SENG275 – Lab 7
Due Sunday 25 Jun, 11.59 pm

Technical note – you now have a content section in Brightspace named lab07.

Welcome to Lab 7. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked.

Quick summary of what you need to do:

1. Use Design by Contract principles to enforce the preconditions, postconditions and invariants in the BankAccount class.
2. Write property tests for the Colours, LeapYear and Palindrome classes.

   You're done!

Overview – Design by Contract

Every method in a Java program makes certain promises to the code that calls it and makes certain assumptions about the arguments that it will be given.

Before assertions, we usually wrote comments that specified how our method should be called, and what rules our return values might follow. Assertions allow us to specify these assumptions and guarantees directly in code.

These defensive programming techniques implement preconditions, post-conditions and invariants. They don't replace regular testing, but if used rigourously they add to our confidence that our code is properly implemented and will work well with other modules in our program.

Assertion statements

Form:

assert *statement: value*

statement is a any java statement that will evaluate to a true or false value. Note the lack of parentheses – assert is not a method. v*alue* is some value that is passed along to the *AssertionError* raised by this statement if it's false – this is usually a string. For example:

// x must be 0 or greater

assert x>=0 : "x was less than zero"

Example

```
/* AlphabeticalString
 * All characters in an AlphabeticalString are guaranteed to be in
 * alphabetical sorted order.  Create one by passing any valid string.
 * 'Alphabetical' here means sorted in ASCII order: 'ABa' is sorted,
 * 'aB' is not.
 * Cannot be null - throws a NullPointerException if null is passed in.
 */


public class AlphabeticalString {
  private String s;
```

```
  public AlphabeticalString(String newString) {
    Objects.requireNonNull(newString);
    this.s = newString;
    this.sort();
    assert this.s.length() == newString.length();
  }

  private void sort() {
    var ch = this.s.toCharArray();
    Arrays.sort(ch);
    this.s = String.valueOf(ch);
  }

  public void toUpper() {
    this.s = this.s.toUpperCase();
  }

  public String toString() {
    return this.s;
  }
}
```

As the comment states, the idea is that all strings of type 'AlphabeticalString' are always sorted alphabetically. If one is created by passing an unsorted string, the resulting AlphabeticalString object will be sorted. Furthermore, these strings cannot be null.

So far, these requirements and guarantees are implicit – we'd like them to be implemented in code. First, let's add a precondition to the constructor, to make sure that null hasn't been passed in:

```
public AlphabeticalString(String newString) {
  Objects.requireNonNull(newString);
  this.s = newString;
  this.sort();
}
```

Now if anyone tries to pass in a null, they'll get a NullPointerException. This exception forms part of the *contract* of our class – it tells anyone using our class what they can expect if they call our constructor.

Let's write a postcondition for the constructor as well. We expect that the sorted string that is created will have the same length as the provided (possibly unsorted) string. Let's check:

```
public AlphabeticalString(String newString) {
  Objects.requireNonNull(newString);
  this.s = newString;
```

```
    this.sort();
    assert this.s.length() == newString.length();
}
```

Note that the precondition uses an exception, while the postcondition uses an assertion. This is typical in Java – preconditions of public methods are part of their contract, so we try to throw exceptions so that the caller will receive a more specialized exception rather than the generic AssertionError. In addition, we don't want to remove our argument-checking code when we ship the final product – assert statements will be removed when we build for release. Post-conditions, however, are part of the implementation, and will be disabled in the final build, so it's fine to use the assert statement for them.

We therefore have a heuristic for deciding whether to use exceptions or assertions – if we're writing a *precondition* in a *public method*, use an exception. Everywhere else, use an assertion.

We might also want to ensure that our arguments remain unchanged by our method – if someone passes a reference to an object into our method, they won't be expecting that object to change unless that change is part of the contract. We can test the argument against a copy for equality, or we can let the compiler ensure immutability by marking the argument final. In our example, these aren't necessary, as Strings are immutable in Java anyway.

We'd also like to ensure that strings of this type are *always* alphabetical – we believe they're created alphabetical, but we'd like to be sure, and we'd like to make sure that other operations we perform on them won't change their sorted status. We need a class *invariant* – something that's *always* true (or something's gone horribly wrong).

```
public boolean invariant() {
    if (this.s == null) return false;
    if (this.s.length() < 2) return true;
    var ch = this.s.toCharArray();
    for (int i = 0; i < ch.length - 1; i++) {
        if (ch[i] > ch[i+1]) return false;
    }
    return true;
}
```

We can start asserting invariant() wherever we think the state of our object might have changed – for example, after we modify the string in toUpper():

```
public void toUpper() {
    this.s = this.s.toUpperCase();
    assert this.invariant() : "Invariant fails in toUpper()";
}
```

Exercise

Use your work with AlphabeticalString to add preconditions, postconditions and invariants to the BankAccount class.

1. Read over the class. What must always be true at any time for the bank account object to be valid? Use an invariant to enforce this.
2. Check each method. What assumptions does each method make about its arguments, and how could you use a precondition to enforce those assumptions? Should you use an exception or an assertion?

Property Testing with Jqwik

Until now, we've been mostly doing *example-based testing* – we've been trying to think up example input to our code under test, and hopefully coming up with good edge cases. It might be better if we could generate large numbers of test inputs randomly – we'd be more likely to test scenarios that might otherwise escape our notice. To do so, we can try property testing – instead of testing particular inputs we come up with in advance, we test code against entire *categories* of input. For example, instead of trying to come up with integers to test a method with, we might try testing a thousand integers, randomly selected from all the integers the computer can represent.

