

(True/False 1 point each)

In general, code with a stronger specification is harder to implement but easier to use.

TRUE FALSE

Methods should always check their preconditions (`@requires`) to ensure arguments are valid.

TRUE FALSE

Java subtypes are always true subtypes.

TRUE FALSE

If A is a true subtype of B, then an instance of A can be safely substituted in for an of B.

TRUE FALSE

The Java type system ensures that objects only refer to fields and call methods that are defined.

TRUE FALSE

The Java type system ensures that all exceptions are either caught or declared in a method's throws clause.

TRUE FALSE

Array subtyping in Java is covariant.

TRUE FALSE

Debugging is easier than writing code, so it is OK to be as clever as possible when developing your program.

TRUE FALSE

Sometimes you should call `paintComponent()` in a Java GUI program

TRUE FALSE

Sometimes you should call `repaint()` in a Java GUI program

TRUE FALSE

(12 points) Given the following class hierarchy

```
class Pet
class Cat extends Pet
class Dog extends Pet
class Husky extends Dog implements Mascot
```

and the following

```
Object o; Pet p; Cat c; Dog d; Husky h; Mascot m;
List <? extends Pet> lep;
List <? extends Cat> lec;
List <? super Husky> lsh;
```

For each of the following, circle OK if the statement has correct Java types and will compile or circle ERROR if there is some type of error.

OK ERROR `lsh.add(h);`

OK ERROR `lep.add(o);`

OK ERROR `lep.add(d);`

OK ERROR `lec.add(p);`

OK ERROR `lsh.add(o);`

OK ERROR `lec.add(null);`

OK ERROR `d = lsh.get(1);`

OK ERROR `d = lec.get(1);`

OK ERROR `d = lep.get(1);`

OK ERROR `o = lec.get(1);`

OK ERROR `p = lsh.get(1);`

OK ERROR `p = null.get(1);`

Using forward reasoning, write an assertion in each blank space indicating what is known about the program state at that point, given the precondition and the previously executed statements. Your final answers should be simplified. Be as specific as possible, but be sure to retain all relevant information. Assume all variables are ints.

(a) (5 points)

{ $y > 0$ }

$x = 3;$

$y = x + y;$

$z = x * y;$

(b) (5 points)

(b) (5 points)

{ $|x| = 42$ }

$x = x / 2;$

$x = x + 3;$

Question 2. (6 pts) Compute the weakest precondition with respect to the given postcondition using backwards reasoning. Show each intermediate condition.

Assume y and z are ints.

```
if (y == 0) {
```

```
    z = 4;
}
```

```
else {
```

```
    z = y+2;
}
```

```
{ z >= 4 || z <= -4 }
```

(10 pts)

Assume the following code compiles:

```
A a = new A();
B b = new C(a);
D d = b;
```

Mark all of the following that must be true:

- a) A is a supertype of B.
- b) A is a subtype of B.
- c) B is a supertype of C.
- d) B is a subtype of C.
- e) B is a subtype of D.
- f) B is a supertype of D.
- g) C is a supertype of D.
- h) C is a subtype of D.
- i) A is a supertype of D.
- j) A is a subtype of D.

(20 points) Consider the following specification, along with Java code that implements it.

```
// Computes x*n.
```

```

// @requires:  n <= 0
// @returns: x*n
int mult(int x, int n) {
    int result = 0;
    while (n < 0) {
        result = result - x;
        n = n + 1;
    }
    return result;
}
// postcondition: result == x*n_orig (n_orig is the input value of n)

```

- a) Given the loop invariant, $(x * n_orig == result + x * n) \ \&\& \ (n \leq 0)$ (n_orig is the input n) Show that the LI hold before the first iteration of the loop. (3 points)

- b) Prove correctness of this code, i.e. show by induction that the LI holds before and after each iteration. (8 points)

- c) Prove that the loop terminates by finding a decrementing function and showing that when it reaches a minimum the loop has terminated. (4 points)

