BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
First Semester, 2016-17
Object-Oriented Programming (CS F213)
LAB-8 [Exception Handling and Text File I/O]

AGENDA

1. Catching exceptions locally
2. Throwing exceptions and catching them remotely
3. Using our own exception classes
4. Reading from a file
5. Writing to a file
6. I/O related exceptions

Overview

When a problem is detected in a Java program during execution, the standard approach is to throw an exception to indicate the problem has occurred. The code that caused the problem immediately stops executing, and Java enters into exception handling mode, which searches for code that can resolve (i.e., handle) the problem. If it can't find that code, the program will crash and a message is displayed describing what went wrong and where the problem occurred. Here are some of the common pre-built exceptions:

- ArithmeticException (e.g., divide by zero)
- ArrayIndexOutOfBoundsException
- NullPointerException

Java programmers use try-catch statements for exception handling. Recall that the basic form of this statement is:

```
try {
        // statements, including method calls, that might cause exceptions
} catch (<exception class name> exception) {
        // statements to handle this class of exceptions
}
```

In this lab you will be working with an application that manages bank accounts, experimenting with catching and throwing exceptions, and adding the ability to do file input and output.

Getting Started

Begin by starting a new project in Eclipse. Name your project yourID(2011_456) and create a package with a name test_bank. Next, copy the following files into this package. These files are supplied to you along with the lab sheet.

- Bank.java
- BankAccount.java
- BankApp.java

- InsufficientFundsException.java

For task 4, you will also need to copy the following files to your package:

- accounts.txt
- badinput.txt

Task 1: Exceptions in Action and Catching Exceptions Locally

Start by running the **BankApp** class. Using the menu, try option 2 to deposit funds even though no accounts have been created. Note that an error message is displayed, but the program doesn't crash. The program detects that we haven't either created a new account or found an existing account to deposit the funds to. How is this done? Take a look at the code in the **BankApp** class. The application uses a reference variable, named account, for the current account, which is initially null. Find case 2 of the switch statement in the **BankApp** class. Note that we've coded the program to try to deposit money into the account, but since the account reference variable is null initially, the JVM (Java Virtual Machine) throws a **NullPointerException**. We've coded our program to use exception handling to catch this kind of exception, which we handle by displaying an error message to tell the program user what corrective action is needed. (The other catch clause in this try-catch statement will be discussed in the next task.)

Next try option 3 and see what happens; it's not the sort of thing we'd like to happen with our programs! Let's correct that problem in a manner similar to what was done in case 2 of the switch statement. Add a catch clause so that the program displays an appropriate error message rather than crashing. What we've done is added an exception handler to catch an exception locally, that is, in the same method where the problem occurs.

Now run the program again and check that option 3 gives error messages when the account is null. Then create a new account using option 1. Give an initial balance of $100.11 and an account number of 0. The program will create the new account and now you can try menu options 2 and 3 to deposit and withdraw funds.

Task 2: Throwing Exceptions and Catching them Remotely

We've seen that the JVM can throw exceptions, but our programs can also throw exceptions as we'll now see. Review the code in the constructor in the **BankAccount** class. What happens if we attempt to make a new account with a negative balance? Give it a try by running the **BankApp** and doing menu option 1 to attempt to create an account with a negative balance.

What happens? An **IllegalArgumentException** exception is thrown with a message we've added saying that this cannot be done. Rather than Java's VM throwing the exception, our program does this using a throw statement if a balance less that 0 is given. For this task, add code to the constructor so that a 5 digit account number is required (Hint: What range of values gives a 5 digit number?). If an account number isn't 5 digits, use exception handling to throw an exception similar to what was done for the negative balance. Change the message that is added so that it's appropriate for this problem, then give your code a try to ensure it works as expected.

We catch the exception in the main method of the **BankApp** class, instead of in the constructor where the problem occurs. When an exception is caught in this manner, that is, in a method other than the one where the problem occurred, we call it remote catching. Exceptions can be either caught locally or remotely depending on where it most makes sense to try to resolve the problem.

Our **BankAccount** class also throws **IllegalArgumentExceptions** in the **deposit()** and **withdraw()** methods to indicate when other problems occur. Review that code for another example.

Task 3: Creating our own Exceptions

So far we've been using the exceptions that are provided by Java, but we can also create our own exception classes! Take a look at the **InsufficientFundsException** class that we've provided. Note that the class body is quite trivial because we're relying on Java's Exception class to do the work for us. How? In the class header we've put the code extends Exception, which results in our class getting its code from Java's already implemented Exception class (this is called inheriting). All we have to do is choose a name for our new exception class and code a constructor in the pattern like what is shown (which enables our exception class to have messages added like what we did in Task 2).

