CSE 331 Summer 2016 Final Exam

Name ____

The exam is closed book, closed notes, and closed electronics.

Please **wait to turn the page** until everyone is told to begin.

Score _______________ / 54

1. ______ / 12

2. ______ / 12

3. ______ / 10

4. ______ / 10

5. ______ / 10

Bonus:

1. ______ / 6

2. ______ / 4

Question 1. Circle the correct answer for each question below.

a. If done correctly, which of the following can rule out any possibility of bugs in a complex method:

Type Checking $\qquad$ $\boxed{\text{Reasoning}}$ $\qquad$ Testing

b. If you believe your reasoning is correct, it is not necessary to write tests.

True $\qquad$ $\boxed{\text{False}}$

c. If you believe your reasoning is correct, it is not necessary to use any runtime assertions.

True $\qquad$ $\boxed{\text{False}}$

d. Which of the following is NOT a benefit of writing method specifications?

can prove correctness $\qquad$ can write tests

code is more readable $\qquad$ $\boxed{\text{code is more efficient}}$

e. Which of the following is NOT a benefit of crashing immediately upon discovery of a bug in the program?

easier debugging $\qquad$ $\boxed{\text{bug is hidden from the user}}$

limits further damage $\qquad$ bug is less likely to go undetected

f. Which of the following is NOT necessary to prove a loop correct?

$\boxed{\text{show precondition holds}}$ $\qquad$ show invariant and termination condition imply the postcondition

show invariant holds initially $\qquad$ show invariant holds after the loop body

Question 2. For each question below, write a short answer (1-2 sentences).

a. If you were redesigning the Java libraries from scratch, would you have **`NullPointerException`** be a checked or unchecked exception?

Unchecked. If it were checked, every method that used a reference variable (i.e., most of them) would have to declare that it could throw one, which would be extremely laborious and would provide little benefit.

b. The following code does not compile:

```
class Foo<T> {
  public void foo() {
    T[] arr = new T[10];  // compiler error
    ...
  }
}
```

What do you think the author should do instead?

Either allocate a new **`Object[10]`** and cast it to **`T[]`** or else just use an **`ArrayList<T>`** instead.

c. Consider the following method specification:

```
/** @returns the sum of a and b */
Double sum(Number a, Number b);
```

Describe two different ways to weaken the specification.

1. Change the return type to **`Number`**.
2. Change either argument type to **`Double`**.

Question 3. Consider the following methods operating on lists:

```
// Returns the sum of the given numbers.
double sumNumbers(Iterator<Number> iter) {
  double s = 0;
  while (iter.hasNext())
    s += iter.next().doubleValue();
  return s;
}

// Returns the sum of the given doubles.
double sumDoubles(Iterator<Double> iter) {
  double s = 0;
  while (iter.hasNext())
    s += iter.next().doubleValue();
  return s;
}
```

Suppose that we also have the following list variables:

```
List<Number> numList = new ArrayList<Number>();
numList.add(1.0);
numList.add(2.0);
numList.add(3.0);

List<Double> dblList = new ArrayList<Double>();
dblList.add(1.0);
dblList.add(2.0);
dblList.add(3.0);
```

a. **Circle** those of the following lines of code that have a compiler error:

```
s = sumNumbers(numList.iterator());

s = sumNumbers(dblList.iterator());

s = sumDoubles(numList.iterator());

s = sumDoubles(dblList.iterator());
```

b. One line above that has a compiler error can be fixed by introducing an **adapter** that wraps the iterator currently used in the code and adapts it to fit the interface needed by the method being called. To use it, the line above would be changed to look like this:

```
sum??(new IterAdapter(??List.iterator()));
```

Write an implementation of **`IterAdapter`** that will make that **one line** compile and run correctly when changed as just shown.

```
/** Converts Iterator<Double> to Iterator<Number>. */
public class IterAdapter implements Iterator<Number> {
  private Iterator<Double> iter;

  /** Creates a wrapper on the given iterator. */
  public IterAdapter(Iterator<Double> iter) {
    this.iter = iter;
  }

  @Override public boolean hasNext() {
    return iter.hasNext();
  }

  @Override public Number next() {
    return iter.next();
  }
}
```

This could then be used above as:

```
sumNumbers(new IterAdapter(dblList.iterator()));
```

c. If you were designing Java from scratch, would you want programmers to have to write the code above? Explain.

No. The adapter above doesn't actually do anything! It just takes the outputs from the iterator and returns them. The only purpose of this adapter is to fix a weakness in the type checker. The compiler should just figure this out on its own.

The next two problems have the same format. Each shows you some code that has a significant bug. Then its ask you to:

(1) explain where the proof of correctness would break down and
(2) describe a test that would have caught the bug.

Here is an example of what we are looking for…

Consider the following code, which has a significant bug:

```
/** @returns (degFahr – 32) * 5 / 9;
public static int fahrenheitToCelcius(int degFahr) {
  int x = 5 * degFahr;
  int y = x / 9;
  int degCelcius = y - 32;
  return degCelcius;
}
```

a. Where does the proof of correctness for this code fail?

   Forward reasoning tells us that degCelcius = degFahr * 5 / 9 – 32. Returning this fails to match the postcondition. The latter wants the number (degFahr – 32) * 5 / 9, which generally is not the same.

b. Describe a test (in English or JUnit code) that would have caught the bug.

