Final Exam
COMP 401
Fall 2014

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

Signature: ________________

Name: ________________

PID: ________________

The exam has 4 parts for a total of 135 points.
Part I: Multiple choice (75 points total, 5 points per question)

Directions:

For each multiple choice question below, indicate all correct answers among the choices provided. More than one answer may be correct and at least one answer will be correct. Each question is worth 5 points. Selecting all of the choices or selecting none of the choices will result in no points awarded.

1. Which of these statements about Java code organization and syntax are true?
   a) A block of statements can be used anywhere an expression is needed.
      * b) A statement ends in a semicolon.
      * c) Classes organized into a package are visible to each other without an import statement.
      * d) A Java source file can only contain one public class or interface.
   e) Local variables declared within a loop are accessible outside of the loop body.

2. Which of these statements about Java arrays are true?
   a) Arrays are indexed starting with 1.
      * b) Arrays are objects.
   c) An array can hold elements of different types.
      * d) An array can not be resized once created.
   e) An array may not be passed to a method as a parameter.

3. Which of these statements about encapsulation are true?
   * a) Using private instance fields is related to the principle of encapsulation.
   b) An object with one or more setters is immutable.
   c) A getter must be associated with a specific instance field.
   * d) An advantage of setters is the ability to validate the new value of a property.
   e) Proper encapsulation requires all methods to be marked public.

4. Which of these statements about polymorphism are true?
   * a) Overloaded methods are an example of polymorphism.
   b) Using the JavaBeans conventions for getters and setters is an example of polymorphism.
   c) Polymorphism requires that at most one class will implement a particular interface.
   d) Using the same local variable name in different methods is an example of polymorphism.
   * e) The "is-a" relationship between a class and the interfaces that the class implements is an example of polymorphism.
5. Which of these statements about subclassing are true?
   a) A subclass can override private methods in its parent class.
      * b) A subclass may only have one parent class.
   c) Two subclasses of the same parent class have an is-a relationship with each other.
   d) A subclass has direct access to private fields in its parent class.
   e) A subclass must redeclare any protected fields declared in its parent class.

6. Which of these statements about interfaces and subinterfaces are true?
   * a) An enumeration can be declared within an interface.
   * b) A subinterface may have multiple parent interfaces.
   c) An interface may include one or more private methods.
   * d) An extended interface adds one or more additional methods to those declared by its parent.
   * e) A public interface should be defined in a Java source file with the same name.

7. Which of these statements about overriding methods are true?
   a) Every method of a parent class should be overridden by a method in a subclass.
   b) An overridden method is not able to call the version of the method as defined in the parent class.
      * c) An overridden method in a subclass has the same method signature as a method in the parent class.
      * d) Overridden methods are virtual by definition in Java.
      * e) A class can simultaneously overload and override the same method.

8. Which of these statements about overloading methods are true?
   a) An overloaded method may not call (i.e., invoke) a different version of the overloaded method.
      * b) An overloaded method is one that is defined more than once within a class with a unique method signature associated with each definition.
      * c) A subclass can overload a method defined in its parent class.
      * d) All versions of an overloaded method must have the same access modifier (e.g., public, protected, private).
      e) All versions of an overloaded method must have the same return type.
9. Which of these statements about the iterator design pattern are true?
   * a) A for-each loop can be used with any collection that implements the java.util.Iterable interface.
   b) An existing iterator must iterate over the entire collection before another iterator can be created for that collection.
   * c) The iterator design pattern provides an abstraction for iterating over a collection without needing to understand how the collection is maintained.
   * d) An iterator can be passed to a method as a parameter.
   e) An iterator must be a subclass of the iterable collection.

10. Which of these statements about the factory design pattern are true?
    a) The factory design pattern uses delegation.
    * b) A class using the factory design pattern prevents direct instantiation of objects.
    c) The factory method in the factory design pattern is usually an instance method.
    * d) The factory design pattern can be used for dynamic subclass binding.
    * e) The factory design pattern can be used to create a singleton.

11. Which of these statements about the decorator design pattern are true?
    * a) The decorator design pattern extends the functionality of an object without subclassing.
    b) A class that implements a decorated interface will not also implement the original interface.
    * c) The decorator design pattern uses delegation.
    * d) The decorator design pattern relies on subinterfacing.
    e) The constructor of a class implementing a decorated interface will not generally need any parameters.

12. Which of these statements about the MVC design pattern are true?
    a) The model component is responsible for invoking controller methods when the user interacts with the application.
    * b) The view component can be an observer of state changes within the model component.
    * c) The MVC design pattern is an application-level pattern that separates the concerns of maintaining application state, rendering a current subset of the application state, and implementing application-level logic.
    d) The controller component is usually a subclass of the view component.
    * e) MVC is often used for interactive applications with user interfaces.
13. Which of these statements about composition and aggregation are true?

* a) Composition and aggregation are two ends of a design spectrum for modeling an object as a "whole" comprised of many "parts".

b) A composition is characterized by a constructor that is provided references to its component parts as parameters.

c) The components of an aggregation are not likely to be useful as independent objects or as part of other aggregations.

