Activity 5: Testing with Assertions (Optional)

Introduction:

In the previous activities, you were asked to design and implement the `Card` and `Deck` classes. Upon completion of those tasks, you were instructed to test those classes by creating objects and testing each of the class' methods. In this activity, we will take a more formal approach to testing and introduce the Java `assert` statement.

Exploration:

What is the purpose of testing? Programmers make mistakes. These range from misunderstanding an algorithm or a problem specification (the most serious), to simple typing errors. The purpose of program testing is to find as many of those mistakes as possible. Let's consider some aspects of testing.

Efficient and organized testing

How should we test a program? Clearly, we need to run the program and see whether its behavior matches what is intended. There's more to it than that, though. Good testing is *systematic*. A programmer should pick test cases not at random but in a way more likely to find errors. For example, one should choose test runs that collectively exercise all parts of a program. Code that isn't executed may contain bugs. Good testing is also *programmer-efficient*. Tests should be easy to run. Test cases should be chosen to be as simple as possible while still being complex enough to expose errors. Simple test cases can also make it easier to see what a program is doing wrong.

For this activity, we will focus on finding two kinds of errors:

- Inconsistencies, where two parts of the program have different expectations about a variable's value, for example; and
- Common "typos of the brain," such as off-by-one errors and substitution of one operator for another (such as using `>` instead of `<`).

Our tests will all involve *assertions*, `boolean` expressions that should be `true` if the program is running correctly. We will incorporate our tests in a "tester" class whose `main` method executes the tests.

Assertions in Java

Our testing code will use the Java `assert` statement. This statement has the following form:

```
assert booleanExpression : StringExpression;
```

If the value of *booleanExpression* is `true`, the program continues with the next statement. If the value of *booleanExpression* is `false`, the program throws an `AssertionError` exception and prints *StringExpression*. It also prints a *Stack Trace* (more on that later). Here's an example using `assert`.

```
Card c1 = new Card("ace", "hearts", 1);
Card c2 = new Card("ace", "hearts", 1);
assert c1.matches(c2) : "Duplicate cards do not match.";
```

This code creates two new `Card` objects, and then checks that they contain the same information. If `c1.matches(c2)` returns `true`, then execution continues with the next statement. However, if the program has an error which results in `c1.matches(c2)` returning `false`, an `AssertionError` is thrown and the message "Duplicate cards do not match." is output.

Assertions are *disabled* by default. To use them, the command-line option `-ea` (Enable Assertions) is used. For example, the following would run the main method in `CardTester` with assertions enabled:

```
java -ea CardTester
```

Organizing tests of the `Card` class

We move on to test the `Card` class. Cards have a constructor and several methods (`suit`, `rank`, `pointValue`, `matches`, and `toString`). Our tests must cover all of these methods.

First, we create a file named `CardTester.java`. This name was chosen to describe its purpose. Its `main` method will start by creating some `Card` objects that will be used to do the testing:

```
Card c1 = new Card("ace", "hearts", 1);
Card c2 = new Card("ace", "hearts", 1);
Card c3 = new Card("ace", "hearts", 2);
Card c4 = new Card("ace", "spades", 1);
Card c5 = new Card("king", "hearts", 1);
Card c6 = new Card("queen", "clubs", 3);
```

The first two cards are identical. Cards `c3`, `c4`, and `c5` each differ from `c1` in only one of the instance variable values. These “one difference” cards will help us find copy/paste errors; for example, if the body of `suit` was copied from `rank` and pasted without change. The last card is different from the others in all of the values.

We start by testing the `Card` accessor methods. These tests merely check, using cards with completely different information, that what's stored is what was provided in the constructor. Note the inclusion in the `String` message of information about which value was involved in each assertion.

```
assert c1.rank().equals("ace") : "Wrong rank: " + c1.rank();
assert c1.suit().equals("hearts") : "Wrong suit: " + c1.suit();
assert c1.pointValue() == 1 : "Wrong point value: "
   + c1.pointValue();
assert c6.rank().equals("queen") : "Wrong rank: " + c6.rank();
assert c6.suit().equals("clubs") : "Wrong suit: " + c6.suit();
assert c6.pointValue() == 3: "Wrong point value : "
   + c6.pointValue();
```

