Exceptions

If there is a problem with a Java statement, then a Java program may generate an exception or an error. Although an error represents a situation that the program can not recover from, an exception can be handled.

Here are your three options when dealing with a thrown exception:

1) Not handle the exception
2) Handle the exception where it occurs
3) Handle the exception at another point in the program.

Option #1
If you choose not to handle an exception, your program will terminate abnormally and produce an error message telling you which statement caused the exception to be thrown. This message is very helpful for debugging purposes. It will print out a call stack trace that tells you exactly which line in which method caused the exception. This printout includes the entire set of method calls leading to the exception. (e.g. main called func1 which called parseInt, which threw an exception.)

Option #2
In order to handle an exception, we must use the try statement. Here is the syntax of the try statement:

```
try {
        <stmts>
}
catch (<Class> <var>) {
        <exp_stmts>
}
```

You may have more than one catch clause. (You can have 0 or more.) Also, you may have a finally clause, but this must go at the end:

```
try {
        <stmts>
}
catch (<Class> <var>) {
        <exp_stmts>
}
finally {
        <fin_stmts>
}
```

If the statements inside of the try block execute normally, then execution skips to the finally block after finishing the try block.
(If no finally block exists, then execution skips to right after the last catch block.)

However, if one of the statements in the try block throws an exception, then the rest of the statements in the try block are NOT executed. Rather, the class of the exception is noted. If this matches any of the catch clauses, then execution is transferred to the appropriate catch block. After that code is executed, execution transfers to directly after the last catch block.

The whole benefit of catching an exception is that you allow your program to continue even if a line of code has thrown an exception.

Example

```
boolean done = false;
int x;
while (!done) {

        try {
                System.out.println("Enter an integer.");
                x = Integer.parseInt(stdin.readLine());
                done = true;
        }
        catch (NumberFormatException exception) {
                System.out.println("Sorry, try entering an int!");
        }
}
```

This piece of code will continue to prompt the user for an integer until they enter a valid one. Notice that without the try and catch in the while loop, this program would terminate after the first time data of the wrong format was entered.

Notice that a block of code may have the possibility of throwing more than one exception. This is why you may have mutliple catches after a try block.

Finally clause

Execution is ALWAYS transferred to the finally clause whether an exception was thrown or not. It is executed last.

An example of when one would like to use a finally clause is when regardless of whether an exception occurs or not, you definitively want to complete an action, such as closing a file. (Look on page 562 in the text for an example of this.)

Exception Propagation

If an exception is not caught immediately in the method that it is thrown, the method will propagate or throw the exception to its calling method. It is possible them for this calling method to handle the exception.

Consider the following situation:

1) Main calls method level1.
2) level1 calls method level2 inside of a try block.
3) level2 calls method level3.
4) A statement in level3 throws an exception that is not caught.
5) The exception is propagated back to level2.
6) Since level2 doesn't handle the exception, it's propagated to level1
7) level1 handles the exception by printing out some pertinent information about the exception.

In the example posted (CircleExcptEx.java) Main has a try block in it, and inside of that try block there is a call to a method getInput(). getInput throws an exception, which is then caught by the try/catch block in main. Here the exception just propagated from getInput to main. However, the chain can go further. We could easily make getInput call another method, and have this method throw the exception to getInput, and then have getInput throw the exception back to main.

One way to think about this is that you have a person at work run into a problem (say a computer problem). He doesn't want to deal with it, so he throws it to his supervisor. He doesn't want to deal with it either, so he throws it to his supervisor. Then, she decides to fix (catch) the problem at her level.

Checked vs. Unchecked Exceptions

A compiler can determine whether or not there is a possibility of a checked exception occurring. An example of a checked exception is a FileNotFoundException. At compile time, we can look at code and determine whether or not this exception is a possibility.

For all checked exceptions, Java requires you to either catch them, or to report them with a throws clause. The throw clause must be included at the beginning of any method that has the possibility of throwing a checked Exception. (We typically did this with main for I/O purposes.)

