(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 checks 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 compiler 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 variable are ints.

(a) (5 points)

{ y > 0 }

x = 3;

{y > 0 && x == 3}

y = x + y;

{ y > 3 && x == 3}

z = x * y;

{ y > 3 && x = 3 && z > 9}

Note: y >= 4 && z >=12 is also possible in the final assertion since y > 3 implies y>=4 for integer values.

(b) (5 points)

{ |x| = 42 }

x = x /2;

{ |x| = 21} => { x = -21 || x = 21 }

x = x + 3;

{ x = -18 || x = 24}

Question 2. (12 pts) Compute the weakest precondition with respect to the given postcondition using backwards reasoning. Show each intermediate condition.

Assume x, y and z are ints.

```
wp(y=2*y,(y==0 && true)
        || (y != 0 && (y >= 2 || y <= -6))
      = (y == 0 || (y >= 2 || y <= -6))
if (y == 0) {
        wp(z=4, z>=4 || z<=-4) = 4 >= 4 || 4 <= -4) = true
    z = 4;
}
else {
        wp(z=y+2, z >= 4 || z <= -4) =
            (y+2 >= 4 || y+2 <= -4) = (y >= 2 || y <= -6)
    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.

c,e,h are true

(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;
}
```

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

LI: x * n_orig == result + x * n && (n <= 0) (n_orig is the input n)

Base: before loop, result = 0; n = n_orig -> x * n_orig == result + x * n_orig == 0 + x * n_orig

n <= 0 by precondition

b) Prove correctness of this code, i.e. show by induction that the LI holds before and after each iteration. (8 points)
Loop: assume LI hold for iteration k (8 points)

x * n_orig == result_k + x * n_k

iteration k + 1:

result_k+1 == result_k – x

n_k+1 == n_k + 1

x * n_orig == result_k + x * n_k

== (result_k+1 + x) + x * (n_k+1 – 1)

== result_k+1 + x * n_k+1

Therefor holds for each iteration

If n == 0 at iteration k, we would not have iteration k+1, n increases by 1 at each step, so n <= 0.

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 = -n (n <= 0, -n >= 0. -n decreases by 1 at each iteration, when –n reaches 0, loop exits

d) (5 points)

!(n < 0) && (x * n_orig == result + x * n) && (n <= 0))
⇨ (n == 0) && (x * n_orig == result + x * n)
⇨ x * n_org == result

(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.

m.containsKey(k)	return true if k is a key in map m
m.containsValue(v)	return true if one or more keys in m map to the value v
m.get(k)	return value associated with k, or null if k is not a key in m
m.isEmpty()	return true if m contains no key-value mappings
m.keySet()	return a Set<K> view of the keys in m
m.put(k,v)	store value v with key k in m; return the previous valueassociated with k or null if k was not a key in m

m.remove(k)	remove any key-value mapping withkey k from m
m.size()	return the number of key-value mappings in m
m.values()	return a Collection<V> view of the values in m

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.

RI: items != null (optional because of constructor) and if <K,V> is a key-value pair in items then K!=null and V != null and V > 0. Must include key and value not null and price > 0.

AF: For each pair <K,V> in items, K is the name of an item on the menu and V is its price.

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.

```
/**
* Return the total cost of the items in the list order.
*
* @param order list of items being ordered (param is optional)
*
* @requires order!=null and every item
*  in order appears in the menu
*
* @throws none
*
* @modifies none
*
* @effects none
*
* @return total price of items in order
*
*/
public double getOrderPrice(List<String> order){
   double total = 0;
   for (String item: order) {
         total += items.get(item);
   }
   return total;
}
```

Alternative

```
/**
```

```
   * Return the total cost of the items in the list order.
   *
   * @param order list of items being ordered (param is optional)
   *
   * @requires order!=null
   *
   * @throws none
   *
   * @modifies none
   *
   * @effects none
   *
   * @return total price of items in order
   *
   */

   public double getOrderPrice(List<String> order){
      double total = 0;
      for (String item: order) {
            if (items.containsKey(item) {
                  total += items.get(item);
            }
      }
      return total;
   }
```

Here's another alternative

```
/**
* Return the total cost of the items in the list order.
*
* @param order list of items being ordered (param is optional)
*
* @requires order!=null unless they check for null
*
* @throws Runtime exception if item is not in menu
*
* @modifies none
*
* @effects none
*
* @return total price of items in order
*
*/
public double getOrderPrice(List<String> order){
double total = 0;
for (String item: order) {
   if (!items.containsKey(item) {
      throw new RuntimeException("non-menu item")
   }
   total += items.get(item);
}
   return total;
}
```

(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.

