CS151 Lab#1: Exceptions & I/O

Starting from this lab, you need to have a TA check off on all your exercises.

In this lab, we will learn about exception handling in Java. We will also learn how to read and write data from files using some of the simpler mechanisms provided in Java. Look in `~dxu/handouts/cs151/labs/01` for code and data files related to this lab.

Understanding Errors: Syntax & Runtime Errors

While editing code, you will invariably have typos or misspellings that will lead to syntax errors. You will notice that code will not compile until all syntax errors are corrected. Tune yourself to the Java syntax so that whenever you write a Java program (whether on paper or in Eclipse) you are always writing correct syntax. This will help improve your understanding of Java as well as save you time during implementation.

Once you have a complete program free of syntax errors, it is time to try and run it. Programs rarely run correctly the first time. There are always bugs. Your program may successfully run to completion, but it may produce incorrect results. This is may be due to faulty logic in your code. You will have to examine the program or its output and fix it. Later, we will learn ways to test your programs to minimize these errors.

Often, during program execution, you will run into errors due to which your program will crash! This happens very often during development and it is important to learn to recognize the cause of errors so that you can fix them. Runtime errors that result in program crashes typically print out the cause of the error, including the line on which the error occurred, in the Java Console. These may appear cryptic at first, but it is time to try and understand them.

Writing Code That Crashes!

The best way to understand runtime errors is to write code that crashes!

Exercise 1: Create the following program (or copy from handouts):

```
public class Crash1 {

	static int[] a = { 10, 20, 30, 40, 50 };

	public static void main(String[] args) {

		for (int i = 0; i < a.length; i++) {
			System.out.println(a[i]);
		}

		System.out.println("Done printing the array!");

	}
}
```

Run the program. This is the correct version. You will get the output shown below.

```
10
20
30
40
50
Done printing the array!
```

Now, let's break it. Ready?

Change the termination test in the for-loop to the following:

```
i <= a.length
```

Can you tell what's going to happen? Why?

Run the program. You should now see the output shown below:

```
10
20
30
40
50
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
	at Crash1.main(Crash1.java:8)
```

You got a runtime error in your program. After printing out the last element in the array, it got the error. Why? Did you guess correctly?

Let's try and parse the error message. Runtime errors are typically called *exceptions* in Java. If you read the first line, it is telling you that there was an exception in the thread "main". What is a *thread*? This is a technical term. In computing, each set of instructions running on the computer is called a thread. At any point there could be several threads of computation running. So, this exception occurred in the "main" thread (that is your `main()` method).

Further, it is also telling you that the error, or exception, that occurred is a known type of error. It is called `java.lang.ArrayIndexOutOfBoundsException`. Just by looking at the name, you should be able to guess that this is an object of some kind. It belongs to the `java.lang` class and its name is:

```
ArrayIndexOutOfBoundsException
```

And, that is exactly the error we introduced when we changed the termination test. We let the loop run beyond the bounds of the array, to an index 5, which in an array of five elements does not exist. Right?

In fact, the index number that caused the exception is also shown (:5), as is the line at which the exception occurred:

```
Crash1.main(Crash1.java:8)
```

That is, at line 8 of Crash1.java (your source code) was where you tried to access an array element that doesn't exist. This is line 8:

```
System.out.println(a[i]);
```

Since I=5 and a[5] does not exist, you get the `ArrayIndexOutOfBoundsException`.

Java includes an elaborate set of classes that define various exceptions. Moreover, when these exceptions occur (technically, when an exception is raised/thrown) it also gives you an option to write your programs so that you include code to handle (or catch) some exceptions. This is called *exception handling*. In many instances, as you will soon see, you are required to provide code for handling anticipated exceptions. We will learn how to handle exceptions next.

Exceptions & Runtime Errors

Java has a class called `Exception` and several dozen types of exceptions that can arise are defined as subclasses of the `Exception` class. Some of the errors occur during input/output (for example, a broken network connection, or a misplaced input file, etc.) and others during the course of executing your logic (for example an out of bounds array access, or trying to access an object before instantiation, etc.). Sometimes, it is possible to recover from an error and to do this Java provides a way for you to detect and specify corrective measures.

Java provides a `try-catch` statement block to detect and handle exceptions. Its basic syntax is shown below:

```
try {

      …some code during whose execution errors may occur…

} catch (<some exception> <variable>) {
   …some corrective action you could perform…
}
```

Essentially, you enclose the code that may (or is likely to) result in an error in a `try-catch` statement block as shown. If, in the course of executing the surrounded code an exception occurs, the code included in the `catch`-block is executed. The following exercise illustrates this.

Exercise 2: Write a program to input a number and output its square root.

Create a new program with the code shown below.

```
import java.util.*;

public class Crash2 {

   public static void main(String[] args) {
       Scanner in = new Scanner(System.in);
       while (true) {
              System.out.print("Enter a number: ");
              String line = in.nextLine();
              int data = Integer.parseInt(line);

              System.out.println("Square root of " + data + " is " + Math.sqrt(data));
       }
   } // main()
} // end of class
```

The code shown above repeatedly prompts the user for a number and then computes and outputs the square root of that number. The number is actually input as a string, as you have learned earlier. You use the Java function:

`Integer.parseInt(<string>)`

to extract a number from the input string. There is also a `Double.parseDouble()` if you want to accept floating-point input.

