Lesson 34: Exceptions

What is an exception?

An exception is simply an error. Instead of saying an error occurred, we say that an **exception is thrown**.

How Java handles exceptions:

Suppose we have a "chain" of methods. The *main* method is the first in the chain. An expression in it calls some other method, and an expression in it calls yet another. Let's assume some piece of code deep in this chain throws an exception. The code immediately surrounding the offending code is examined for a *try-catch* statement. (More on this later; however, for now, suffice it to say that if a *try-catch* is present, it handles the error gracefully without the entire program coming to a halt). If no *try-catch* is found, control immediately passes up the chain to the code that called the method in which the error just occurred. Again, a *try-catch* statement is sought, and if none is found, control is passed back up the calling chain one level. This continues until the *main* method is reached. If it has no *try-catch*, the program is halted with a trace of the method calls, the type of exception, and its error message.

Actually, the above is oversimplified just a bit. In the absence of a *try-catch* statement, control is passed up the calling chain one level **only when we specify it to happen**. This is done with a **throws** specifier in the method signature. (More on **throws** on the next pg)

Forcing an error:

You can **force Java to appear to give an error** as in the following example from a *BankAccount* object. In this example we look at **preconditions** that we require before this method is called. The method we will discuss is *withdraw( )*. Certainly, we can't withdraw more than what we have in the account. Also, it would be meaningless to withdraw a negative or zero amount, so we detail these preconditions in the rems above the method signature.

```
/* precondition: amount <= balance
* precondition: amount > 0 */
public void withdraw(double amount)
{
        if (amount > balance)
        {
                String s = "Can't withdraw more than the balance.";
                IllegalArgumentException e = new IllegalArgumentException(s);
                throw e; //Presents the s message and the entire program stops
        }

        if (amount <= 0)
        {
                String s = "Withdrawal amount must be greater than 0.";
                IllegalArgumentException e = new IllegalArgumentException(s);
                throw e; //Presents the s message and the entire program stops
            }
            . . . remainder of code for this method. . .
        }
```

In the above code we used the exception class, *IllegalArgumentException*. See Appendix K for a list of some other exception classes.

Two types of exceptions:

Checked: those that Java requires handling by the programmer. These are typically errors over which the programmer has no control. The IOException is a classic example.

> **Mnemonic memory aide:** For errors that are out of your control, such as an *IOException* due to a corrupt file, etc., Java is **extra vigilant** and **checks** for errors for you.

Unchecked; those that the programmer may or may not handle. These are errors that are the programmer's fault. A typical example would be a division by 0 giving an *ArithmeticException*. Another example would be a *NumberFormatException* that might occur when you try to do *Integer.parseInt(m)* and the *String m* can't convert to an *int*. (The example code on the previous page concerned unchecked exceptions.)

Two choices for handling checked exceptions:

Now let's suppose we have a method that has the **potential** of throwing a **checked** exception. We **must handle** the exception with one of two choices. Notice that with checked exceptions, doing nothing is **not** a choice, it won't even compile unless you do one of the following:

1. Handle the exception with ***try, catch, finally*** as we will see later in this lesson.
2. Put a *throws IOException* (or some other appropriate checked exception) tag on the method signature as in the following example:

```
public void readTheDisk( ) throws IOException
{
        … code that uses a file reader…might encounter a corrupt file…
}
```

So, what's the purpose of this ***throws*** specifier in a method? The **purpose** is so the calling-method (the method that called the *readTheDisk* method) is signaled that an *IOException* **may** occur in *readTheDisk* and that the calling method is to handle the exception. Of course, in the calling-method you can make the choice of putting another ***throws*** specifier in **its** signature, or to actually handle the exception right there with *try, catch,* and *finally*.

Thus, we see that the ***throws*** specifier is a way to **defer** the handling of an exception. We can keep postponing the actual handling of the exception right up the calling chain. Of course we can defer it all the way up to the *main* method and if no *try-catch* is there, then the program terminates.

The ***throws*** specifier can provide for **multiple** exceptions as in the following example:

```
public void aMethod(int x) throws IOException, ClassNotFoundException
{
}
```

We should also mention that unchecked exceptions can also make use of ***throws*** to defer handling of the exception…or you can handle them at any level in the calling chain with ***try, catch,*** and ***finally***. Unchecked exceptions need not be handled at all. You can just let the program terminate (crash) immediately upon detection of such an error.

Catching exceptions…(referred to above as "handling" the exception):

```
public class MyClass
{
        public void myMethod(double d)
        {
                …some code in which you are not
                worried about an exception occurring….

                try //if error occurs, rest of code doesn't execute...jumps to
                //appropriate catch
                {
                        …some code where you might expect an exception…
                        String s = in.nextLine( );      //might produce an IOException
                        int x = Integer.parseInt(s);    //bad s might produce a
                                                        // NumberFormatException
                        …more code …
                }
                catch (IOException e)
                {
                        System.out.println("Input/output error " + e);
                        // Continues execution after last catch below.
                }
                catch (NumberFormatException e)
                {
                        System.out.println("Input was not a num " + e);
                        // Continues execution immediately after this catch.
                }
                …code execution continues here after try block is finished…or, if an
                exception occurs, execution continues here after catch block finishes…
        }
}
```

Usage of the *finally* block: (this block is optional)

```
try
{ … }

catch( … )
{ … }

finally
{
```

…This block of code is **guaranteed** to execute, regardless of whether there was an exception or not. …This is **typically used to release resources**, such as closing a file or releasing a network connection. If an exception occurs in the *try* block and none of the *catch* statements are appropriate to handle the exception, control passes to this *finally* block and the code here executes… then the exception is actually thrown and passed up the calling chain where we can attempt to *catch* it.

Even if a *catch* block throws an exception of its own, control is still passed to the *finally* block.

```
}
```

Designing your own Exception Types:

```
if (amount > balance)
{
        GoofyException e = new GoofyException( "You made a dumb mistake!");
        throw e;
}

public class GoofyException extends RuntimeException
{
        public GoofyException( ) // It is customary to provide a default constructor,
        { // even if empty.
        }

        public GoofyException(String reason) //Your own constructor
        {
        super(reason);
        }
}
```

Some unusual facts concerning exceptions:

1. If you use the following two *catch* statements, they must go in the order shown…subclass on the top, superclass on the bottom. (Note that *FileNotFound* is a subclass of *IOException*)

   ```
   catch(FileNotFoundException e)
   {
           System.out.println("FileNotFound");
   }
   catch(IOException e)
   {
           System.out.println("IOException");
   }
   ```

   In this particular case, if the *try* block generated a *FileNotFoundException*, only "FileNotFound" would be printed. The second *catch* would be ignored.

2. If you have a method that throws an *IOException* up to the next level in the calling chain, we should be aware that in addition to *IOException* being thrown to the next level, all its subclasses are also thrown to the next level in the chain.

3. If you put a *try* inside a block of code (such as those belonging to loops and *if* statements), then the corresponding *catch* statements must also reside in that same block.

4. It is permissible to have *try* and *finally* blocks without a *catch* block.

5. If there is a *finally* block, there must be at least a *try* block.