# *Head First Java: Chapter 11 Notes*

*September 3, 2019*

## *Throwing and Catching Exceptions*

In Java, exceptions handling involves declaring (*throwing*) anticipated possible exceptions in methods, and acknowledging plausible exceptions in method calls (*catching*). Practically all properly developed Java API's classes and methods throws exceptions, and they can be seen in their documentation online. For example, the documentation for `javax.sound.midi.Sequencer.getSequencer()` throws the `MidiUnavailableException`. Because this particular exception was *thrown* in the method declaration and description, when calling it elsewhere it must be caught via the **`try...catch...finally`** block. For example, in order to use the `getSequencer()` method described just now:

```
import javax.sound.midi.*;

public class MusicTest1 {
    public void play() {
        try {
            Sequencer sequencer = MidiSystem.getSequencer();
            System.out.println(''Successfully got a sequencer.'');
        } catch(MidiUnavailableExeption ex) {
            System.out.println(''Fail to get sequencer.'');
            ex.printStackTrace();
        }
    }
}
```

This is similar to Python try/except blacks except in Java it is *mandatory*, if a method throws a (or more than one) exception(s), code that calls that method **must** use the try/catch block to anticipate it/them. In the above example, the `play()` method will run the `Sequencer sequencer = MidiSystem.getSequencer();` line of code, if it success, it will run the remaining code in the `try` block. If a `MidiUnavailableExeption` occur, the compiler will store the error in the exception object `ex`, and skip the remaining code in the `try` block, and run the code in the `catch` block instead. Code in the `catch` block is not ran unless the declared exception occurred. Because of this, the code within the `catch` block is usually to the last chance to recover from the exception to ensure the program will continue to run, otherwise this is an opportunity to regularly exit from the program, rather than crashing. Using the `printStackTrace()` or other diagnostic method, the user can then proceed to investigate what went wrong.

To throw exceptions in a method, it must be declared first. For example:

```
public void takeRisk() throws BadException {
    if (exceptionOccurs) {
    throw new BadException();
    }
}

public void tryingRisks() {
    try {
        anObject.takeRisk();
    } catch(BadException ex) {
        System.out.println(‘‘Bad exception occurred.’’)
    }
}
```

Note that in the above example, `BadException` is not a real and defined `Exception` subclass object, but user-defined exceptions can be defined just like any other objects. The reasoning of the above code listing is that, while writing `takeRisk()` as a method, the user anticipated a specific exception `BadException`, which must be declared at the method declaration with keyword `throws`. Multiple exceptions can be declared here, separated by commas. Then, within the body of the method code, the exception can be thrown by using the `throw` keyword by initializing a object of the exception (also using the `new` keyword.) Once the exception is declared and the initialization of the exception is properly coded in the method, whenever the method is called, the exception must be caught via a try/caught block. If multiple exceptions were thrown, they must all be caught.

## *Exception Categories and Hierarchy*

Exceptions are handled like objects in Java. All declarable exceptions are a subclass of class `Exception` (which like all objects is a subclass of `Object`). All subclasses of `Exception` are checked during compilation by the Java compiler, except for the subclass `RuntimeException` and its own subclasses. By definition, *runtime exceptions* occur only at runtime, and usually due to either faulty code logic or user input. Runtime exceptions should not be caught by try/catch blocks but rather should be avoided through thorough testing and proper coding.