The while-loop ensures that the program does this forever. Run the program and enter some numbers to try it out. Enter only positive integer values at first. Then, after a few successful tries, enter something that's not a number. What happens?

The program crashes with the following error:

```
Exception in thread "main" java.lang.NumberFormatException: null
       at java.lang.Integer.parseInt(Integer.java: 454)
       at java.lang.Integer.parseInt(Integer.java 527)
       at InputErrorDemo.main(Crash2.java:10)
```

Can you guess why that happens?

Notice, the error is occurring at line 10 of your program. That is the line where you are trying to convert the string to an integer. The exception flagged, `java.lang.NumberFormatException`, is indicating that there was a problem in the number format: it wasn't a number that you entered.

To handle this, we need Java's exception handling.

The `try-catch` Statement

Exercise 3: Read and understand the code below to see how a try-catch block is structured, and used. Modify your program to work this way and experiment with it.

```
import java.util.*;
public class Crash2 {
  public static void main(String[] args) {
    while(true) {
      Scanner in = new Scanner(System.in);
      try {
        System.out.print("Enter a number: ");
        String line = in.nextLine();
        double data = Double.parseDouble(line);
        System.out.println("The square root of " + data + " is " + Math.sqrt(data));
      }
      catch(NumberFormatException e) {
        System.out.println("That's not a number!");
      }
    }
  }
}
```

Run the above program several times. Enter integer values, floating point values, as well as other garbage. Notice how your program is now robust enough to do the right thing when a number is input, and recovers in situations when nothing or some non-numeric input is provided. It does not crash!

File Input/Output (I/O): The `Scanner` class

Assignments in this course will typically deal with lots of data. This data will often be read by your program from a data file. Java has several ways to access files and read data from them. The `Scanner` class is one of the simplest.

Exercise 4: Write a program that reads a text file and prints out is contents. First, create a small data file with a few lines of text. I used these, from a random tweet:

```
@LiamNeeson
I always wanted to be a cowboy
And Jedi Knights
6
Are basically cowboys in space Right?
```

Save the file (I named mine `LiamNeeson.txt`). Now that you have a text file to read, we can build a program that reads and prints out its contents. In order to read text from a file, one line at a time, you do the following:

1. Import the `java.io` and `java.util` packages:
   ```
   import java.io.*;
   import java.util.*;
   ```
2. Create a new `Scanner` object linked to the data file using the `File` class in Java:
   ```
   Scanner input = new Scanner (new File(<name of input file>));
   ```
3. Use the `hasNextLine()` and `nextLine()` methods to look for and read the next line:
   ```
   while(input.hasNextLine()) {
     String line = input.nextLine();
     ...
   }
   ```
4. Close the input stream:
   ```
   input.close();
   ```

Further, the `Scanner` class forces that you enclose the entire operation in a `try-catch` block. This is because in the process of opening the file (Step 2) and testing and reading from it (Step 3), things could go wrong. For instance, the file may not exist, or you typed its name or location incorrectly, etc. The program below puts all these together to show how to read text from a file and then output its contents to the console.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class TextIO {

   public static void main(String[] args) {
       String inFileName = "LiamNeeson.txt";
       Scanner input;
       String line;

       try {
              // Create a new Scanner for the input file
              input = new Scanner(new File(inFileName));

              while (input.hasNextLine()) { // test if there is a line to read
                     // read the next line
                     line = input.nextLine();

                     // output it to Console
                     System.out.println(line);
              }

              // Close the input stream
              input.close();
       }
       catch (FileNotFoundException e) {
              System.out.println("Error in opening the file:" + inFileName);
              System.exit(1);
       }
   }
}
```

Run the program. Did you see the contents of the text file in the console? If not, what went wrong? Once it is running correctly, deliberately change the name of the file and run the program. What do you see? Exceptions at work!

Uncaught Exceptions

What if you don't want to catch an exception for whatever reason and just letting your program crash? After all, you were getting that `ArrayIndexOutOfBoundsException` all the time when you were learning arrays and have never heard of `try/catch`, right? However, sometimes Java insists that certain exceptions MUST be either caught or raised. This means that you cannot simply do nothing. If you prefer to not catch an exception and let your program crash, Java wants you to declare it by throwing the exception (to show that you are aware of it).

`FileNotFoundException` is such an exception.

Take out the `try/catch` block above, save and compile. What happens? You will get the following compilation error:

```
TextIO.java:11: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
       input = new Scanner(new File(inFilename));
               ^
1 error
```

The compilation can be allowed to go forward by having the method in which the exception is generated – in this case `main` to declare throwing that particular exception:

```
public static void main(String[] args) throws FileNotFoundException
```

Make this change now and try it.

Exercise 5: Program your quiz questions, make sure all are working! That is, program the methods themselves, then add appropriate driver code to call the methods you implemented with a variety of parameters to demonstrate that all the methods are working correctly.