WRITTEN TEST 2C

This is a 60 minute test. The test is closed book (no aids are allowed).

Written question instructions

- There are 8 short answer questions, each worth 2 marks.
- There are 3 questions that require students to explain their answers, each worth 8 marks.
- Answer the written questions in a text file named `answers.txt`—a suitable file should open in a text editor when the test starts. Feel free to use a different text editor if you wish.
- Make sure to save your work before submitting it! Every year a small number of students end up submitting an empty file because they forgot to save their work.
- You may submit your work as many times as you wish.
- A few minutes before the end of the test you will receive a warning that the test is ending soon. The message will repeat every minute until the end of the test. Use this time to submit your work; it is difficult to work effectively with the message popping up every minute.

Submission instructions

- Open a terminal (if one is not already open)
- Find the directory where you have saved your `answers.txt` file; the file should be in your home directory.
- Type the following command:

  ```
  submit 2030 test2C answers.txt
  ```

  and press enter.

1. [2 marks] What is a `public static final` field normally used for?

> **Solution:** To represent a constant value.

2. [2 marks] Consider the following class that represents a line segment connecting two points (its start point and its end point):

```
public final class LineSegment {
    private final Point2 start;
    private final Point2 end;

    public Line(Point2 p1, Point2 p2) {
        this.start = new Point2(p1);
        this.end = new Point2(p2);
    }

    // remainder of class not shown
}
```

Which statement best describes the class `LineSegment`? Give your answer as A, B, C, or D.

A. `LineSegment` is an aggregation of two points

B. `LineSegment` is a composition of two points

C. `LineSegment` is mutable

D. `LineSegment` is a superclass

> **Solution:** `B`

3. [2 marks] What must be true about the keys in a Java `Map`?

> **Solution:** The keys must be unique (no duplicated keys).

4. [2 marks] What is the definition of the term *privacy leak*?

> **Solution:** A privacy leak occurs when an object exposes a reference to a mutable field.

5. [2 marks] When a class has fields that are not of primitive type, we often need to consider using composition instead of aggregation. What is the main disadvantage of using composition instead of aggregation?

Solution: The memory and time needed to create the defensive copies.

6. [2 marks] What does the keyword `final` mean when it is used as a modifier on a method?

Solution: The method cannot be overridden.

7. [2 marks] Consider the following Java statement:

```
Counter c = new AscendingCounter();
```

Which class is the superclass and which class is the subclass? Write your answer like:

superclass:
subclass:

and complete each line with a class name.

Solution: superclass: Counter
subclass: AscendingCounter

8. [2 marks] Which features of the superclass does a subclass inherit?

Solution: All of the non-private fields and methods (but not the constructors).

9. Consider the following implementation of a class named `Counter`:

```java
// IMPLEMENTATION #1
public final class Counter {

       public static final Counter COUNTER = new Counter();

       private int value;

       private Counter() {
              this.value = 0;
       }

       public int value() {
              return this.value;
       }

       public void incr() {
              this.value++;
       }
}
```

Also consider a second alternative implementation of the `Counter` class:

```java
// IMPLEMENTATION #2
public final class Counter {

       private static final Counter COUNTER = new Counter(); // HERE

       private int value;

       private Counter() {
              this.value = 0;
       }

       public int value() {
              return this.value;
       }

       public void incr() {
              this.value++;
       }

       public static Counter getCounter() { // HERE
              return Counter.COUNTER;
       }
}
```

The lines marked with the comment `HERE` indicate the differences between the two implementations.

Both implementations are examples of a well known design pattern used in software engineering.

(a) [2 marks] Is the `Counter` class mutable or immutable? Justify your answer using a single sentence. [Note: The fact that both classes expose a reference to their static field does not determine whether the class is mutable or immutable in this case because the type of the field is the same as the class that it is defined in.]

Solution: `Counter` is mutable because there is a mutator method (`incr`).

(b) [2 marks] Both implementation #1 and implementation #2 ensure something about the number of `Counter` objects that can be created. What is it that both implementations ensure?

Solution: There is exactly one `Counter` object.

(c) [2 marks] Implementation #1 uses a `public` field to provide access to the `COUNTER` field. Implementation #2 uses a `private` field and a `public` method to provide access to the `COUNTER` field. Making fields `private` and providing access to the state of an object through a constant `public` interface is known as what principle?

Solution: Information hiding.

(d) [2 marks] What advantage does implementation #2 have compared to implementation #1? [Hint: Consider what happens if you decide that the property in part (b) is no longer needed]

Solution: If you decide that you do not need to enforce that there is only one `Counter` object then the method `getCounter` can be changed to return a referece to a new `Counter` without changing the API of the class.

