

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

CSE 331 Summer 2016 Final Exam

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

Reasoning

Testing

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

True

False

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

True

False

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

can prove correctness

can write tests

code is more readable

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

bug is hidden from the user

limits further damage

bug is less likely to go undetected

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

show precondition holds

show invariant and termination condition
imply the postcondition

show invariant holds initially

show invariant holds after the loop body

CSE 331 Summer 2016 Final Exam

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?

- 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?

- 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.

CSE 331 Summer 2016 Final Exam

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());
```

CSE 331 Summer 2016 Final Exam

- 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.

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

CSE 331 Summer 2016 Final Exam

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:

```
1  /** @returns (degFahr - 32) * 5 / 9;
2  public static int fahrenheitToCelcius(int degFahr) {
3      int x = 5 * degFahr;
4      int y = x / 9;
5      int degCelcius = y - 32;
6      return degCelcius;
7  }
```

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

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

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

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

or

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

CSE 331 Summer 2016 Final Exam

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?

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

CSE 331 Summer 2016 Final Exam

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

```
1  /** @requires max >= 2
2   * @returns the largest prime not bigger than max */
3  public static int getLargestPrime(int max) {
4   int lastPrime = 2;
5   int n = 2;
6   // Inv: lastPrime is largest prime not bigger than n
7   while (n != max) {
8     if (isPrime(n))
9       lastPrime = n;
10    n += 1;
11  }
12  return lastPrime;
13 }
14
15 /** @returns true iff the number is prime */
16 public static boolean isPrime(int n) { ... }
```

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

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

CSE 331 Summer 2016 Final Exam

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.

2.

CSE 331 Summer 2016 Final Exam

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;
    }
}
```

CSE 331 Summer 2016 Final Exam

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.)