Comp 413: Test 1 Name: Answer Sheet   Grade: / 15

Dr. Yacobellis • Fall 2015 • Tue 22 September • 50 minutes • open book/notes

You may use your computer or e-reading device while taking this test, provided it is not used for any other purpose other than reading course textbooks and other course-related textual materials including the code examples provided for this course. That is, it must not be used for communication, programming, or anything that would specifically help you to answer the questions. As a clarifying example, an internet search from a web browser is a specific violation of the communication requirement. If you are unable to work within these constraints, you are required to turn off the computer/device at this time and rely on your own memory and/or printed materials.

By signing below, you agree that you have acted ethically while preparing for and taking this exam and agree to be bound by the applicable university academic integrity policies and procedures. (Please do not forget to print your name above.)

Signature (required): ____   Date: _______________

Please answer briefly and concisely, providing justification where appropriate. Concepts are more important than syntax, so please use pseudo-code when you are unsure about syntactic details and provide additional explanations where appropriate.

Problem 1 (2.5 points)

a) **1.5** For *each* of the following requirements, indicate whether it is *1) a functional, 2) a static (evolution) nonfunctional, or 3) a dynamic (execution) nonfunctional* requirement. **(0.3 points each)**

- The program can be changed easily to enhance its functionality: **static nonfunctional (2)**
- The program runs in linear time in the number of input lines: **dynamic nonfunctional (3)**
- The program counts the number of its input lines: **functional (1)**
- The program can be tested easily: **static nonfunctional (2)**
- The program crashes in fewer than 5% of all invocations: **dynamic nonfunctional (3)**

b) **0.5** Why can testability be considered the most important nonfunctional requirement? *Circle exactly one.*

- [✓ Testability enables us to examine whether a program satisfies its functional requirements.]
- Testability also guarantees performance and reliability.
- Testability usually results in higher program portability.
- Testability requires pair programming.
- All of the above.

c) **0.5** How are performance and maintainability typically related? *Circle exactly one.*

- The more maintainable a program, the faster it will run.
- The faster a program runs, the more maintainable it will be.
- It may be necessary to sacrifice some maintainability in favor of performance.
- Maintainability requires pair programming, while performance does not.
- All of the above.

Problem 2 (3.5 points)

Remember that boolean values print as “true” or “false”.

a) **0.5** What is the output of the following Java code? Briefly justify your answer. **(0.25 points each)**

```
float x = 1.3;
float y = x;
System.out.println(x == y); // answer here -> true (float values are the same)
y = 2.8;
System.out.println(x == y); // answer here -> false (independent float values, now different)
```

```
float y = x;
System.out.println(x == y); // answer here -> true (float values are the same)
y = 2.8;
System.out.println(x == y); // answer here -> false (independent float values, now different)
```

Now consider the following Java class. Recall that *x instanceof C* tells you whether the *dynamic* type of x is C or a subtype of C.

```
class Coin {
   private float value;
   private float weight;
   public void setValue(final float value) { this.value = value; }
   public float getValue() { return value; }
   public float getWeight() { return weight; }
   public Coin(final float value, final float weight) {
      this.value = value;
      this.weight = weight;
   }
   public boolean equals(final Object that) {
       return (that instanceof Coin) && (this.getValue() == ((Coin) that).getValue());
   }
}
```

b) **0.5** ***TRUE or FALSE (circle one)?*** The **equals** method does not care about the weight of the coin because coins are compared based on their face value.

c) **1** Given the above Coin class, what is the output of the following code? **(0.25 points for each true/false answer)**

```
final Coin n1 = new Coin(0.05, 5.0);
final Coin n2 = new Coin(0.05, 5.1);
System.out.println(n1.equals(n2) + " " + (n1 == n2)); // answer here -> true false (same value, different objects)
n2.setValue(0.10);
System.out.println(n1.equals(n2) + “ “ + (n1 == n2)); // answer here -> false false (different values and objects)
```

d) **1** Given the above Coin class, what is the output of the following code? **(0.25 points for each true/false answer)**

```
final Coin n1 = new Coin(0.05, 5.0);
final Coin n2 = n1;
System.out.println(n1.equals(n2) + " " + (n1 == n2)); // answer here -> true true (same object)
n2.setValue(0.10);
System.out.println(n1.equals(n2) + " " + (n1 == n2)); // answer here -> true true (same object, updated)
```

e) **0.5** Given the above Coin class, what is the output of the following code? **(0.25 points for each true/false answer)**

```
final Coin n1 = new Coin(0.05, 5.0);
System.out.println(n1.equals("0.05")); // answer here -> false (object is String, not Coin)

System.out.println(n1.equals(Double.valueOf(0.05))); // answer here -> false (object is Double, not Coin)
```

