Final Exam
COMP 401
Spring 2014

I have not received nor given any unauthorized assistance in completing this exam.

Signature: ____________________

Name: ______________________

PID: ________________________

The exam has 6 parts for a total of 145 points.

PID: ____________________

This page intentionally left blank.

PID: ________________________

Part I: True or False

45 questions, 1 points each, 45 points total.

___ A local variable in a method may not have the same name as an instance variable.

___ Method parameter names serve as valid local variables within the body of a method.

___ The scope of a local variable in a method is limited to the statement block where it is declared.

___ The type of a variable can be changed after it is declared.

___ A protected instance variable can not be directly accessed by methods in a subclass.

___ When using the factory design pattern, a factory method for creating new instances is usually defined as a static class method.

___ A constructor may throw an exception.

___ One constructor can invoke another constructor in the same class using the `super` keyword as if it were a method name.

___ Class variables can be accessed by instance methods but instance variables cannot be accessed by class methods.

___ A setter method conforming to the Java Beans convention should return the new value of the property.

___ An interface definition can include private or protected methods.

___ An iterator provides a way to access elements of an aggregate object sequentially by exposing its underlying representation.

___ Traversing an array using an integer index incremented as a for-loop variable is an example of the iterator design pattern.

___ An expression can always be used where ever a value is required.

___ The state of an object is defined by the current values of all of its instance fields.

___ An iterator assumes that the underlying collection is not changed or modified while the traversal occurs.

___ Class names, interface names, and enumeration names can all serve as the data type of a variable.

___ In an instance method, the `this` keyword always provides a reference to the specific instance used to invoke the method.

___ A method's signature is defined by its return type and the sequence of types associated with its parameters.

___ Polymorphic methods with the same name may have different return types.

___ Polymorphic methods with the same name must have distinct method signatures.

___ Enumerations provide a type-safe mechanism for an object property that can be set to one of a distinct set of values.

___ Different classes that implement the same set of interfaces may also have different and additional behavior not declared by those interfaces.

___ Two object references that have the same value always refer to the same object.

___ A derived property is computed as a function of an object's state.

___ The principle of encapsulation suggests that instance fields should generally be marked public.

___ The Decorator pattern makes use of delegation.

___ The state of an immutable object cannot be changed after it is created.

___ A class that implements a subinterface must provide implementations for methods declared by the subinterface as well as any parent interfaces of that subinterface.

___ A subclass has direct access to private and protected fields and methods in its parent class.

___ All classes implicitly inherit from Object.

___ The catch or specify policy applies to all subclasses of Exception.

___ In general, you should catch an exception at the earliest point at which the exception can be detected.

___ Java user interface components employ the delegation design pattern by registering "listeners" for specific types of events.

___ All overriding methods in Java are virtual.

___ If a catch block is provided for the Exception parent class, it should appear first before any other catch blocks.

___ A composite object can claim to implement any interface associated with its components by employing delegation.

___ When a top-level Java Swing window is made visible, a new thread of execution is started.

___ All methods of a class that are declared "synchronized" must be executed concurrently by separate threads.

___ An abstract class can not be instantiated directly.

____ A Java array can dynamically change the number of elements it contains.

____ The Model-View-Controller design pattern employs the Observer/Observable pattern.

____ The return type of a getter method following JavaBeans conventions may be void.

____ Casting a reference to an object from a subclass type to a parent class type is an example of cotravariance.

____ A class that acts as an Aggregation of other objects will usually have references to its component objects provided as parameters to its constructor.

Part II: Evaluating and Understanding Code

30 points total (5 points each for subparts a and b, 10 points each for subparts c and d).

a) Given the following definition of class Foo:

```
public class Foo {
  public static int a = 8;
  public int b;

  public Foo(int c) {
    b = c;
  }

  public void bar() {
    if (b < a) {
      b = a*3;
    } else {
      a = b*2;
    }
  }

  public void bar(Foo f) {
    if (b < f.b) {
     b = f.b * 2;
    } else {
      f.b = a + b;
    }
  }
}
```

Fill in the table to the right, providing the value of the expressions f1.b, f2.b, and Foo.a **after** each of the lines 3 through 7 of the following code snippet is executed (note, line numbers provided for reference and are NOT part of the code).

```
1: Foo f1 = new Foo(7);
2: Foo f2 = new Foo(12);
3: f1.bar();
4: f2.bar();
5: f1.bar(f2);
6: f2.bar(f1);
7: f2.bar();
```

Line #	f1.b	f2.b	Foo.a
3			
4			
5			
6			
7			

b) Given the following definition of the class Foo:

```
public class Foo {
  private int[] bar;

  public Foo(int[] b) {
    bar = b;
  }

  public int getIndex(int index) {
    return bar[index];
  }

  public void incrementAtIndex(int index, int delta) {
    bar[index] += delta;
  }

  public int getProduct() {
    int product = 1;
    for (int i : bar) {
      product *= i;
    }
    return product;
  }
}
```