@Property

We'll use Jqwik to handle our property testing. Tests look similar to Junit parameterized tests – they have the annotation @Property, and they take arguments, just like a parameterized test. We annotate not only the test method, but each argument as well – we specify how much freedom jqwik will have to randomly choose actual argument values.

For example, if I want to test a method that takes two integers, my test signature might look like this:

```
@Property
public void test(@ForAll int x, @ForAll int y) {assertTrue(someMethod(x, y));}
```

Jqwik will select random integers for x and y, 1000 times (by default) and call someMethod, passing those arguments and asserting that true is returned.

We can further constrain the generation of these values. For example, '@ForAll @Positive int x' gives us only positive integers, and '@ForAll @StringLength(min = 5, max = 7) String s' will set s to 1000 random strings of length 5 to 7.

For more examples of random argument constraints, check the section 'Constraining Default Generation' at the Jqwik documentation page.

@Provide

Sometimes the constraints provided by jqwik aren't specific enough. For example, we may wish to test *only* integers that are evenly divisible by three. We can write a @Provide method that returns arbitrary numbers that match this criteria:

```
@Provide
Arbitrary<Integer> multiplesOf3() { return Arbitraries.integers().filter(n -> n % 3 == 0); }
```

Once we have our arbitrary number provider, we can specify it as a source for our property test:

```
@Property
public void someTest(@ForAll("multiplesOf3") int x) {...}
```

This will generate a thousand random numbers each divisible by three, and run a test for each.

The best way to see what sorts of arbitraries are possible is to look up the basic type you're working with in the left column of the Arbitraries Class Summary, and click on the *type*Arbitrary corresponding to that type. For example, if I want to test my method with a randomly-chosen whitespace character, I'll look at CharacterArbitrary, and use:

return Arbitraries.chars().whitespace();

Example:

The TwoIntegers class provides a static method that adds two integers, each between 1 and 99 (inclusive) and returns their sum. It throws an IllegalArgumentException if either argument is outside this range.

```
class TwoIntegers {

  /*
   * Given two integers, each between 1 and 99 inclusive, returns their sum.
   */

  public static int sum(final int n, final int m) {
    if (n < 1 || n > 99 || m < 1 || m > 99) {
      throw new IllegalArgumentException();
    }
    int value = m + n;
    return value;
```

```
  }
}
```

We'd like to test all the values that fall within the valid range for both arguments. We create a test using Jqwik's @Property annotation, and we specify that this test has two variables (which for convenience we'll name n and m, although they don't need to be named the same as the private variables in the method we're testing. We use @IntRange to specify the acceptable ranges for the two variables, and assert that the method returns the expected value.

```
@Property
void pass(@ForAll @IntRange(min = 1, max = 99) int n, @ForAll @IntRange(min = 1, max = 99) int m) {
  assertThat(TwoIntegers.sum(n, m)).isEqualTo(n + m);
}
```

We can do the same to test what happens if m is valid, but n is too low. Note that the expected behaviour is a thrown IllegalArgumentException, which we specify using assertThrows, and that we need to use a lambda expression to actually call the method we're testing.

```
@Property
void invalidN(@ForAll @IntRange(min = Integer.MIN_VALUE, max = 0) int n, @ForAll @IntRange(min = 1,
max = 99) int m) {
  assertThrows(IllegalArgumentException.class, () -> TwoIntegers.sum(n, m));
}
```

We could continue doing this, and complete the tests, but testing m and n with invalid values that are either too high or too low will involve a lot of test writing. It would be nice if we could call a method that gives us some arbitrary number outside the valid range, and use that in our call to sum(). We can do this with the @Provide annotation, and by specifying that we want one number from one of two ranges – either a number 0 or less, or a number 100 or greater.

```
@Provide
private Arbitrary<Integer> invalidRange() {
  return Arbitraries.oneOf(Arbitraries.integers().lessOrEqual(0), Arbitraries.integers().greaterOrEqual(100));
}
```

Every time we call this method, we'll get one (thanks to the oneOf method) invalid number (thanks to the methods we call on the number returned by the call to Arbitraries.integers(). We can then test our 'what if n is invalid and m is valid' scenario for for values of n that are *either* too high or too low, in one test:

```
@Property
void invalidN(@ForAll("invalidRange") int n, @ForAll @IntRange(min = 1, max = 99) int m) {
  assertThrows(IllegalArgumentException.class, () -> TwoIntegers.sum(n, m));
}
```

Exercises:

Write property tests for the Colours, LeapYear and Palindrome classes, using the TwoIntegers example as a guide. Write your tests in the provided classes in the test/java/lab07 folder.

1. Colours.rgbBytesToInt() takes three red, green and blue values as arguments, and produces a single integer with the three values 'packed' in rgb order. For example, red=255 (or 0xFF), green = 51 (or 0x33), blue = 17 (or 0x11) would be converted to 0xFF3311, or 16724753.
2. LeapYear.isLeapYear() will probably require you to @Provide some Arbitraries. To do this, you will probably want to return Arbitraries.integers(), but with a different subset of integers for each test. It's helpful to apply a filter – for example, Arbitraries.integers().filter(n -> n % 400 == 0) will provide only random numbers that are multiples of 400.
3. Palindrome.isPalindrome() might require the following annotations for arguments:

   @AlphaChars - ensures only characters from the alphabet
   @UniqueElements – ensures there are no duplicates in an array
   @Size(min = n) - ensures that an array has at least size n

   The method reverse() is provided for your work.