* d) Delegation may be used within both compositions and aggregations.

* e) An aggregation is more likely to provide getters and setters for its components than a composition.

14. Which of these statements about multithreaded programming are true?

a) All objects of the same class share a single lock.

b) Two different threads may be simultaneously executing within different methods of an object that are marked as synchronized.

* c) A synchronized method may call a different synchronized method of the same object because it has already obtained the object lock.

d) A thread can only obtain one object lock at a time.

* e) Multithreaded programming is the key to building software that can take advantage of advances in modern hardware.

15. Which of these statements about exceptions are true?

a) As a general principle, one should catch exceptions are early as possible.

* b) An exception is an event that occurs during the execution of a program that signals a disruption in the normal flow of the program.

c) Every exception is subject to Java’s catch-or-specify rule.

* d) It is possible to nest a try-catch block within a catch block.

* e) The finally block will always execute even if an exception is not thrown.
Part II: Evaluating and Understanding Code

20 points total (5 points each for subparts a and b, 10 points each for subpart c).

a) Given the following definition of class Foo:

```java
public class Foo {
    public int x;
    public int y;

    public Foo(int a) {
        x = a;
        y = 1;
    }

    public int bar(int a) {
        x += a;
        y += 1;
        return (x+y);
    }

    public void bar(Foo f) {
        if (f.x < x) {
            f.bar(x);
        } else {
            bar(f.x);
        }
    }
}
```

Fill in the table to the right, providing the value of the expressions f1.x, f1.y, f2.x, and f2.y 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).

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

<table>
<thead>
<tr>
<th>Line #</th>
<th>f1.x</th>
<th>f1.y</th>
<th>f2.x</th>
<th>f2.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>3</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>3</td>
<td>29</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
<td>3</td>
<td>29</td>
<td>4</td>
</tr>
</tbody>
</table>
b) Given the following definition of the class Foo:

```java
public class Foo {
    private int[] vals;

    public Foo(int[] vals) {
        this.vals = vals;
    }

    public int[] getVals() {
        return vals;
    }

    public int getIndex(int i) {
        return vals[i];
    }

    public void addOtherVals(int[] other_vals) {
        for (int i=0; i<vals.length; i++) {
            vals[i] += other_vals[i];
        }
    }

    public void swapBiggerVals(int[] other_vals) {
        for (int i=0; i<vals.length; i++) {
            if (vals[i] > other_vals[i]) {
                int tmp = other_vals[i];
                other_vals[i] = vals[i];
                vals[i] = tmp;
            }
        }
    }
}
```

What are the values of the variables $r_1$, $r_2$, $r_3$, $r_4$, and $r_5$ after the following code executes:

```java
Foo f1 = new Foo(new int[] {1, 4, 5});
Foo f2 = new Foo(new int[] {2, 3, 6});
f1.swapBiggerVals(f2.getVals());
f2.addOtherVals(f1.getVals());
f2.swapBiggerVals(f1.getVals());
int r1 = f1.getIndex(0);
int r2 = f1.getIndex(1);
int r3 = f1.getIndex(2);
int r4 = f2.getIndex(1);
int r5 = f2.getIndex(2);
```

Answers:

$r_1 = ___3___$

$r_2 = ___7___$

$r_3 = ___11___$

$r_4 = ___3___$

$r_5 = ___5___$
c) Given the following class and interface definitions (actual bodies of these definitions are not important and are represented by ellipses as a placeholder):

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

Suppose the following variables are defined:

```java
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.

```java
InterA ia1 = (InterA) c1;            // Legal
InterB ib1 = (InterB) d1;            // Legal
A a2 = (A) e1;                      // Illegal
InterC ic1 = (InterC) a1;            // Illegal
B b2 = (B) e1;                      // Legal
InterC ic2 = (InterC) d1;            // Illegal
InterA ia2 = (InterA) c1;            // Legal
D d2 = (D) c1;                       // Illegal
InterB ib2 = (InterA) ((B) c1);      // Legal
InterD id1 = (D)((B) e1);            // Legal
```
Part III: Inheritance

20 points total (credited at discretion of grader)

Given the following definitions for classes Student, Faculty, and Staff, reorganize the code to employ inheritance as most appropriate. Your new version of the code should include a parent class called Person.

```java
public class Student {
    public enum Status {FRESHMAN, SOPHOMORE, JUNIOR, SENIOR};

    private String name;
    private int pid;
    private Status status;

    public Student(String name, int pid, Status status) {
        this.name = name;
        this.pid = pid;
        this.status = status;
    }

