Lab 7

Objectives

- Describe the difference between checked and unchecked exceptions
- Write code in Java to throw and catch checked exceptions
- Add code to stack and queue implementations to support exceptions

Part 1

For Part 1, you will be inspecting an implementation of the Stack interface that throws exceptions (in place of where we there we pre-conditions earlier in the semester).

1. Download the files for this lab into your Lab7 folder.
2. Open `Stack.java`, `StackArrayBased.java` and `Lab7Tester.java`.
3. Find the methods in the Stack interface that throw the `StackEmptyException`. Look at the implementation of these methods in the `StackArrayBased.java` file.
4. Look at the calls to the Stack methods in the `testBasicStack()` method in `LabTester.java` and answer the following questions on paper:
    a. Which method calls are wrapped in try/catch blocks
    b. What does the tester do within the catch block if it does not expect to reach the catch block?
5. Search for the TODO tag in `Lab7Tester.java`. In its place, write tests as described to test whether the exception is correctly thrown when it should be.
6. When you have finished verifying the array-based stack uses exceptions correctly, you will move on to the `reverseString` and `doBracketsMatch` methods in `Lab7Tester.java`. For each method (**one at a time**) do the following:
    a. Uncomment the code within the method you are working on.
    b. Determine which method calls on the stack are calling a method that throws an exception.
    c. Wrap all such calls in a try/catch block.
    d. Within the exception catch block, add appropriate code for the logic of the method. That is, if **pop** is called on an empty stack, what should be done in the catch block? What happens in the `reverseString` method's catch block is likely different than it is in `doBracketsMatch`?

CHECKPOINT – At this point you should be able to compile and run `Lab7Tester.java` with `testStackUseFunctions` uncommented. If this isn't working, please ask a TA for support. If you discuss your progress with a TA, make sure you can:

- demonstrate your tests to ensure the exceptions are thrown in the correct case
- how you have used the catch block to control the flow of your program in the methods tested by `testStackUseFunctions`.

Part 2

In Part 2 of this lab you will be inspecting an implementation of a generic Queue interface and updating it to throw exceptions in place of the implicit preconditions.

1. Open `Queue.java`, `QueueRefBased.java`. This is a generic implementation of a Queue.
2. Find the methods in the Queue interface that have the precondition that the Queue must not be empty. For each of these methods, remove the precondition and change the method so it throws a `QueueEmptyException` (provided for you) in place of the precondition. Again, this will cause you to change your code in multiple places for these updated methods:
   a. Signatures of method prototypes in the interface
   b. Signatures of method implementations in the class that implement the interface
   c. Add code to throw the new exception in the correct case in the method implementations
   d. All calls to these methods (in `Lab7Tester.java`) must be wrapped in a try/catch block.
3. Search for the TODO tag in `Lab7Tester.java`. In its place, write tests as described to test whether the exception is thrown when it should be.

REMEMBER: if you get a warning like the following…

```
Note: Lab7Tester.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Do as suggested and recompile as follows (it will indicate the line numbers of where the problem code is):

```
javac -Xlint:unchecked Lab7Tester.java
```

CHECKPOINT – At this point you should be able to compile and run all of the tests in `Lab7Tester.java`.

Part 3

For Part 3, you will be tracing code. You will need to demonstrate the output for each part to a TA to receive credit for Part 3.

1. What is the difference in output between the code examples below, given an initial call to `method1()`?

```java
public static void method1() {
   System.out.println("start of method1");
   int[] array = {5, 4, 3};

   System.out.println(array[3]);

   System.out.println("end of method1");
}
```

```java
public static void method1() {
   System.out.println("start of method1");
   int[] array = {5, 4, 3};
   try {
      System.out.println(array[3]);
   } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("caught exception");
   }
   System.out.println("end of method1");
}
```

2. What is the output of the following code, given an initial call to `first()`?

```java
public static void first() throws ExceptionA {
   System.out.println("start of first");
   other(100);
   System.out.println("done other(100)");
   other(23);
   System.out.println("done other(23)");
   other(15);
   System.out.println("done other(15)");
   System.out.println("end of first");
}
```

```java
public static void other(int x) throws ExceptionA {
   System.out.println("start of other");

   if (x%2 == 0) {
      System.out.println("other throws");
      throw new ExceptionA();
   }
   System.out.println("end of other");
}
```

3. What would happen if we tried to compile and run the program again after removing the `throws AException` from the method signature of `first`? The signature would now be:
`public static void first() {`

4. Given we add `try` and `catch` blocks to first, as shown below, what is output?

```java
public static void first() {
   System.out.println("start of first");
   try {
      other(100);
      System.out.println("done other(100)");
      other(23);
      System.out.println("done other(23)");
   } catch (ExceptionA) {
      System.out.println("caught ExceptionA");
   }
   System.out.println("end of first");
}
```

```java
public static void other(int x) throws ExceptionA {
   System.out.println("start of other");
   if (x%2 == 0) {
      System.out.println("other throws");
      throw new ExceptionA();
   }
   System.out.println("end of other");
}
```

5. What is the output of the program now that a third method has been added?

```
public static void first() {

   System.out.println("start of first");

   try {
      other(23);
      System.out.println("done other(23)");

      other(15);
      System.out.println("done other(15)");

   } catch (ExceptionA) {
      System.out.println("caught A in first");
   }

   System.out.println("end of first");
}
```

```
public static void other(int x) throws ExceptionA {
   System.out.println("start of other");
   if (x%2 == 0) {
      System.out.println("other throws");
      throw new ExceptionA();
   }
   try {
      System.out.println("calling third");
      third(x);
      System.out.println("done calling third");
   } catch (ExceptionB e) {
      System.out.println("caught B in other");
   }
   System.out.println("end of other");
}

public static void third(int x) throws ExceptionB {
   System.out.println("start of third");
   if (x%3 == 0) {
      System.out.println ("third throws");
      throw new ExceptionB();
   }
   System.out.println("end third");
}
```

6) What is the output now the `other` does not have `try` and `catch` blocks for `ExceptionB`?

```
public static void first() {

   System.out.println("start of first");

   try {
      other(23);
      System.out.println("done other(23)");

      other(15);
      System.out.println("done other(15)");

   } catch (ExceptionA) {
      System.out.println("caught A in first");
   } catch (ExceptionB) {
      System.out.println("caught B in first");
   }

   System.out.println("end of first");
}
```

```
public static void other(int x)
      throws ExceptionA, ExceptionB {

   System.out.println("start of other");
   if (x%2 == 0) {
      System.out.println("other throws");
      throw new ExceptionA();
   }
   System.out.println("calling third");
   third(x);
   System.out.println("done calling third");

   System.out.println("end of other");
}

public static void third(int x) throws ExceptionB {
   System.out.println("start of third");
   if (x%3 == 0) {
      System.out.println ("third throws");
      throw new ExceptionB();
   }
   System.out.println("end third");
}
```

CHECKPOINT 3 – LAB COMPLETE – Remember to demonstrate the code trace you came up with for all 6 exercises in Part 3 to a TA to receive credit.