CSCE A201
Introduction to Exceptions and File I/O

An **exception** is an abnormal condition that occurs during the execution of a program. For example, divisions by zero, accessing an invalid array index, or trying to convert a letter to a number are instances of exceptions. Java gives you a way to anticipate the occurrence of an exception and write code to handle it.

For example, consider the following code:

```
public static void main(String[] args)
{
      int x = 50;
      int y = 0;
      int z = 0;

      z = x / y;
      System.out.println(z);
}
```

When we reach the line "z = x / y" the program crashes with the message:

```
  Exception in thread "main" java.lang.ArithmeticException: / by zero
      At Excep.main
```

It doesn't even continue on to the next line, but exits immediately. Now, let's see how to handle the exception ourselves.

The exception itself is an object. For simple purposes, you can think of the exception as an error message that indicates what happened. The code or event that makes the exception happen is something that **throws** the exception. The code that handles what to do when the exception is thrown is code that will **catch** the exception. If we don't write code to catch the exception, then Java will catch the exception and halt the program (as well as display an error message).

In order to catch an exception, Java requires that we place the code we think will cause the exception into a **try** block. The example below illustrates a simple example:

```
public static void main(String[] args)
{
	int x = 50;
	int y = 0;
	int z = 0;

	try {
		// Place code we think might cause an
		// exception inside the "try"
		z = x / y;
	}
	catch (Exception e)
	{
		System.out.println("There was an exception!");
		System.out.println("It was: " + e.getMessage());
	}
	System.out.println("The value in z = " + z);
}
```

The try block contains the code we would like to execute but watch for an exception. This is followed by the catch block. Catch is followed by parentheses and takes a single parameter of type Exception. This is the most general class of Exception and is a parent to more specific types of exceptions. If an exception occurs in the try block, an object for that exception is created and is called "e". We can examine the exception or print it out to see what happened, and any other code inside the catch block is executed if the exception occurs.

This program will generate an exception from the division and then begin execution in the catch block. It will output:

```
There was an exception.
It was: java.lang.ArithmeticException: / zero
The value in z = 0
```

Note that the program continues execution after our catch block finishes. It doesn't crash and end as what happens when Java handles the exception!

Here is an example of a program that doesn't crash if we enter letters when we're expecting an integer:

```
import java.io.*;
public class Excep
{
	public static void main(String[] args)
	{
		int i = 0;
		Scanner keyboard = new Scanner(System.in);
		boolean repeatLoop = true;

		while (repeatLoop == true) {
			try {
				i = keyboard.nextInt();
				// If we got here, there was no problem
				// parsing the integer
				repeatLoop = false;
			}
			catch (Exception e)
			{
				System.out.println("There was a problem. ");
				System.out.println("Enter number again.");
				keyboard.nextLine();  // Newline
			}
		}
		System.out.println("You input: " + i);
	}
}
```

This program doesn't actually print the exception, but if it occurs, we don't set the flag that prevents the outer while loop from exiting unless a number is entered, e.g. here is a sample execution, where bold is user input:

hello
There was a problem. Enter number again.
four
There was a problem. Enter number again.
4
You input: 4

There are derived classes from Exception that we can use, and it is usually a good idea to use the most specific class. For example, consider the following program that combines the two possible exceptions together:

```
import java.io.*;
public class Excep
{
	public static void main(String[] args)
	{
		String s = "";
		int i = 0;
		int x = 10;
		Scanner keyboard = new Scanner(System.in);
		boolean repeatLoop = true;

		while (repeatLoop == true) {
			try {
				i = keyboard.nextInt();
				x = x / i;
				repeatLoop = false;
			}
			catch (Exception e)
			{
				System.out.println("There was a problem. ");
				System.out.println("Enter number again.");
				keybard.newLine();
			}
		}
		System.out.println("X = " + x);
	}
}
```

By catching any exception, we can't tell the difference between a divide by 0 exception or a Number format exception. Consider the following execution:

foo
There was a problem. Enter number again.
0
There was a problem. Enter number again.
1
X = 10

We can distinguish the two types of exceptions by using more specific Exception classes that are derived from the generic class Exception. Here is an example where we catch three separate types:

```
import java.io.*;
public class Excep
{
	public static void main(String[] args)
	{
		String s = "";
		int i = 0, x = 10;
		Scanner keyboard = new Scanner(System.in);
		boolean repeatLoop = true;

		while (repeatLoop == true) {
			try {
				i = keyboard.nextInt(s);
				x = x / i;
				repeatLoop = false;
			}
			catch (ArithmeticException e)
			{
				System.out.println("Don't enter zero.");
			}
			catch (NumberFormatException e)
			{
				System.out.println("Don’t enter letters.");
			}
			catch (IOException e)
			{
				System.out.println("I/O exception " + e);
			}
		}
		System.out.println("X = " + x);
	}
}
```

Now we get different behavior for each type of exception when we run the program, e.g.:

0
Don’t enter zero.
Foo
Don’t enter letters.
10
X = 1

Refer to the Java help for a list of the Exception classes. Some common ones are:

Exception	- Any exception
IOException	- Problem with general I/O
EOFException	- Reading of end of line
FileNotFound	- File not found
NumberFormatException	- parse problem from string to number
ArithmeticException	- divide by 0, infinity, etc.

There is more to exceptions, for example you can throw your own exceptions, but that covers the basics. See the text for more details.

File I/O

I/O refers to input and output. So far we can do input from the keyboard and output to the screen. However, it is very useful to be able to take input from another file on disk. Here we will cover only the basics for inputting data from a text file (a file containing only ASCII text data). The textbook has details about reading binary files (a file containing direct bytes of information), which are generally more efficient than text files.