You might have noticed that you never bother throwing ArithmeticException, for example. That is because ArithmeticException is an unchecked exception. It is one that the compiler can't determine whether or not it might occur.

An unchecked one is not required to be thrown or caught. It may still be in the best interest of the programmer to try to catch these and track them down, but the compiler does not require it.

Method printStackTrace()

This method is defined in the Throwable class, from which Exception inherits. Thus, anytime you have an Exception object, you can call this method on that object. It will print out the stack of calls in the chain that created originally threw the exception. These are the SAME EXACT messages that the Java compiler prints when a regular run-time error (which occurs by an exception being thrown) happens.

Creating your Own Exceptions

Due to Java's framework, creating a basic exception is very, very easy. In order to do it, you need to create a class that inherits from Exception. From there, all that is required is a constructor (either a default or one that takes in a String).

Here's a quick example that we'll use with the AddressBook. The AddressBook is a class that maintains a list of several contacts. It's possible that it might get filled, so that if someone tries to add a Contact to a full AddressBook, then they should get a FullAddressBookException:

```
public class FullAddressBookException extends Exception {

  public FullAddressBookException(String s) {
    super(s);
  }
}
```

Once we've defined this class, then we're free to either throw or catch this exception as necessary.

Clearly, it could only be triggered by the method addContact.

Let's take a look at two versions of the AddressBook program: one that just ends up throwing this exception and another that handles it gracefully.

The example that I've added to our class examples is the Divide by Zero Exception:

```
public class DivideByZeroException extends Exception {
     public DivideByZeroException(String s) {
          super(s);
          System.out.println("Please do not divide by zero!");
     }
}
```

Once we create this exception, we can throw it from any class we make. Here is an example from the fraction class:

```
  public fraction divide(fraction f)  throws DivideByZeroException {
     if (this.undefined() || f.undefined()) return new fraction(1, 0);

     int num = this.numerator*f.denominator;
     int den = this.denominator*f.numerator;

     if (den == 0) throw new DivideByZeroException("");
     return new fraction(num, den);
  }
```

Then, in subsequent code, we can either try to catch the exception or not. Here is an example where we gracefully catch the exception:

```
public class TestExceptionCatching {

	public static void main(String[] args) {

		// Read in info about two fractions.
		Scanner stdin = new Scanner(System.in);
		System.out.println("Enter the num and den for the first frac.");
		int x = stdin.nextInt();
		int y = stdin.nextInt();
		fraction first = new fraction(x, y);
		System.out.println("Enter the num and den for the second frac.");
		x = stdin.nextInt();
		y = stdin.nextInt();
		fraction second = new fraction(x, y);
		fraction result = null;

		// Here is how we try our division and try to catch the exception.
		try {
			result = first.divide(second);
		}
		catch (DivideByZeroException exception) {
			System.out.println("Division isn't possible.");
		}

		// Only print if the reference isn't null.
		if (result != null)
			System.out.println("The result is "+result);
	}
}
```

Alternatively, we could let the exception get thrown:

```
public class TestExceptionCatching {

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

		// Read in info about two fractions.
		Scanner stdin = new Scanner(System.in);
		System.out.println("Enter the num and den for the first frac.");
		int x = stdin.nextInt();
		int y = stdin.nextInt();
		fraction first = new fraction(x, y);
		System.out.println("Enter the num and den for the second frac.");
		x = stdin.nextInt();
		y = stdin.nextInt();
		fraction second = new fraction(x, y);
		fraction result = null;

		result = first.divide(second);
		System.out.println("The result of your calculation is "+result);
	}
}
```

Syntax-wise, notice that if we catch all situations where the exception could happen, we don't need to report that it could get thrown. But, if it could get thrown from some method, because that method makes a call to another one that could throw the exception, and we don't catch it, then we must ourselves, report that the exception could be thrown.