Next, we test the `Card` method `matches`. Two cards match if and only if they have the same rank, suit, and point values. A likely implementation of `matches` will involve some comparisons and some uses of `&&`. Common bugs are the copy/paste error mentioned above and the substitution of `||` for `&&`. Comparing `c1` to all the others should reveal these kinds of errors.

```
assert c1.matches(c1) : "Card doesn't match itself: " + c1;
assert c1.matches(c2) : "Duplicate cards aren't equal: " + c1;
assert !c1.matches(c3)
   : "Different cards are equal: " + c1 + ", " + c3;
assert !c1.matches(c4)
   : "Different cards are equal: " + c1 + ", " + c4;
assert !c1.matches(c5)
   : "Different cards are equal: " + c1 + ", " + c5;
assert !c1.matches(c6)
   : "Different cards are equal: " + c1 + ", " + c6;
```

Finally, we test `toString`, again on two completely different objects.

```
assert c1.toString().equals("ace of hearts (point value = 1)")
   : "Wrong toString: " + c1;
assert c6.toString().equals("queen of clubs (point value = 3)")
   : "Wrong toString: " + c6;
```

If all of the tests pass, we provide a message that says so:

```
System.out.println("All tests passed!");
```

Systematic testing

Cards didn't involve any data structures more complicated than strings. When testing a class with more complex structures, it makes sense to start small. With a `Deck` class, for example, it might make sense to first provide tests that use a 1-card deck, and then a 2-card deck with different cards.

But, would all these tests get out of control? Just as in other programming you've done, it makes sense to split a long sequence of statements into "helper" methods. The result may be a smaller test program, and some of the assertion sequences might be easier to reuse. The `main` method in a `DeckTester` class might be the following:

```
public static void main(String[] args) {
   test1CardDeck();
   test2CardDeck();
   testShuffle();
   System.out.println("All tests passed!");
}
```

Exercises:

1. The folder **Activity5 Starter Code** contains the four subfolders **Buggy1**, **Buggy2**, **Buggy3**, and **Buggy4**. Each of these contains a different buggy version of the `Deck` class. These buggy decks have been precompiled; only the `Deck.class` bytecode file is included. These buggy versions each contain one error caused by either moving a statement, or substituting one symbol for another, e.g., `1` for `0` or `>` for `<`. Test each of them with the `DeckTester` application provided in each folder:

   ```
   java -ea DeckTester
   ```

   If you are using a Windows-based system, you can just execute the provided **DeckTester.bat** file.

   Each of the four different `DeckTester` runs will produce an `AssertionError` exception, along with information about why the error occurred. For each error that occurs, write down which method or constructor of the buggy `Deck` class could contain the bug, and make an educated guess about the cause of the error. You might find it helpful to refer to your completed `Deck` class from Activity 4.

   Note: The Buggy1 test will produce output similar to the following:

   ```
   ... >java -ea DeckTester
   Exception in thread "main" java.lang.AssertionError: isEmpty is
   false for an empty deck.
      at DeckTester.testEmpty(DeckTester.java:98)
      at DeckTester.test1CardDeck(DeckTester.java:28)
      at DeckTester.main(DeckTester.java:12)
   ```

The last three lines of output are a stack trace that tells you that the

- `AssertionError` occurred in the `testEmpty` method at line 98.
- `testEmpty` method was called from the `test1CardDeck` method at line 28.
- `test1CardDeck` method was called from the `main` method at line 12.

Record your conclusions below:

Buggy1:

Constructor or Method (write method name):

Describe a Possible Code Error:

_____

_____

_____

Buggy2:

Constructor or Method (write method name):

Describe a Possible Code Error:

_____

_____

_____

Buggy3:

Constructor or Method (write method name):

Describe a Possible Code Error:

_____

_____

_____

Buggy4:

Constructor or Method (write method name):

Describe a Possible Code Error:

_____

_____

_____

2. Now, examine the Buggy5 folder. This folder contains a `Deck.java` file with multiple errors. Use `DeckTester` to help you find the errors. Correct each error until the `Deck` class has passed all of its tests.

   Note that you may receive a runtime error other than `AssertionError` when running `DeckTester`. If so, you may find it helpful to switch the order of 1-card deck and 2-card deck tests as follows:

```
public static void main(String[] args) {
   test2CardDeck();   // order swapped
   test1CardDeck();   // order swapped
   testShuffle();
   System.out.println("All tests passed!");
}
```