I/O in Java is based on the concept of a **stream**. A stream is a flow of data, where the data could be characters, bytes, or numbers. An input stream is when data flows into your program, which we have been using via System.in. An output stream is when data flows out of your program, which we have been using via System.out.

A file will have two "names": one will be the name of the file on the disk, and the other will be the name of the stream that is connected to the file.

Reading data from a text file

The process of reading data from a text file is similar to the above for setting up the file. Once the file is open, reading from the file is just like reading from the keyboard. The only difference is that the input data will be coming from the file instead of being typed by the user.

Let's say that we have a file on the root of our c:\ drive named indata.txt. The file contains the following data:

c:\indata.txt
```
hello world
300
400
```

Here is a program that can read in this data. Reading in data is very similar to reading from System.in:

```
import java.io.FileInputStream;     // Need this to read a file
import java.util.Scanner;

public class FileInput
{
      public static void main(String[] args)
      {
            int val1 = 0, val2 = 0;
            String s;
            Scanner inputStream = null;
            try {
                  inputStream = new Scanner(new
                        FileInputStream("c:\\indata.txt"));
                  // If we got here, the file was opened
```

```
                s = inputStream.nextLine();
                val1 = inputStream.nextInt();
                val2 = inputStream.nextInt();

                System.out.println("Line = " + s + " val1 = " + val1
                        + " val2 = " + val2);

                inputStream.close();
            }
            catch (Exception e)
            {
                // If any exception occurs, exit
                System.out.println("Error " + e);
                System.exit(0);
            }
        }
}
```

We need the two slashes in c:\\indata.txt because \ is the escape character. Two slashes means one slash, so this is equivalent to the string c:\indata.txt. If you don't put a full pathname (i.e. only put the name of the file) then Java will look for the file in the same folder as your Java class files.

When we open the file, it should be inside a try/catch block. When reading from a file, an exception might occur (such as the file doesn't exist or is corrupted).

First, here is what we get if the file indata.txt is not located on the c:\ drive, we stop executing code in the try block and hit the catch:

> Error java.io.FileNotFoundException: c:\indata.txt (the system cannot find the file specified)

Here is what we get if the file does exist with the data:

Output:

> s = hello world val1 = 300 val2 = 400

The book also has a description of the StringTokenizer class that makes it easy to extract values separated by some delimiter.

Here is another example that reads in words from a dictionary file to solve this word puzzle: "Name a common word, besides tremendous, stupendous and horrendous, that ends in dous."

We can solve this problem by loading up a file of words and checking each one to see if:

1) It is more than 4 letters long
2) The word contains "dous" at the end

Our strategy will be to read each line of the file, see if it contains the text "dous" and if it does then print it out. This will result in some false positives since "dous" might be found mid-word, but there should not be many of these cases. We can just visually inspect the list of words for the fourth word that ends in "dous. "

We can keep reading a line until the `hasNextLine( )` method returns false. This occurs when we've read every line in the file.

Assume we have a file of English words located at C:\WORDS.TXT, code follows:

```
import java.io.FileInputStream;
import java.util.Scanner;

public class WordFinder
{
	public static void main(String[] args)
	{
		String s;
		Scanner inputStream = null;
		try {
			inputStream = new Scanner(new
				FileInputStream("c:\\words.txt"));
			while (inputStream.hasNextLine())
			{
				s = inputStream.nextLine();
				s = s.toLowerCase();
				if (s.indexOf("dous")>0)
				{
					// Candidate word, print it out
					System.out.println(s);
				}
			}
			inputStream.close();
		}
		catch (Exception e)
		{
			// If any exception occurs, exit
			System.out.println("Error " + e);
			System.exit(0);
		}
	}
}
```

Exercise: Modify the trivia game from the Array lecture so the trivia questions are read in from a file instead of hard-coded in the program.

Saving data to a text file

First let's see how to save data to a text file. To do so, we will use a class called PrintWriter. Here is an example program that creates a file named "output.txt" and saves the numbers 1 through 10 in it:

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
public class TextFile
{
      public static void main(String[] args)
      {
            int i;
            PrintWriter outData = null;   // Initialize stream to empty

            // First we must open the file.  Opening must be
            // inside a try/catch block
            try
            {
                  outData = new PrintWriter(
                        new FileOutputStream("output.txt"));
            }
            catch (Exception e)
            {
                  System.out.println(e);  // Print error
                  System.exit(0);         // Exit if an error
            }
            // If we get to this point, the file was
            // successfully opened
            for (i=0; i<10; i++) {
                  // Println outputs to file instead of screen
                  outData.println("Number " + i);
            }
            outData.close();  // Close file when done
      }
}
```

This program declares a variable of type "PrintWriter" called outData. This will be a variable connected to our file for output. Initially it is set to null, which is empty.

To create the file, we create a new PrintWriter with a new FileOutputStream object. This particular invocation will destroy any file that might currently exist with the same name in the same directory as the new file. The parameter to the FileOutputStream object is a string that is the name of the file we want to create. If we give it just a file name, as in "output.txt" then when this program is run it will create a new file called "output.txt" in the current directory. If you're using NetBeans then it will create this file in your project directory.

Note that we could also give a complete path if we wanted to make the file somewhere else. The following would have created the file on the root of the C: drive:

```
outData = new PrintWriter(
      new FileOutputStream("c:\\output.txt"));
```

To save data to the text file, we use the stream variable followed by the println method. Println behaves just like we've been doing with System.out.println. In this example, we are just outputting ten lines with numbers in them.

When we're done writing to the file, make sure to close the file with the close() method.

This program creates a file named output.txt in the default directory with the contents:

```
Number 0
Number 1
Number 2
Number 3
Number 4
Number 5
Number 6
Number 7
Number 8
Number 9
```