Exercise 3, Exceptions – when things go wrong

Objective

The primary objectives for this lab are to be able to throw and catch exceptions and have a full understanding of the Java runtime’s propagation mechanism for exceptions until a handler is found.

Overview

Firstly, you will handle the exceptions by throwing an exception object in your Account class constructor if an invalid value is passed in for the initial account balance. You will then add code to use exceptions and write your own exception type.  
  
In a second practical you will enhance the MotorwaySimulator to throw a ‘No RegPlates Available’ type of Exception from the RegistrationPlateFactory which then can be re-thrown by the Vehicle constructor (that was trying to get a RegistrationPlate) before being caught by the program that can display a suitable message.

Part 1 – Step by step

Using try/throw/catch in the Account class

1. In Package Explorer expand **Starter.Account.**
2. Execute the application. Note that we have ‘successfully’ created an account with a negative balance! This is because the Account's constructor does not test for this possibility. We'll redesign the constructor to throw an exception.
3. Open **Account**.java and locate the constructor that takes three parameters. Add code to this constructor that checks the incoming balance parameter.   
     
   If this parameter is negative, throw a new *IllegalArgumentException* object. Pass in a sensible error message. Your code may look like:

**if** (balance < 0.0) {

**throw** **new** IllegalArgumentException("Negative balances are not supported");

}

1. Run the program. You should see the unhandled exception message.
2. Open the main() method in Program.java. Place the code that constructs the two Account objects inside a try block. Catch exceptions of type *IllegalArgumentException* and display the exception's message.   
   Your code might look like this:

**try {  
 ac1 = new Account( ... );  
 ac2 = new Account( ... );  
}  
catch( IllegalArgumentException exn ) {  
 System.out.println( exn.getMessage() );  
}**

1. Build your code. You get two compiler errors. Why? And how can you fix them?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The errors are caused because the compiler has detected it’s possible that **ac1** or **ac2** might not have been created when their getDetails() is called.

1. Move the following lines inside the try block, after **both** the objects are created.  
    Account.transfer(500, ac1, ac2);

System.***out***.println(ac1.getDetails());

System.***out***.println(ac2.getDetails());

1. Build and run the code now.   
   The Console displays the information from the exception.
2. Examine the Account class again. Where else can you see parameters that ought to have their ranges tested?
3. The withdraw() and deposit() methods should both have their amount parameters checked, because withdrawing a negative amount would act as a deposit, and depositing a negative amount would be like a withdrawal!

Quickly fix these two methods now using the same technique that you used before, throwing the same type of exception but with a **different** ‘message’.  
**Tip: if** (balance < 0.0) {

**throw** **new** IllegalArgumentException("Negative balances are not supported");

}

1. Now **include appropriate** method calls in your test harness to test both a negative deposit() and a negative withdraw().
2. Why don't you need to fix the transfer() method? Think about the flow.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. There is no need to test the amount parameter in the transfer() method, because it calls the withdraw() method from within itself.   
   If the withdraw() method throws an exception, the transfer() method doesn’t contain a try or catch block and so the exception would ripple back to the client (the main method – where you do have a handler).
2. Let's prove this now. Ensure the initial balance for the "Jane Doe" account is a £200. Remove any calls to deposit() or withdraw() from the test harness. Add a line of code inside the try block in main after the two account objects are created, which attempts to transfer an amount of £50 from Joe’s account to Jane’s account. (Tip: Account.transfer(50, ac1, ac2);)
3. Build and run. Jane should now have 250 units in her account.
4. Change the code so that you try to transfer() -50 units.   
   Tip: Account.transfer(-50, ac1, ac2);
5. Build and run the code again. You now get the exception.
6. View the Account's transfer() method and spot the auditing code.   
   We need this code to execute **no matter** whether an exception occurs or not. You will also notice that the audit code uses the local result variable to indicate success or failure. How can you ensure that this statement will **always** execute, even in the face of an exception?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Put a try block around the code that calls the withdraw() and the deposit() methods. But put the from.Audit() statement inside a finally block that follows the try block.  
     
   Your code for the transfer() method should now look like this:

**boolean** result = **false**;

**try** {

result = from.withdraw(amount);

**if** (result) {

to.deposit(amount);

}

result = **true**;

} finally {

from.audit("Transfer attempt to " + to.number +   
 ". Completion Status: " + result);

}

1. Build and run the application. You will always see the audit information regardless of whether an exception is thrown.