Let's put this new exception to work. In the **BankAccount** class's **withdraw()** method, add code to check if the amount given exceeds the account's balance. If it does, then throw a new **InsufficientFundsException**. Add a message to the constructor's argument list saying "You can't withdraw more than you have!"

There's one more thing we need to do to this method when we use Exception in this manner. Java gives us two choices, checked exceptions and unchecked exceptions. Exception classes that extend Exception are checked and those that extend **RunTimeException** are unchecked. Checked exceptions are used for the kinds of things that could still go wrong even if we carefully coded our program (e.g., **FileNotFoundException**, **ArithmeticException**). For these, the compiler actually checks that the programmer knows the exception might be thrown by requiring either a throws clause or a catch block in the method(s) that might throw checked exceptions. Since our new exception class is checked, we'll add a throws clause to the withdraw method's header so it looks like this:

```
public void withdraw(double amount) throws InsufficientFundsException {
```

This clause tells the compiler that we know that the exception might be thrown and that the withdraw method won't try to fix the problem. Instead, it throws the exception to the method that called withdraw.

Next, we'll remotely catch this exception in the **BankApp's** main method. In case 3, add a catch clause to handle this kind of exception. All it needs to do is display an error message along with the exception's message like what was already done for the **IllegalArgumentException**. When you've completed this

code, give your program a try. Create an account and try to overdraw the balance. You shouldn't be able to!

Task 4: Reading information from a file

In this task, you will be creating code to read bank accounts from a file. We'll isolate file handling from our other classes by creating a separate class that will handle all file input (and output). Create the new class and named it **BankIO**. This class will not be instantiable (i.e., it won't have any instance members). The **BankIO** class has the following two class (i.e., static) methods:

- **public static void readFile(String filename, Bank bank)**
  This method should create a **Scanner** object that reads from the given file specified by filename. While there's input left in the file, the method repeatedly calls the **readAccount** method below that returns a **BankAccount** object, and add that bank account to the given bank.
- **public static BankAccount readAccount(Scanner in)**
  This method is used to read in one bank account. It's passed a **Scanner** object and reads the **account number** (an int) and the **balance** (a double) of one line of the file. The method should then create a new **BankAccount** object and return it.

Recall the pattern of code for reading from a file is:

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

... method(...) {

    Scanner in;
    try {
        in = new Scanner(new File(filename));
    }
    catch (FileNotFoundException e) {
        ...
    }

    ...
}
```

If a **FileNotFoundException** is thrown, the method should display an error message and return. You may assume the file is in the correct format. In the challenge task below, you'll be improving your code to handle files that are improperly formatted.

Reading from a Formatted File

Often we are required to read information from a file that is in some specified format. The format tells you the order and layout of information found in the file. When doing file processing, we want to be able to read valid files that follow the format specifications (and reject invalid files) For our bank program, each line in a file has two pieces of information; an integer account number followed by a double account balance (for an example, see **accounts.txt**). Input files may have any number of lines. To know how many lines are in the file, you can use the **hasNextLine()** method of the **Scanner**. For each line, you'll need to separate out the account number and the balance. Here are two ways to do this:

Approach 1:

The easiest way to read information from a file requires that the values be separated by whitespace (i.e., spaces, tabs, newlines). Our file is formatted this way so we'll use the Scanner to read each piece of information separately (i.e., use **nextInt()** and **nextDouble()**). This approach works well as long as you're careful with processing the newline, which marks the end of each line.

Approach 2:

If the values in a formatted file are separated by some other delimiter, such as a comma, it is better to read each entire line of the file separately (using **nextLine()**) and then divide the line into the appropriate pieces. There are several ways to do this, but we suggest using the split method of the String class. Try the following code to get a feel for how split works:

```
String line = "sample,line,1,23.32";

//this means that each piece is separated by a comma
String delims = ",";

//this divides the line into pieces
String[] tokens = line.split(delims);

System.out.println("Line contains " + tokens.length
                                    + " pieces of information:");

//print out the tokens array to see what it contains
for (int i = 0; i < tokens.length; i++) { System.out.println(tokens[i]);}

//now convert the third token to an int value
int myInt = Integer.parseInt(tokens[2]);

//now convert the fourth token to a double value
double myDbl = Double.parseDouble(tokens[3]);
System.out.println(myInt + ", " + myDbl);
```

Testing

Create a new class named **BankFileTester** that creates a **Scanner** object and asks for a file name from the user. Pass the file name to the **readFile** method. Test using **accounts.txt** file.

Task 5: Writing information to a file

Add a method, named write, to the **BankIO** class which takes a Bank and a String filename and writes the bank to a file with the given name. The method should make use of the **toString** method of the Bank class. If a **FileNotFoundException** is thrown, the method should display an error message and return. Recall to write to a file you use the following pattern:

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;

... method(...) {

    PrintWriter out;
    try {
        out = new PrintWriter(filename);
    }
    catch (FileNotFoundException e) {
        ...
    }
    ...
}
```

Testing

Add code to the **BankFileTester** class which will prompt the user for a file name and then call the write method of the **BankIO** class. Check that your method worked by looking in your project file. In the Package Explorer window of Eclipse, select the folder for your Bank Lab project, and then press "F5" to refresh the project. You should now see the file name you specified, which you can open to verify it has the correct contents. If contents in the file are incomplete, is there something you forgot to do when you finished using your **PrintWriter** object?