What are the values of the variables `r1`, `r2`, `r3`, `r4`, and `r5` after the following code executes:

```
int[] int_array = {3, 2, 1};
Foo f1 = new Foo(int_array);
Foo f2 = new Foo(int_array);
f1.incrementAtIndex(2, 5);
f2.incrementAtIndex(1, -5);
int r1 = f1.getIndex(2);
int r2 = f2.getIndex(1);
int r3 = int_array[1];
int r4 = int_array[2];
int r5 = f1.getProduct() - f2.getProduct();
```

Answers:

r1 = ______

r2 = ______

r3 = ______

r4 = ______

r5 = ______

c) Given the following class and interface definitions (actual bodies of these definitions are not important and are represented by ellipses as a placeholder):

```
public interface InterA {...}
public interface InterB extends InterA {...}
public interface InterC extends InterA {...}
public interface InterD extends InterC {...}
public class A implements InterA {...}
public class B implements InterB {...}
public class C extends A implements InterC {...}
public class D extends B implements InterD {...}
public class E extends C implements InterB {...}
```

Suppose the following variables are defined:

```
A a1 = new A();
B b1 = new B();
C c1 = new C();
D d1 = new D();
E e1 = new E();
```

Indicate whether each of the following lines is legal or illegal by circling the appropriate word in the comment that trails each line.

```
InterA ia1 = (InterA) d1;            // Legal or Illegal
InterB ib1 = (InterB) c1;            // Legal or Illegal
A a2 = (A) c1;                       // Legal or Illegal
InterC ic1 = (InterC) a1;            // Legal or Illegal
E e2 = (E) c1;                       // Legal or Illegal
InterC ic1 = (InterC) e1;            // Legal or Illegal
InterA ia2 = (InterA) b1;            // Legal or Illegal
B b2 = (B) d1;                       // Legal or Illegal
InterA ia3 = (InterA) ((InterD) d1); // Legal or Illegal
InterB ib2 = (InterB) ((C) e1);      // Legal or Illegal
```

d) The following code contains at least 10 errors. Identify and correct them.

```
class StringFinder {
  private String needle;

  public StringFinder(String needle) {
    needle = this.needle;
  }

  public boolean findNeedleInOne(String haystack) {
    for (i = 0; i < haystack.length(); i++) {
      if (haystack.charAt(i) = needle.charAt(0)) {
        boolean found = true
        for (int j = 1; j <= needle.length; j++) {
          try {
            if (needle.charAt(j) !=
                haystack.charAt(i + j)) {
              found = false;
              break;
            }
          } catch (IndexOutOfBoundsException e) {
            found == false;
            break;
          }
        }
        if (!found) {
          return true;
        }
      }
    }
    return false;
  }

  public boolean[] findNeedleInMany(String[] haystacks) {
    boolean results = new boolean[haystacks.length];
    int i = 0;
    for (h : haystacks) {
      i += 1;
      results[i] = findNeedleInOne(h);
    }
    return results;
  }
}
```

Part III: Writing Code To Specification

15 points total.

Implement a histogram, which keeps track of the number of times a value occurs. You can assume that the values are in the range 0…MaxValue where MaxValue is a read-only property of the histogram specified when the object is instantiated. The histogram provides a method to add an occurrence of a value and another to return the number of occurrences of a value. The interface of the histogram is given below:

```
public interface Histogram {
  public void addOccurence(int value);
  public int numOccurences(int value);
  public int getMaxValue();
}
```

The following code illustrates the use of a the Histogram class:

```
public static void main(String[] args) {
  // new histogram for values in range 0..4
  Histogram histogram = new AHistogram(4);

  // adding occurrences
  histogram.addOccurence(0); // an  occurrence of 0
  histogram.addOccurence(2); // an occurrence of 2
  histogram.addOccurence(4); // an occurrence of 4
  histogram.addOccurence(0); // another occurrence of 0
  System.out.println(histogram.getMaxValue()); // prints 4

  // retrieving occurrences
  System.out.println(histogram.numOccurences(0)); // prints 2
  System.out.println(histogram.numOccurences(1)); // prints 0
  System.out.println(histogram.numOccurences(2)); // prints 1
  System.out.println(histogram.numOccurences(4)); // prints 1
}
```

Write an implementation of the class AHistogram used in the code above. The constructor and method headers are given to you on the next page. To get full credit, you should create a single array in the class. You do not have to check for error conditions.

```
public class AHistogram implements Histogram {
// 4 pts, instance variables




// 4 pts, constructor given max value
  public AHistogram(int aMaxValue) {





  }
// 1 pt, max value in range. Min value is 0
  public int getMaxValue() {





  }
// 4 pts, add an occurrence of value
  public void addOccurence(int value) {






}
// 2 pts, number of occurrences of value
  public int numOccurences(int value) {







  }
}
```

