

```
    }
    public void f(Boolean throwVar) {
        System.out.println("--Entering f");
        if (throwVar)  throw new checkedException();
        System.out.println("--Leaving f");
    }

```

And we get a compilation error:

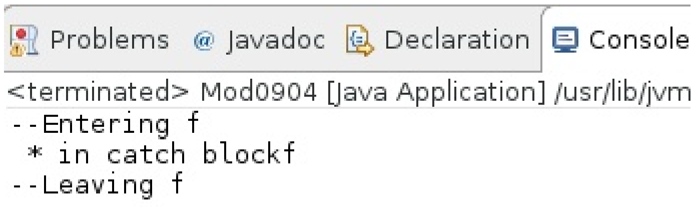


```
<terminated> Mod0904 [Java Application] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.131-3.b12.fc24.x
Exception in thread "main" java.lang.Error: Unresolved compilation problem: 
	Unhandled exception type checkedException

	at Mod0904.f(Mod0904.java:10)
	at Mod0904.main(Mod0904.java:6)
```

In fact we have to make significant code changes to make this work.

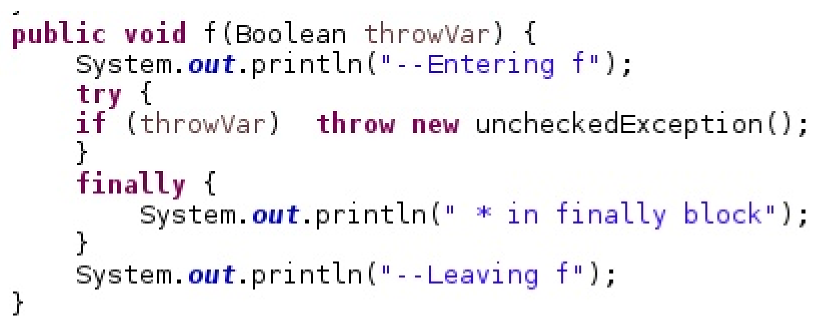
```
    public void f(Boolean throwVar) {
        System.out.println("--Entering f");
        try {
        if (throwVar)  throw new checkedException();
        }
        catch (checkedException e) {
            System.out.println(" * in catch blockf");
        }
        System.out.println("--Leaving f");
    }
```

Which produces the output:



```
<terminated> Mod0904 [Java Application] /usr/lib/jvm
--Entering f
 * in catch blockf
--Leaving f
```

Just to make things a lot more complicated, we can use an unchecked exception in a try block just like a checked exception needs a catch block where as an unchecked exception does not.



```
public void f(Boolean throwVar) {
    System.out.println("--Entering f");
    try {
    if (throwVar)  throw new uncheckedException();
    }
    finally {
        System.out.println(" * in finally block");
    }
    System.out.println("--Leaving f");
}
```

This is legal for an unchecked exception, we just need the finally bock so that the compiler knows where the try-catch construct ends. However, if we used an checked exception, we would get and error telling us there was no corresponding catch block.

9.4.1 The Java Built-In Exceptions

There are hundreds of Java built in exceptions. We will not list them here, but rather suggest you look into the JavaDoc and explore them there. We can use any of the provided exceptions as a super class for our own exception classes.

9.5 Exception Scoping

The way we have been looking at exceptions makes it look like they are lexically scoped, while in fact they are dynamically scoped.

Consider for example the following code.

If we run this with the true changed to a false in h(), we can see the call stack which works as we expect.

But if we run it with the unchecked exception being thrown, we see that even though the exception is through in method h(), it is caught in method f();

What happens is when an exception is thrown, it first looks for a handler in the current method, in our case h(). If there is not one in h(), it unwinds the stack and looks in the function that called h(), in this case g(). Since it can't find one in g(), it unwinds the stack again and looks in the function that called g() which is f() where it finally finds a handler.

Remember that we are throwing an unchecked exception. But what it we change that to a checked exception? In this case we get and error because the compiler cannot tell at compile time if the throw statement is in the dynamic scope of the catch statement – it can only look at the lexical scope and it is not able to figure that out.

For checked exceptions, we have to help out the compiler by leaving essentially a trail of bread crumbs so that it can figure out, more or less, what the dynamic scope of a try block is.

In the following we can see that by adding a notation on the method signature, we can tell the compiler what sort of exception is throw by this method or by any method it calls. Again we don't have to this for unchecked exceptions because the compiler does not have to do any checking at compile time.

9.5.1 Un-handled Exceptions

When using unchecked expressions, it is quite possible to throw an exception that never gets caught When this happens, the Java JVM issues an unhandled exception exception and immediately exits.

To throw unhandled exceptions is considered to be very poor programming style.

9.5.2 Nested Try Blocks

The case we have looked at so far is the simplest case, there is only one try block. Now lets complicate things by adding a try block in the body of function g() in addition to the one in the body of function f().

When function h() throws the checkedException, the stack unwinds to find the first try block. If that try block has an catch block to process the exception, then it is handled there.

This is illustrated by the code and output following.

Which produces the output:

But now, lets change it so that the catch block in g() no longer catches our checkedException.

And we leave everything else exactly as it is. Then we get the following output.

Compare the two outputs and the two different sets of code and make sure you can trace the flow of control in each. This example only uses two nested try blocks but there is no practical limit on how many try blocks can be nested. However, more than a handful of nested try blocks may be a symptom of poor code structure or design.

9.5.3 Rethrowing Exceptions

In the previous example, we could have their the try block in g() or the try block in f() handle the exception. But what if we wanted both? We wanted the catch block in g() to some local fixups but then wanted to have the block in f() also do its fixups?

This actually is a fairly common situation where an exception is raised in a module, and some local processing is done that is module dependent, but the module try block fails in its attempts to fix things up.

At this point we want the module to escalate the exception to the next higher level of control basically saying “I can't handle this, you decide what to do.”