   ```
   assertEquals(0, farhenheitToCelcius(32));
   ```

   or

   Calling fahrenheitToCelciums with input 32 would return the answer 32 * 5 / 9 – 32, which is not zero, whereas the correct answer is (32 – 32) * 5 / 9 = 0.

Question 4. Consider the following code, which has a significant bug:

```
1   /** @returns the greatest common denominator of m & n */
2   public static int gcd(int m, int n) {
3     if (m >= n)
4       return gcdHelper(m, n);
5     else
6       return gcdHelper(m, n);
7   }
8
9   /** @returns the gcd of m and n if m >= n
10    * @throws IllegalArgumentException if m < n */
11  public static int gcdHelper(int m, int n) { ... }
```

Hint: you don't need to know what the greatest common denominator (gcd) is to solve this problem.

c. Where does the proof of correctness for this code fail?

Forward reasoning tells us that m < n before line 6. As a result, the precondition of `gcdHelper` does not hold, so we cannot infer the postcondition we need after the call.

d. Describe a test (in English or JUnit code) that would have caught the bug.

```
assertEquals(1, gcd(1, 2)); // fails due to exception
```

Question 5. Consider the following code, which has a significant bug:

```
/** @requires max >= 2
  * @returns the largest prime not bigger than max */
public static int getLargestPrime(int max) {
  int lastPrime = 2;
  int n = 2;
  // Inv: lastPrime is largest prime not bigger than n
  while (n != max) {
    if (isPrime(n))
      lastPrime = n;
    n += 1;
  }
  return lastPrime;
}

/** @returns true iff the number is prime */
public static boolean isPrime(int n) { ... }
```

a. Where does the proof of correctness for this code fail?

   Forward reasoning inside the loop body (starting from the loop invariant) tells us that `lastPrime` holds the largest prime not bigger than `n - 1`, but that is not the same as Inv, so we can't conclude that the loop invariant holds after the loop body.

b. Describe a test (in English or JUnit code) that would have caught the bug.

   ```
   assertEquals(3, getLargetPrime(3));  // fails
   ```

Bonus Question 1. Consider the following code:

```
public class IntList {
  ...

  /** @modifies this
    * @effects Removes all entries in the list, from the
    *   given index to the end, with the given value.
    *   E.g., on [1, 2, 3, 2, 5, 2], removeFrom(3, 2)
    *   would change the list to [1, 2, 3, 5] */
  public void removeFrom(int index, int value);

  /** @modifies this
    * @effects Removes all entries after the first
    *   occurrence of the given value appearing in the
    *   list after the given index. E.g., on
    *   [1, 2, 3, 2, 5, 2], removeAfter(3, 2) would
    *   change the list to [1, 2, 3, 2]. */
  public void removeAfter(int value, int index);

  ...
}
```

There are *at least three* different ways in which this code is worrisome — ways in which it is likely to lead to bugs in the client code. Describe **two** of them.

1. The method `removeFrom` has two arguments of the same type, so clients could easily mix up the arguments and see no compiler error. (The same issue exists with `removeAfter`.)

2. The two methods `removeFrom` and `removeAfter` are inconsistent in the order of the two arguments (`value` and `index`), which makes a mistake by the client even more likely.

3. The two methods `removeFrom` and `removeAfter` have very similar names. The names do not make clear enough which is which. It would be easy for the client to mix up these methods.

The descriptions of these methods are also hard to follow, which is not good.

The above class is a wonderful example of how **not** to write code.

Bonus Question 2. Consider the following code:

```
/** Maintains a map built from a list of (key,value)
  * pairs read one-at-a-time (i.e., from a "stream"). */
public class MapStream<K,V> {
  private Map<K,V> map = new HashMap<K,V>();

  /** @returns the value (if any) for the given key */
  public V get(K key) { return map.get(key); }

  /** @effects Adds the next (key,value) from the stream.
    * @returns the key from the next pair */
  public K next() {
    K key = nextKey();
    V val = nextValue();
    map.put(key, val);
    return key;
  }

  /** @returns key from the next (key,value) pair. */
  protected abstract K nextKey();

  /** @returns value from the next (key,value) pair. */
  protected abstract V nextValue();
}

/** Maintains a map of pairs (n, p) where p is the n-th
  * prime number. These are added in order by n. */
public class PrimeStream
    extends MapStream<Integer,Integer> {
  private int n = 0;
  private int lastPrime = 1;

  @Override protected Integer nextKey() {
    n += 1;
    lastPrime += 1;
    while (!isPrime(lastPrime))
      lastPrime += 1;
    return n;
  }

  @Override protected Integer nextValue() {
    return lastPrime;
  }
}
```

This above code works correctly, but one aspect is extremely worrisome, particularly if the superclass and subclass were written by different people.

Describe why it would be easy for the author of the superclass to break the code in the subclass **without realizing it**. (This should only take a few sentences.)

The subclass only works correctly because `nextKey` is called before `nextValue`. If the superclass were to call these in the opposite order, the subclass code would break.

It is unlikely that the author of the superclass realized that the client would be written in such a way as to become dependent on the order of these two calls, so they might change the order for some reason in the future and not realize that the change would break some subclasses.