PID: ________________________

Part IV: Inheritance

25 points total (credited at discretion of grader)

Given the following definitions for classes `Terrier`, `Beagle`, `Siamese`, and `Tabby`, reorganize the code to employ inheritance as most appropriate. Your new version of the code should include a parent class called `Pet` with two subclasses called `Dog` and `Cat`. Your new versions of `Terrier` and `Beagle` should be subclasses of `Dog` and your new versions of `Siamese` and `Tabby` should be subclasses of `Cat`.

```
public class Terrier {
  private enum TailPosition {TAIL_LEFT, TAIL_RIGHT};

  private String name;
  private TailPosition tail_position;

  public Terrier(String name) {
    this.name = name;
    tail_position = TAIL_LEFT;
  }

  public void wagTail() {
    if (tail_position == TailPosition.TAIL_LEFT) {
      tail_position = TailPosition.TAIL_RIGHT;
    } else {
      tail_position = TailPosition.TAIL_LEFT;
    }
  }

  public String speak() {
    return “Woof”;
  }

  public String getName() {
    return name;
  }
}
```

```
public class Beagle {
  private enum TailPosition {TAIL_LEFT, TAIL_RIGHT};

  private String name;
  private TailPosition tail_position;

  public Beagle(String name) {
    this.name = name;
    tail_position = TAIL_RIGHT;
  }

 public void wagTail() {
    if (tail_position == TailPosition.TAIL_LEFT) {
      tail_position = TailPosition.TAIL_RIGHT;
    } else {
      tail_position = TailPosition.TAIL_LEFT;
    }
  }

  public String speak() {
    return “Woof”;
  }

  public String getName() {
    return name;
  }
}
```

```
public class Siamese {
  private String name;
  private boolean is_clean;

  public Siamese(String name) {
    this.name = name;
    is_clean = true;
  }

  public void makeDirty() {
    is_clean = false;
  }

  public void makeClean() {
    is_clean = true;
  }

  public boolean isClean() {
    return is_clean;
  }

  public String speak() {
    return “Meow”;
  }

  public String getName() {
    return name;
  }
}
```

PID: ____________________

```
public class Tabby {
  private String name;
  private boolean is_clean;

  public Tabby(String name) {
    this.name = name;
    is_clean = true;
  }

  public void makeDirty() {
    is_clean = false;
  }

  public void makeClean() {
    is_clean = true;
  }

  public boolean isClean() {
    return is_clean;
  }

  public String getName() {
    return name;
  }
}
```

PID: ____________________

```
// Put your part IV code here.
```

PID: ____________________

```
// Continue your part IV code here if necessary.
```

PID: ____________________

```
// Continue your part IV code here if necessary
```

Part V: Decorator

20 points (credited at discretion of grader)

Suppose the Person interface is defined as follows:

```
public interface Person {
  double getWeightInKG();
  double getHeightInMeters();
}
```

Use the decorator pattern as part of an implementation for a class called `Population` that represents a collection of `Person` objects. A `Population` instance should provide the following methods:

- `public int size()`
  - Returns the number of Person objects in the collection.
- `public void addToPopulation(Person p)`
  - Adds a person to the population.
- `public void removeFromPopulation(Person p)`
  - Removes a person from the population.
- `public double getAverageBMI()`
  - Returns the average Body Mass Index (BMI) for the population. The BMI of each person is calculated by the formula: weight / height^2, where weight is in kilograms and height is in meters.
- `public Person[] findPeopleInBMIRange(double min_bmi, double max_bmi)`
  - Returns an array of Person objects from the population that have BMI values in the range min_bmi to max_bmi (inclusive).

You may (and in fact will need to) define additional interfaces and/or classes to support your implementation as necessary. You can assume that the ArrayList class from java.util is available to you. Below is an ArrayList cheat sheet:

To create an ArrayList with items of type E:

```
     ArrayList<E> alist = new ArrayList<E>();
```

To find the number of items in an ArrayList:

```
     int num_items = alist.size();
```

To add to the ArrayList:

```
     alist.add(item)
```

To remove an item from the ArrayList:

```
     alist.remove(item);
```

To convert an ArrayList with items of type E to an array of type E

```
     E[] e_array = alist.toArray(new E[alist.size()]);
```

PID: ______________________

```
// Put your code for Part V here.
```

PID: ____________________

```
// Continue your code for Part V here if necessary.
```

PID: ________________________

Part VI: Model-View-Controller

10 points (1 pt per statement)

Label each of these statements with an M, V, or C depending on whether the statement pertains to the Model, View, or Controller in the classic MVC design pattern.

____ Defines application behavior.

____ Encapsulates application state.

____ Registers as an observer of view components.

____ Renders application state to the user.

____ May dynamically update or change view as a response to user interaction.

____ Responds to state queries.

____ Provides mechanism for observing user gestures.

____ Maps user actions to model updates.

____ Registers as an observer of model components.

____ Notifies view components of changes.