Challenge Task: Bad Input Files

In the time that is remaining in the lab, modify your **readAccount** method so that it throws an **IOException** if the line of input being read is invalid. Recall that **IOException** is a checked exception so you'll need to add a throws clause to the method header. Modify your **readFile** method so it also throws **IOExceptions**.

Lastly, modify your **BankFileTester** program to catch **IOExceptions** and handle them by displaying an error message and giving the user the opportunity to enter a new filename. Test using **accounts.txt** and **badinput.txt**.

Excerise-1

Exception Handling in Java

This exercise makes use of the working principle of a thermostat to demonstrate the concept of Exception Handling in Java.

A thermostat is a control system which regulates the temperature of a system so that the system's temperature is maintained near a desired temperature. The thermostat does this by switching heating or cooling devices on or off to maintain the correct temperature.

When the temperature hits an extreme (UPPER_LIM=100, LOWER_LIM = 50), the process is reversed i.e., if the temperature was increasing then the heating is stopped and the cooling process is started and vice versa.

In this exercise, when the temperature hits either extreme, an exception (***HighTemperatureException*** in case of upper extreme & ***LowTemperatureException*** in case of lower extreme) is thrown. This exception causes the reversal of process. The handling mechanism of this mechanism would be to start the reverse process.

The program defines a class Thermostat which has control methods to start and stop the thermostat. It also has a parameterized constructor to initialize the thermostat to an initial temperature. The temperature limits are specified initially before the start of the thermostat operation. Once the thermostat operation is started by creating an instance of the Thermostat class inside the ThermostatDriver class, which is the driver class for the Thermostat class, it goes in to a cycle of heating and cooling. The cycle proceeds until it is interrupted by the user.

Incomplete Skeleton of the code is given below and **You have to complete it as per commented specification**.

```
import java.util.*;
// Exeception Classes
class InvalidInitialTemperatureException extends Exception
{
	private int temp;
	InvalidInitialTemperatureException(int temp)
	{
		this.temp = temp;
	}
	public String toString()
	{
		return "InvalidInitialTemperatureException : "+this.temp;
	}
}

/*******************************************************************************/

class HighTemperatureException extends Exception
{
```

```
	HighTemperatureException()
	{}
	public String toString()
	{
		return "\nHigh Temperature Exception : Cooling down\n";
	}
}

/*******************************************************************************/

class LowTemperatureException extends Exception
{
	LowTemperatureException()
	{}
	public String toString()
	{
		return "\nLow Temperature Exception : Heating\n";
	}
}

/*******************************************************************************/

// Class Thermostat
class Thermostat
{
   private int temperature;
   static final int LOWER_LIM = 50;
   static final int UPPER_LIM = 100;

   Thermostat(int initTemp) throws InvalidInitialTemperatureException
   {
			if((initTemp>=LOWER_LIM)&&(initTemp<=UPPER_LIM))
			{
				this.temperature=initTemp;
				System.out.println("Thermostat Starting. With Initial Temprature
:"+temperature);
			}
			else
			{
				throw new InvalidInitialTemperatureException(initTemp);
			}
   }

   public void startThermostat() throws HighTemperatureException
   {
	System.out.println("**********Thermostat Started**********");
	/*
	This method increments (+1) and displays the temperature of the Thermostat after every
	1000 ms. When the temperature reaches the UPPER_LIM it raises
	HighTemperatureException.
	*/
	/WRITE YOUR CODE HERE
   }

   public void stopThermostat() throws LowTemperatureException
   {
```

```
        System.out.println("*****************Thermostat Stopping*************************");
        /*
        This method decrements (-1) and displays the temperature of the Thermostat after every
        1000 ms. When the temperature reaches the LOWER_LIM it raises
        LowTemperatureException.
        */
        /WRITE YOUR CODE HERE

}
/*******************************************************************************/
class ThermostatDriver
{
   public static void main(String[] args) throws InvalidInitialTemperatureException
   {
                Thermostat t=new Thermostat(55); //Setting the initial temperature of the thermostat as 55.
                while(true)
                {
                        try     {       t.startThermostat();    }
                        catch(HighTemperatureException e)   {       System.out.println(e);   }

                        try     {       t.stopThermostat();     }
                        catch(LowTemperatureException ex)   {       System.out.println(e);   }
                }
   }// End of main()
}// End of ThermostatDriver
```

==========