- d) Show that at return, given that the LI is correct, the LI and the loop exit condition implies the postcondition. (5 points)

(8 points) Consider the following code:

```
public class ObnoxiousFinalExamQuestion {
    private List<String> names;
    private final List<String> nickNames;
    private final List<JButton> buttons;
    ...
    public List<String> getNames()
        { return names; }
    public List<String> getNickNames()
        { return nickNames; }
    public String getFirstName()
        { return names.get(0); }
    public JButton getFirstButton()
        { return buttons.get(0); }
}
```

Assume that all of the lists are defined after an instance of the class is constructed and that each list contains at least one item.

Circle the methods that could cause a representation exposure.

(25 points) Consider the following partially implemented class. Menu items are Strings like “pizza” or “hamburger”. Prices are stored as Numbers as values for menu items in a HashMap. This restaurant doesn’t give things away or pay you to eat, so all prices are > 0. The restaurant doesn’t allow null items in their menu.

```
public class Menu {
    // menu data (instance variable)
    private HashMap<String, Number> items;

    /** construct empty Menu */
    public Menu() {
        items = new HashMap<String, Number>();
    }

    /** store item in menu with given price */
    public void addItem(String name, double price) {
        items.put(name, price);
    }

    /**
     * Return true if item is included in this menu */
    public boolean contains(String item) {
        return items.get(item) != null;
    }

    /** Return the price of the named item */
    public double getPrice(String item) {
        ... // implementation omitted
    }
}
```

```

    }

    /** Return the total price of all of the items in an order
        (a list of item names), assuming that all of the items
        in the order list actually appear in the menu.
    */
    public double getOrderPrice(List<String> order) {
        ... // implementation omitted
    }
}

```

Reference information about maps

If *m* is variable of type `HashMap<K,V>`, where *K* is the key type and *V* is the value type, the following methods are available.

<code>m.containsKey(k)</code>	return true if <i>k</i> is a key in map <i>m</i>
<code>m.containsValue(v)</code>	return true if one or more keys in <i>m</i> map to the value <i>v</i>
<code>m.get(k)</code>	return value associated with <i>k</i> , or null if <i>k</i> is not a key in <i>m</i>
<code>m.isEmpty()</code>	return true if <i>m</i> contains no key-value mappings
<code>m.keySet()</code>	return a <code>Set<K></code> view of the keys in <i>m</i>
<code>m.put(k,v)</code>	store value <i>v</i> with key <i>k</i> in <i>m</i> ; return the previous value associated with <i>k</i> or null if <i>k</i> was not a key in <i>m</i>
<code>m.remove(k)</code>	remove any key-value mapping with key <i>k</i> from <i>m</i>
<code>m.size()</code>	return the number of key-value mappings in <i>m</i>
<code>m.values()</code>	return a <code>Collection<V></code> view of the values in <i>m</i>

a) (10 points) Give a suitable representation invariant (RI) and abstraction function (AF) for the `Menu` class. The RI should state when the data is valid. The AF should give the meaning of a valid `Menu` object. Prices cannot be less than or equal to zero.

b) (15 points) `getOrderPrice()` is supposed to take a list of strings that represent an order and return the total price of the order. For example, if the menu contains `<"cheeseburger", 3.49>` and `<"pepsi", 1.20>`, `getOrderPrice()` should return 8.18 for the input `{"cheeseburger", "cheeseburger", "pepsi"}`.

Give a PoS specification and an implementation for `getOrderPrice()`. In the PoS spec, if an item does not occur, indicate it by writing "none". For example, if your implementation does not return anything write `@return: none`.

(10 points) For each of the following design patterns, give a brief explanation of the design problem that it solves and an example of a situation where it would be appropriate to use the pattern.