Singleton: we want to ensure that only one instance of a class is created. Examples would be to ensure there is only one random number generator shared throughout a program, or there is only one instance of a class that controls a particular printer or other device.

Observer: Reduce the coupling between objects that produce information and objects that need to examine them. The observed object does not need to know the actual identity of the observers. Examples are views in MVC architectures.

Interning: Ensures that there is only one copy of each abstract value of a class stored in the system. Avoids the costs of constructing multiple copies and the storage and garbage collection expenses of managing them. Also can allow faster comparison of abstract values using == identity tests instead of equals(). Examples are interning strings or Booleanvalues to avoid multiple copies.

Visitor: Provides a standard way to traverse a hierarchical data structure with the correct
method being used to process each node in the structure depending on its actual type. There are many examples such as traversals of abstract syntax trees in compilers, processing structured documents, and so on.

Factory: Hides details of object creation from client code. Examples are code that can operate on specialized representations such as sparse or dense matrices. We would like to isolate the choice of representation to Factory methods or objects so the rest of the code is independent of these decisions.

(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."

   Writing the test helps the author understand better what the code is supposed to do. The effort spent writing tests should ease the work needed to actually write correct code for the implementation later.
   Give points for any reasonable answer.

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.

Loops. (Most programs contain loops that will execute an unknown number of times. It is impossible to execute all possible paths, which would mean all possible numbers of iterations.)

Combinatorial explosion. (Even without loops, the number of paths through the code is an exponential function of the number of branches in the code. In real code it is usually not feasible to execute all possibilities in reasonable amounts of time.)

Rarely used or unusual paths through the code. Some code exists to handle "should never happen" or "almost never could happen" situations and devising tests that exercise all of these paths may not be feasible.

Aliasing.

There may be paired conditions such that if code goes through one branch it cannot got through another.

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

a) What is a reason for refactoring programs?
To increase the maintainability of object-oriented software systems
To increase the testability of object-oriented software systems
To enable the reuse of the object-or iented code in other applications

b) When should a method be split up into sub methods?
When the nesting level of the control logic is too deep
When the method is too large
When the method is performing different logical functions

c) When should a local variable become a class attribute?
Readability
When similar/same variables are used in multiple methods

d) What can be done with complex case statements?
Replace the cases with polymorphic methods

e) What is meant by pulling up a method?
transferring a local method of two or more sub classes to a super class

(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, src.size()), dest[i]post = src[i]pre
and for i=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
|
V
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.

The key is to avoid the path where we take both false branches. For example, a test-suite could be: min3(2,3,4) (expected 2) and min3(3,2,1) (expected 1), though many other answers are possible.

b. Give a test that does not pass.
min3(3,2,4) (expected 2, observed 3)

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.)

The view and the controller

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

It executes on the UI thread, so the application cannot respond to any GUI interactions while the callback executes.

(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.)
They are all registered and all get notified when the button gets pressed.

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

The object's actionPerformed object will be called when any of the buttons is pressed. It can use its argument to determine which button was pressed.

(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.

False

(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();

   Error

b. Bird b2 = new Canary();
   b2.move(17);

   ```
   chirp!
   chirp!
   ```

c. Bird b3 = new Duck();
   b3.move(42);

   ```
   flap flap!
   quack!
   ```

d. Bird b4 = new RubberDuck();
   b4.move(3);

   ```
   squeak!
   paddle!
   squeak!
   ```

e. Duck donald = new RubberDuck();
donald.swim();

Error

f. Duck donald2 = new RubberDuck();
donald2.move();

```
squeak!
paddle!
```