Chapter 3: Test automation

1. **Refer to Chapter 3, problem #6**

   Consider `PrimeNumbers.java` class (`PrimeNumbers.java` is attached to this assignment). This class has three methods as follows:

   - `computePrimes()` → takes one integer input and computes that many prime numbers
   - `iterator()` → returns an Iterator that will iterate through the primes
   - `toString()` → returns a string representation of primes.

   `computePrimes()` has a fault that causes it **not** to include prime numbers whose last digit is 9. For example, it omits `19, 29, 59, 79, 89, 109,…`

   Answer the following questions `a` through `d`, if the test could be created describe what the test case would be or explain why it cannot be created:

   a. A test that does not reach the fault

   for computePrimes(0), test that does not reach the fault as it does not enter the while loop.

   b. A test that reaches the fault, but does not infect

   computePrimes(4) reaches the fault state but do not infect as none of the numbers contains 9 digit at units place.

   c. A test that infects the state, but does not propagate

   All infections will propagate.

   d. A test that propagates, but does not reveal

   The following test demonstrates this:

   n >= 8

   PrimeNumbers primeNumbers = new PrimeNumbers();

   primeNumbers.computePrimes(8);

   assertEquals (2, primeNumbers.iterator().next());

   Though the error propagates, the failure is not revealed through the assertion.

   e. A test that reveals the fault – For this answer submit the code for a Junit test class `PrimeNumbersTest.java`, the class only needs to contain a single test case that can be run on the book code, that for this specific condition ("A test that reveals the fault")

```
computePrimes(8);
Expected = [2, 3, 5, 7, 11, 13, 17, 19]
Result = [2, 3, 5, 7, 11, 13, 17, 23]
```

```
class PrimeNumbersTest {
    PrimeNumbers p = new PrimeNumbers();
```

```
    @Test
    void Test5() {
        p.computePrimes(8);
        assertEquals("[2, 3, 5, 7, 11, 13, 17, 19]", p.toString());
    }
}
```

2. **Refer to Chapter 3, problem #9**

When overriding the `equals()` method, programmers are also required to override the `hashCode()` method; otherwise clients cannot store instances of these objects in common `Collection` structures such as `HashSet`. For example, the `Point` class from Chapter 1 is defective in this regard (`Point.java` is attached to this assignment).

Answer the following questions and make it easy to evaluate your work. Specifically, since you are evaluating different versions of the same code, provide evidence of execution (screenshots) at each phase, and briefly describe all of your work.

a) Demonstrate the problem with `Point` using a `HashSet`

```
Point a1 = new Point(5, 6);
Point a2 = new Point(5, 6);
Set<Point> s = new HashSet<Point>();
s.add(a1);
boolean b = s.contains(a2);
System.out.println(b);
```

b) Write down the mathematical relationship required between `equals()` and `hashCode()`

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. The inverse is not true: it is perfectly fine for unequal objects to share a hash code. You must override hashCode in every class that overrides equals.

c) Write a simple Junit test to show that `Point` objects do not enjoy this property

```
@Test
public void hashTest() {
    Point a1 = new Point(5,6);
    Point a2 = new Point(5,6);
    assertTrue(a1.hashCode() == a2.hashCode(),"Hash codes must match");
}
```

d) Repair the `Point` class to fix the fault

```
import java.util.HashSet;
import java.util.Set;

public class Point
```

```
{
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x=x;
        this.y=y;
    }

    @Override
    public int hashCode(){
        int result = 20;
        result = 11 * result + x;
        result = 11 * result + y;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        // Location A
        if (!(obj instanceof Point))
            return false;
        Point p = (Point) obj;
        return (p.x == this.x) && (p.y == this.y);
    }

    public static void main(String[] args) {
        Point a1 = new Point(5, 6);
        Point a2 = new Point(5, 6);
        Set<Point> s = new HashSet<Point>();
        s.add(a1);
        boolean b = s.contains(a2);
        System.out.println(b);
    }
}
```

3. **Refer to Chapter 3, problem #8**

Develop a set of data-driven JUnit tests for the `Min` program. These tests should be for normal, not exceptional, returns. Make your method produce both `String` and `Integer` values. Submit the code for a Junit test class.

```
@ParameterizedTest
@MethodSource("minValues")
public void minDataDrivenTest(List list, Object minElement){
   assertEquals(minElement, Min.min(list));
}

private static Collection<Object[]> minValues() {
   Object[][] data = new Object[][]{
         {new ArrayList (Arrays.asList (1,2)), 1},
         {new ArrayList (Arrays.asList (5, 4, 78, 99)), 4},
         {new ArrayList (Arrays.asList ("cat","bat","hat")), "bat"},
         {new ArrayList (Arrays.asList ("cat","dog")), "cat"}
   };
   return Arrays.asList(data);
}
```