- a) Singleton
- b) Observer
- c) Interning
- d) Visitor
- e) Factory

(7 points)

- a) (3 points) Test Driven Development is a strategy where the tests for a module or function are always written before the actual code. Give a main reason why this is a useful strategy other than “it ensures that the tests will actually get written eventually.”
- b) (4 points) We observed that 100% statement coverage wasn’t sufficient to guarantee that a test suite caught all possible bugs. A more comprehensive metric is 100% path coverage, where every possible execution path is executed at least once. But we also said that 100% path coverage is not realistic in most systems. Give a reason why this is true.

(10 points) Give short answers (one or two sentences) to the following questions about refactoring.

- a) What is a reason for refactoring programs?
- b) When should a method be split up into sub methods?
- c) When should a local variable become a class attribute?
- d) What can be done with complex case statements?

e) What is meant by pulling up a method?

(10 points) You find 4 versions of a function that copies the first n elements from List `src` to List `dest`.

```
void partialcopy(List<Integer> dest, List<Integer> src, int n)
```

Fortunately, all the implementations have specifications written in PoS style.

Specification A

@requires: $n > 0$

@modifies: `dest`

@throws: `ArrayOutOfBoundsException` if `src.size() < n`

@effects: for $i=1..n$, `dest[i]post = src[i]pre`

Specification B

@requires: $n > 0$

@modifies: `src`, `dest`

@throws: `ArrayOutOfBoundsException` if `src.size() < n`

@effects: for $i=1..n$, `dest[i]post = src[i]pre`

Specification C

@requires: $n > 0$ and `src.size() >= n`

@modifies: `dest`

@throws: nothing

@effects: for $i=1..n$, `dest[i]post = src[i]pre`

Specification D

@requires: $n > 0$

@modifies: `dest`

@throws: nothing

@effects:

for $i=1..\min(n, \text{src.size}())$, `dest[i]post = src[i]pre`

and for $i=\text{src.size}()+1..n$, `dest[i]post = 0`

In the following diagram, draw an arrow from X to Y if and only if X is stronger than (implies) Y.

A B

C D

(10 points) Consider this method:

```
@returns the least of the 3 arguments
int min3(int x, int y, int z) {
    int a;
    if(x < y) {
        a = x;
    } else {
        a = y;
    }
    if(z < a) {
        return z;
    } else {
        return x;
    }
}
```

- a. Give a test suite for this method with full branch coverage and where all tests pass.
- b. Give a test that does not pass.

(9 points)

(a) The purpose of a Model-View-Controller design is to keep code for the model, the view, and the controller separate. The Java GUI library's use of listeners and callbacks helps keep two of these three things separate. Which two? (No explanation required.)

(b) Why is it bad for a GUI callback method to take a long time to return? (1{2 sentences should be plenty.)

(c) What happens if a client of JButton calls addActionListener on the same button multiple times passing in different objects? (1{2 sentences should be plenty.)

(d) What happens if a client of JButton calls addActionListener on multiple buttons passing in the same object? (1{2 sentences should be plenty.)

(e) True or false (no explanation required): To register an event listener in Java's GUI library, you need to create an anonymous inner class.

(12 points)

Consider the following classes

```
abstract class Bird {
    public abstract void speak();
    public void move() { System.out.println("flap flap!"); }
    public void move(int n) { move(); speak(); }
}

class Canary extends Bird {
    public void speak() { System.out.println("chirp!"); }
    public void move(int n) { speak(); speak(); }
}

class Duck extends Bird {
    public void speak() { System.out.println("quack!"); }
}

class RubberDuck extends Duck {
    public void speak() { System.out.println("squeak!"); }
    public void move() { speak(); swim(); }
    public void swim() { System.out.println("paddle!"); }
}
```

For each of the following groups of statements, write the output that is produced or indicate if there is a compiler error.

- a. Bird b = new Bird();
b.move();
- b. Bird b2 = new Canary();
b2.move(17);
- c. Bird b3 = new Duck();
b3.move(42);
- d. Bird b4 = new RubberDuck();
b4.move(3);
- e. Duck donald = new RubberDuck();
donald.swim();
- f. Duck donald2 = new RubberDuck();
donald2.move();