## *Flow Control in `Try/Catch` Blocks*

As mentioned above, code in `try` and `catch` blocks are in an either/or situation. Consider the following code listing:

```
try {
    Foo f = x.doRiskyThing();
    int b = f.getNum();
} catch (SomeException ex) {
    System.out.println(''Failed.'')
} finally {
    System.out.println(''Exiting...'')
}
System.exit(0);
```

There are two lines of code within the `try` block, how many of them run depends on if and when `SomeException` was thrown. If no exception was thrown, all code within the `try` block will run, then all code within the `catch` block will be skipped. If `SomeException` does get thrown, one or two lines of the code may be ran within the `try` block, depending when the exception was thrown. Regardless, once the exception was thrown, all code within the `try` block is skipped, and the code within the `catch` block is ran. The `finally` block is optional, but it is always ran regardless when all possible flow within the try/catch block. Code that exist outside of the try/catch(/finally) block only runs when all code within the block(s) are ran according the the flow described above (note that it is possible for the flow of the code within the block to lead elsewhere other than the next line if code outside of the block, *e.g.* via `return` or exiting the program within the `catch` block; if that occurs, the code outside the try/catch block might not run, but code within the `finally` block *will* run regardless.)

## *Exceptions and Polymorphism*

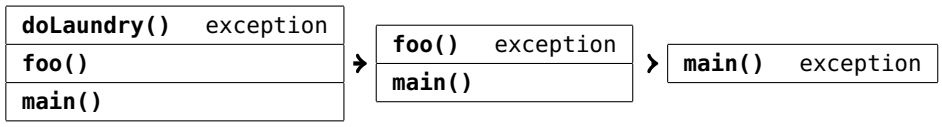
Because exceptions in Java are objects, all polymorphism concepts and behavior applies. If an exception is thrown, it can be caught by a superclass of the exception, but not the other way around. Because all exceptions are subclasses of `Exception`, theoretically `Exception` can catch all possible exceptions, but this is bad coding practices as it will lead to ambiguity on what exact exception has occur, and hence recovery would be difficult. If multiple exceptions are thrown, and they are of from the same inheritance tree but of a different hierarchal level, when coding the multiple `catch` blocks, they must be in the order from bottom subclass towards the superclass. The reason is if they are out of order, the exception that exist higher on the inheritance tree will catch all thrown exceptions that includes itself and its subclass, which leaves the `catch` block code intended to catch exceptions at a lower inheritance position meaningless.

## *Exception Ducking*

When exceptions are thrown, they have to be caught eventually. However, it is possible to pass exceptions along a chain of method calls, which catches the exception at this end without intervention in the intermittent method calls. Consider:

```
public class Washer {
    Laundry laundry = new Laundry();
    public void foo() throws ClothingException {
        laundry.doLaundry();
    }
    public static void main(String[] args) {
        Washer a = new Washer();
        a.foo();
    }
}
```

The above code illustrate a case where the code will compile but at runtime it will run into problems. Assuming that `doLaundry()` and class `Laundry` is properly defined and that the method throws a `ClothingException`, it would appear that there isn't any code written to catch the thrown method due to all the method ducking the thrown exception. The above (assume that the classes `Laundry`, `Washer`, and the exception `ClothingException` are properly defined elsewhere), when ran, can be visualized on the stack as such:




This the stack visualization above, `main()` first appear on the stack, which (aside from initiating a `Washer` class object and assigning it to reference variable `a`) runs the method `foo()`, whose frame is now in scope above the `main()` frame. Then, `foo()` calls the method `doLaundry()`, whose frame is now at the top of the stack. As mentioned earlier, `doLaundry()` initiates the `ClothingException` object and throws it (not shown above). As `doLaundry()` resolves and is popped of the stack, `ClothingException` is thrown to the method that called it, in this case `foo()`, whose frame is now in scope. However, as shown above, `foo()` has no try/catch code block to catch `ClothingException`, rather, it declares that it throws `ClothingException` without actually including code to throw a new instance of the exception. In this case, the `ClothingException` is "*ducked*", which means it avoid catching the exception and expects another method call below it on the stack to catch it. The code above

was deliberately written to cause a runtime exception, because as `foo()` is popped off the stack, the exception is thrown further down to `main()`, which like `foo()`, also throws `ClothingException` and has no try/catch block to catch the exception. This is cause runtime exception that shuts down the JVM at run time. Therefore, as many ducks can be used to pass exception down the stack, but it must eventually be caught before it reaches `main()` to avoid runtime errors.