Problem 3 (9.5 points)

Following a test-driven development (TDD) approach, consider first this JUnit test fragment:

```
final IntPair p = new DefaultIntPair(3, 5);
final IntPair q = new DefaultIntPair(5, 3);

assertEquals(3, p.first());
assertEquals(5, p.second());
assertEquals("<3, 5>", p.toString());
assertTrue(p.equals(p));
assertFalse(p.equals(q));
assertFalse(q.equals(p));
assertFalse(p.equals(null));
assertTrue(p.equals(q.reverse()));
```

a) **1.5** Your first job is to **reverse-engineer the IntPair interface** in a way that is consistent with the test shown above. Leave out any methods already provided by the class java.lang.Object. Write the interface here:
(0.5 points per method; -1 if the interface is not declared properly; -0.25 for each Object method included)

```
public interface IntPair { // public not required
   int first();
   int second();
   IntPair reverse();
}
```

b) **3** Your next job is to **write the DefaultIntPair class** that implements the IntPair interface in a way that the test shown above passes. *I.e., implement all pair methods used in the test.* Keep in mind that pair instances, once created, are *immutable. Hints: It is OK to create new pair instances when needed; use Coin.equals as a guide;*

b) **3** Your next job is to **write the DefaultIntPair class** that implements the IntPair interface in a way that the test shown above passes. *I.e., implement all pair methods used in the test.* Keep in mind that pair instances, once *created, are immutable. Hints: It is OK to create new pair instances when needed; use Coin.equals as a guide; you don't have to implement hashCode, but don't forget toString.*
(0.5 points for each correctly implemented method & the constructor (must be public); -1 if no "implements")

```
public class DefaultIntPair implements IntPair {
    private int first, second;

    public DefaultIntPair(int first, int second) {
        this.first = first; this.second = second;
    }
    // methods required by the IntPair interface
    public int first() { return first; }
    public int second() { return second; }
    public DefaultIntPair reverse() {
        return new DefaultIntPair(second, first);
    }
    // Overridden Object methods required by @Test method calls
    @Override // optional
    public String toString() {
        return "<" + first + ", " + second + ">";
    }
    @Override // optional
    public boolean equals(final Object that) { // "final" is optional
        return (that instanceof DefaultIntPair) && (this.first() == ((DefaultIntPair) that).first())
                    && (this.second() == ((DefaultIntPair) that).second());
    }
}
```

c) **2** Now **generalize the IntPair interface by making it *generic in the item type of the pair***, allowing us to store two items not only of type int but of *arbitrary reference type as long as both items are of that same type.* Show the code (or explain what needs to be changed) for the resulting generic interface **Pair**.
(0.5 points per method, 0.5 points for declaring the generic version of the interface; -1 if the interface is not declared properly; -0.25 for each Object method included)

```
public interface Pair<T> { // note: not Pair<T, U>, must be the same type; Pair<T, T> is illegal, can't repeat the type var
    T first();
    T second();
    Pair<T> reverse(); // return type is now generic
}
```

d) **1 Redefine p and q from the test fragment above using the new generic types Pair and DefaultPair** (the generic version of DefaultIntPair). *(Do not show the rest of the test code, just the first 2 modified lines!)*
(0.5 points for each)

```
final Pair<Integer> p = new DefaultPair<Integer>(3, 5); // "final" optional
final Pair<Integer> q = new DefaultPair<Integer>(5, 3); // "final" optional
```

e) **0.5 *TRUE or FALSE (circle one)?*** No changes to the *rest of the test code* above are required for applying the test to the new generic types Pair and DefaultPair.

f) **0.5** Why is it beneficial for Pair and its implementation class(es) to be generic? *Circle exactly one.*

- Performance: the program will run faster.
- Reliability: the program will crash less often.
- Generality: Pair can be used whenever a pair of two values of the same type is required.
- None of the above.

g) **0.5** Why is it beneficial to write client code against the Pair interface instead of the DefaultPair class? *Circle exactly one.*

- Performance: the program will run faster.
- Flexibility: we can switch from DefaultPair to some other implementation of Pair without changing the client code.
- Conciseness: the resulting client code is much shorter.
- None of the above.

h) **0.5** Why would it make sense for the Pair interface to extend the List interface? *Circle exactly one.*

- None of the above.

h) **0.5** Why would it make sense for the Pair interface to extend the List interface? *Circle exactly one.*

- Performance: the program will run faster.
- Compatibility: existing methods on List will work on Pair instances.
- Conciseness: the resulting client code is much shorter.
- None of the above.

1