Creating a custom exception class

The remaining parts of the practical exercise involve deriving your own exception class from class RuntimeException (this means the compiler will not demand that it is caught). This section of necessity scratches the surface of inheritance techniques that will be covered fully very soon.

Step by step

In this section of the lab you will create your own exception class called InsufficientFundsException. You will then use it inside the withdraw method to indicate that there are insufficient funds for the withdrawal, rather than use the boolean return value.

1. We have added already a class to the project called: InsufficientFundsException.
2. Set the base class for the InsufficientFundsException to be RuntimeException, by removing the comment shown below.

public class InsufficientFundsException   
 // extends RuntimeException   
{

}

1. The class contains two public fields. The first, called acc, is of type Account. The second, called amt, is of type double.
2. We have started the constructor for you. It accepts two parameters, of type Account and double respectively, you must assign these parameters into the two final fields mentioned above. Your code should look like this:

public class InsufficientFundsException extends  
 RuntimeException {  
  
 public Account acc;  
 public decimal amt;  
  
 public InsufficientFundsException(Account ac,   
 decimal amount ) {  
 ?? = ??;  
 ?? = ??;  
 }  
}

This class is now complete, close the file.

1. Go to class Account.   
   We need to modify the withdraw() method so that it throws an InsufficientFundsException when appropriate.   
     
   To avoid you making numerous changes to your current withdraw() method, we have partly written a new ‘replacement’ one for you, so that you can keep both in your code and contrast and compare.   
     
   **Comment out** the current withdraw() method totally and **uncomment** the replacement that is hidden below it.
2. You will notice that it is a void method that no longer declares, initialises, changes and returns a boolean variable called result. It either throws one of two exception types or changes the balance.  
     
   Those are the three possible outcomes. Any method that calls withdraw will get an unhandled exception back if they ‘try’ to withdraw too much or ‘try’ to withdraw a negative amount.   
     
   But one line is still commented out. Uncomment and complete the line of code that throws an InsufficientFundsException**.**Which account does not have enough money? *This* one!

public void withdraw( decimal amount ) {  
 if( amount < 0.0 ) {  
 throw new IllegalArgumentException( ... );  
 }  
  
 if( amount > balance ) {  
 throw new InsufficientFundsException(**???**, amount);  
 }  
  
 balance -= amount;  
}

1. Build your code, transfer() does not compile as it is trying to ‘catch’ the return from a ‘void’ method.
2. You will now need to **carefully** modify the transfer method by changing its return type to void. However, you **cannot remove** the ‘result’ variable as it is needed by ‘audit’ to indicate success or failure.   
     
   Internally, its code can now also be simplified. Can you work out how? This is a good test of your understanding.  
     
   Remember that auditing **always happens** and it **must show ‘true’ if the** withdraw() **and** deposit() **both worked**.  
     
   Many students struggle here – think hard about flow and what you are trying to achieve.
3. Finally, modify the try / catch section in the main method of the Program class by introducing a second catch block to handle the InsufficientFundsException.   
     
   Display an appropriate message by accessing the Acc and Amt values of your exception object.  
      
   Again, your code might look something like this:

}  
catch( IllegalArgumentException exn ) {  
 ...  
}  
catch( InsufficientFundsException isfe ) {  
 System.out.println(“Not enough cash to withdraw ” +  
 + isfe.amt + “ from Account:“ + isfe.acc.getNumber() +  
 " as “ + isfe.acc.getHolder() +   
 “ has a current balance of only “ +   
 isfe.acc.getBalance());  
 // or (better)  
 System.out.printf("Not enough cash to withdraw   
 %7.2f from Account:%s as %s has a current   
 balance of only %7.2f\n",   
 isfe.amt,   
 isfe.acc.getNumber(),   
 isfe.acc.getHolder(),   
 isfe.acc.getBalance());

}

1. Now modify your test code in main() in Program to force an InsufficientFundsException to be generated. Build and test the application and prove that the exception is generated, caught and elegantly handled.
2. For a full ‘belt and braces’ approach you could add a third exception handler for type Exception – this would ensure the user is never looking at a ‘Just-In-Time’ debugging window.