10. Consider the following class that represents an integer number having up to `Integer.MAX_VALUE` digits:

```
/**
 * An integer value having up to as many digits as will fit in a list.
 * A BigInt object owns all of its digits.
 */
public class BigInt {
       private List<Digit> digits;

       public BigInt(List<Digit> d) {
              /* IMPLEMENTATION NOT SHOWN */
       }

       public int numDigits() {
              return this.digits.size();
       }

       public Digit getDigit(int index) {
              /* IMPLEMENTATION NOT SHOWN */
              /* RETURNS THE DIGIT AT THE SPECIFIED INDEX IN THE NUMBER */
       }

       public void reverse() {
              reverse(this.digits);
       }

       // RECURSIVELY REVERSES A LIST OF DIGITS BY SWAPPING THE FIRST
       // AND LAST DIGITS OF t
       private static void reverse(List<Digit> t) {
              /* BASE CASE(S) NOT SHOWN */

              String first = t.get(0);
              String last = t.get(t.size() - 1);
              t.set(0, last);
              t.set(t.size() - 1, first);
              reverse(t.subList(1, t.size()));
       }

}
```

(a) [2 marks] Suppose that you make a `BigInt` object using an existing list of digits `t`:

```
BigInt huge = new BigInt(t);
```

Without seeing the code in the constructor of `BigInt` how would you check if the constructor was setting the field `this.digits` to an alias of the parameter `d` or to a copy (either shallow or deep) of the parameter `d`? You can describe what you would do, or you could write 2 or 3 lines of Java code to show what you would do.

Solution: Change the size of the list `t` and then check if `huge.numDigits()` returns the original size of the list `t` or the new size of the list `t`; if it returns the original size then the constructor is making a copy otherwise the constructor is making an alias. For example:

```
t.clear();
System.out.println(huge.numDigits() == 0);
```

(b) [2 marks] Suppose that you make a `BigInt` object using an existing list of digits `t`:

```
BigInt huge = new BigInt(t);
```

(2 marks for a suitable explanation or sample of Java code)
Without seeing the code in the method `getDigit` how would you check if the method was returning an alias to a `Digit` in the number or a new copy of a `Digit` in the number? You can describe what you would do, or you could write 1 or 2 lines of Java code to show what you would do.

Solution: You can check if repeated calls to `getDigit` returns the same reference:

```
System.out.println(huge.getDigit(0) == huge.getDigit(0));
// if true then getDigit probably returns an alias
```

Alternatively, you could get one of the digits, modify it, and then get the digit again and check if the digit has changed:

```
Digit d = huge.getDigit(0);
if (d.getValue() != 1) {
       d.setValue(1);
}
else {
       d.setValue(2);
}
Digit e = huge.getDigit(0);
System.out.println(d.equals(e));
// if true then getDigit returns an alias
```

(2 marks for a suitable explanation or sample of Java code)

(c) [2 marks] What is a suitable base case (or base cases) for the recursive method `reverse`? The method should not throw an exception for any argument list `t`.

Solution: The base case occurs when `t.size()` equals 1.

```
if (t.size() == 1) {
      return;
}
```

`t.size()` equals 0 does not work as a base case because the recursive call will throw an exception if `t.size()` is 1.

(d) [2 marks] Besides the missing base case, there is one error in the recursive method `reverse`. What is the error?

Solution: The recursive call should use the sublist `t.subList(1, t.size() - 1)` because the first and last digits have already been swapped.

11. Consider the following two classes related by inheritance:

```
public class Lock {

        private boolean isLocked;

        public Lock(boolean isLocked) {
                if (isLocked) {
                        this.lock();
                } else {
                        this.unlock();
                }
        }

        public Lock(Lock other) {
                this.isLocked = other.isLocked;
        }

        public void lock() {
                this.isLocked = true;
        }

        protected void unlock() {
                this.isLocked = false;
        }

}

public class CombinationLock extends Lock {

        private Combination combo;


        public CombinationLock(Combination combo) {
                [PART (a)]
                this.combo = new Combination(combo);
        }

        public CombinationLock(CombinationLock other) {
                [PART (b)]
        }

        @Override
        protected void lock() {
                // NOT SHOWN; LOCKS THE LOCK
                this.combo.shuffle(); // randomly shuffles the dials on the lock
        }

        public void unlock(Combination tryMe) {
                if (this.combo.equals(tryMe)) {
                        [PART (c)]
                }
        }

}
```

(a) [2 marks] What single line of Java would you write on the line labelled `[PART (a)]` to complete the `CombinationLock` constructor? Why is the line required?

Solution: Either `super(true)` or `super(false)` is required because the superclass constructor must be called.

(b) [2 marks] What two lines of Java would you write to replace the line labelled `[PART (b)]` to complete the copy constructor of `CombinationLock`?

Solution: Only one solution is possible:

```
super(other); // copies the state of the lock (locked or unlocked)
this.combo = new Combination(other.combo); // copies the combination
```

There is no other way to correctly copy the state of the lock because `isLocked` is private in `Lock`.

(c) [2 marks] The `unlock` method in `CombinationLock` needs to set the state of the lock to unlocked on the line labelled `[PART (c)]`. What single line of Java would you write to replace the line labelled `[PART (c)]` to complete the method?

Solution: `super.unlock();` or `this.unlock();` are both acceptable.

(d) [2 marks] What sequence of events occurs when a user of the two classes makes a new `CombinationLock`? Is the `CombinationLock` always successfully initialized?

Solution: The `Lock` constructor is called which might call the method `lock` which is overridden by `CombinationLock`. If this happens then the overridden version of `lock` runs which will cause a `NullPointerException` because `this.combo` is `null` when `lock` is running. Therefore, the `CombinationLock` might not be successfully initialized.

(2 marks for stating the overridden version of `lock()` causes an exception to be thrown when it is called from the `Lock` constructor.)