    public String getName() {
        return name;
    }

    public int getPID() {
        return pid;
    }

    public Status getStatus() {
        return status;
    }

    public String speak() {
        return "Yo!";
    }
}
```
public class Faculty {
    public enum Status {ASSISTANT, ASSOCIATE, FULL};

    private String name;
    private int pid;
    private Status status;

    public Faculty(String name, int pid, Status status) {
        this.name = name;
        this.pid = pid;
        this.status = status;
    }

    public String getName() {
        return name;
    }

    public int getPID() {
        return pid;
    }

    public Status getStatus() {
        return status;
    }

    public String speak() {
        return "Greetings!";
    }
}
public class Staff {
    public enum Status {PART_TIME, FULL_TIME};

    private String name;
    private int pid;
    private Status status;

    public Student(String name, int pid, Status status) {
        this.name = name;
        this.pid = pid;
        this.status = status;
    }

    public String getName() {
        return name;
    }

    public int getPID() {
        return pid;
    }

    public Status getStatus() {
        return status;
    }

    public String speak() {
        return "Cheers!";
    }
}
public abstract class Person {
    private String name;
    private int pid;

    protected Person (String name, int pid) {
        this.name = name;
        this.pid = pid;
    }

    public String getName() { return name; }
    public int getPID() { return pid; }

    abstract public String speak();
}

public class Student extends Person {
    public enum Status {FRESHMAN, SOPHOMORE, JUNIOR, SENIOR};

    private Status status;

    public Student(String name, int pid, Status status) {
        super(name, pid);
        this.status = status;
    }

    public Status getStatus() {return status; }

    @Override
    public String speak() { return "Yo!";}
}
// Continue your part III code here if necessary.

public class Faculty extends Person {
    public enum Status {ASSISTANT, ASSOCIATE, FULL};

    private Status status;

    public Faculty(String name, int pid, Status status) {
        super(name, pid);
        this.status = status;
    }

    public Status getStatus() {return status; }

    @Override
    public String speak() { return "Greetings!";}
}

public class Staff extends Person {
    public enum Status {PART_TIME, FULL_TIME};

    private Status status;

    public Staff(String name, int pid, Status status) {
        super(name, pid);
        this.status = status;
    }

    public Status getStatus() {return status; }

    @Override
    public String speak() { return "Cheers!";}
}
Part IV: Observer/Observable
20 points (credited at discretion of grader)

Consider the following classes:

class DicePair {
    private int die1;
    private int die2;

    public DicePair() { roll(); }
    public void roll() {
        // The following sets each field
        // to a random number from 1 to 6.
        die1 = ((int) (Math.random() * 6.0)) + 1;
        die2 = ((int) (Math.random() * 6.0)) + 1;
    }

    public int getDie1() { return die1; }
    public int getDie2() { return die2; }
}

class DoublesCounter {
    private int num_doubles;

    public DoublesCounter() { num_doubles = 0; }
    public int getNumDoubles() { return num_doubles; }
    public void detectDoubles(int v1, int v2) {
        if (v1 == v2) {
            num_doubles++;
        }
    }
}

class SevenDetector {
    private int num_sevens;

    public SevenDetector() { num_sevens = 0; }
    public int getNumSevens() { return num_sevens; }
    public void detectSeven(int value) {
        if (value == 7) {
            num_sevens++;
        }
    }
}
Rewrite these classes in order to apply the Observer/Observable design pattern so that DoublesCounter and SevensDetector objects can be used as observers of DicePair objects. A DicePair object should inform any observers when the roll() method is called. DoublesCounter objects should keep track of the number of times “doubles” is rolled by any DicePair objects observed. SevenDetector objects should keep track of the number of times the sum of the dice of any observed DicePair objects is equal to seven. You are free to create any new interfaces and/or classes you need in order to appropriately apply the Observer/Observable design pattern. Do NOT rely on java.util.Observer and/or java.util.Observable. You may rely on ArrayList<> as a collection class. Below is an ArrayList cheat sheet for your reference:

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);
interface DiceObserver {
    observeDicePair(DicePair dp);
}

class DicePair {
    private int die1;
    private int die2;
    private List<DiceObserver> observers;

    public DicePair() {
        observers = new ArrayList<DiceObserver>();
        roll();
    }

    public void roll() {
        // The following sets each field
        // to a random number from 1 to 6.
        die1 = (((int) (Math.random() * 6.0)) + 1);
        die2 = (((int) (Math.random() * 6.0)) + 1);
        notifyObservers();
    }

    public void addObserver(DiceObserver o) { observers.add(o); }
    public void removeObserver(DiceObserver o) { observers.remove(o); }
    public void notifyObservers() {
        for (DiceObserver o : observers) {
            o.observeDicePair(this);
        }
    }

    public int getDie1() { return die1; }
    public int getDie2() { return die2; }
}
// Continue your code for Part IV here if necessary.

class DoublesCounter implements DiceObserver {
    private int num_doubles;

    public DoublesCounter() { num_doubles = 0; }
    public int getNumDoubles() { return num_doubles; }
    public void detectDoubles(int v1, int v2) {
        if (v1 == v2) {
            num_doubles++;
        }
    }

    public void observerDicePair(DicePair dp) {
        detectDoubles(dp.getDie1(), dp.getDie2());
    }
}

class SevenDetector implements DiceObserver {
    private int num_sevens;

    public SevenDetector() { num_sevens = 0; }
    public int getNumSevens() { return num_sevens; }
    public void detectSeven(int value) {
        if (value == 7) {
            num_sevens++;
        }
    }

    public void observerDicePair(DicePair dp) {
        detectSeven (dp.getDie1() + dp.getDie2());
    }
}