Part 2 – Step by step – when you have time

Using try/throw/catch in the MotorwaySimulator

1. In Package explorer in project 03\_Exceptions\_WhenThingsGoWrong. Expand ‘src’ and the three packages Starter.Main, Starter.FactoryStuff and Starter.Vehicle.  
     
   Open the four classes in these three packages.  
     
   This ‘starter’ is where you progressed the application to at the end of the last chapter, i.e. the ‘solution’ solution from chapter ‘More on statics’.
2. Go to the RegistrationPlateFactory.  
   Replace the statement that creates the dummy “XXXX” plate with a line that simply throws an instance of class Exception, passing “No more reg plates” to its constructor.  
     
   Your code will not build. Class Exception is most definitely a ‘checked’ exception type and so the method signature must acknowledge that and ‘pass the buck’ to whoever is calling it.  
     
   Add the code ‘throws Exception’ to the method signature.  
     
   Now the method compiles but immediately class Vehicle fails to compile.
3. Now go the code that calls the RegPlateFactory method – the Vehicle constructor.  
   The first two statements that assign values to Speed & Lane can stay as they are.

Ensure that the ‘getting’ of a RegistrationPlate is the 3rd statement and the incrementing of the Count the fourth (so that the count is not incremented if the call to produceNextRegistrationPlate() throws an exception).  
  
If you were to put these final two statements in a try block (don’t), followed by an empty catch block, (for type Exception) your code would compile.  
  
But you can’t leave your code like that as now anyone who tries to create a Vehicle would have no notification that there was a problem and so will probably then make a call to the getDetails() method of the ‘apparently successfully created’ Vehicle and will get a null reference exception when it tries to extract the registrationNo from the ‘null’ registration. You need to pass the buck again by declaring the Vehicle constructor as potentially throwing the Exception.  
  
Add the statement ‘throws Exception;’ to the constructor signature.  
  
  
The Vehicle code effectively pretends it does not know who is running the Vehicle constructor and simply passes the buck.

1. So now main() of Program will not compile.   
   Go there now.  
   The creation and adding of the Vehicle references should now be placed in a try block with a catch block displaying the message.

catch(Exception e) {  
 System.err.println(e.getMessage());  
}

1. Re-test the application and you will see the ‘No more reg plates available’ message appearing in ‘red’ in the console output.  
     
   Also the count of vehicles will only be six as we bypassed the count++ statement when the produceNextRegistrationPlate() method threw an unhandled exception.  
     
   Don’t worry that the red message does not necessarily appear ‘last’ in the output. This is just a timing feature of how the IDE flushes out stuff to standard ‘out’ and standard ‘err’.
2. However, there are two problems.  
   Firstly, if you change the first loop to run say 10 times you will get four ‘No more plates’ messages. You could issue a break; statement inside the catch block so that no more Vehicles are created when the Reg Plates are all used up.  
     
   Secondly, a bigger problem – the ONLY way that this ‘No more plates’ problem can be handled is by catching EXCEPTION (the actual class that is being thrown), as in throw new EXCEPTION(“..”);.  
     
   So if now any code throws any other sort of exception for whatever reason that does not have a specific handler, then it will be caught by this ‘catch every other sort of Exception handler’. This could result in the ‘No more reg plates available’ message appearing when it is not the problem at all.

Enhancement

1. Quickly define a class called NoMorePlatesException which extends Exception.

class NoMorePlatesException extends Exception {  
   
}

1. Alter the Factory class to throw this exception type instead (can’t pass anything to its constructor) and change the throws clause in the method signature.
2. Make the corresponding changes to class Vehicle so that its constructor signature says ‘throws NoMorePlatesException’.
3. Now **crucially** main is not forced to include a catch of EXCEPTION, only a catch of NoMorePlatesException. Change the catch block in the ‘for’ loop.  
     
   Class Program can code a separate catch block for Exception (good practice) for any other indeterminate type of Exception that might get thrown at some future time.
4. Re run your code, expecting it all to work but you will see ‘null’ in red as there is no ‘message’ inside the NoMorePlatesException object.  
     
   Quickly change the first catch block in main to print “No more plates” to System.err.  
     
   Soon we will show you (later chapter) how you could pass a parameter to the construction of the NoMorePlatesException object AND cause your String to be stored away such that e.getMessage() would retrieve it.