

CIS 120

Midterm I — February 4, 2009

Name: _____

Student ID: _____
(from your PennCard)

Email _____ @seas.upenn.edu

Lab:(circle one)

201	202	203	204
W 1:00 PM	W 2:30 PM	R 12:00 PM	R 1:30 PM

- This exam text and attached code, the *Java Backpack Reference Guide*, and your brain are the only information sources you can use during this exam.
- You have 50 minutes to answer all of the questions. The entire exam is worth 50 points. The point value of each question is given to help you plan your time.
- Write your answers on the exam pages. The back side of each page may be used as a scratch pad.
- The answers to the questions require only basic Java language facilities and the code given as appendix. No other classes or packages are needed.
- Questions during the exam should be about the wording of the exam only. If you have a question, raise your hand and we'll come to you. (This is less disruptive for others than if you come to us.)
- **Partial credit will be given where appropriate.**
- DON'T PANIC! If you find a question that you cannot solve right away, consider moving on and returning to it after you finish the other questions.

	Score	Max
1		10
2		15
3		5
4		10
5		10
Total		50

1. (10 points) This problem refers to the shape classes that were discussed in lectures, provided in the code handout, with two extensions, also shown in the code handout. First, a new interface `Rotateable` is defined. Second, `Rectangle` is modified to implement `Rotateable`.

For each box, circle either the word **ok** if the interaction succeeds or the word **error** if the interaction causes an error. Assume that the interactions occur in the order below and the interactions pane is *not* reset between interactions.

(a) `> Square s = new Square (new Point (1,2), 5)`
`> Rotateable r = new Rectangle (new Point (0,0), 3, 4)`
`> s.move(2,2)`

ok **error**

(b) `> r.getAngle()`

ok **error**

(c) `> ((Rectangle) r).getAngle()`

ok **error**

(d) `> s.getAngle()`

ok **error**

(e) `> Area a = r`

ok **error**

(f) `> Area a = ((Rectangle) r)`

ok **error**

(g) `> a.getArea()`

ok **error**

(h) `> r.getArea()`

ok **error**

(i) `> r instanceof Rectangle`

ok **error**

(j) `> if (r instanceof Rectangle) r.getArea();`

ok **error**

2. (15 points)

Consider the following class:

```
public class TwoIntArray {
    int[] array1;
    int[] array2;

    TwoIntArray(int[] a1, int[] a2) {
        array1 = a1; array2 = a2;
    }

    public int[] arrayDiff() {
        int[] diffArray = new int[array1.length];
        for (int i = 0; i < array1.length; i++) {
            diffArray[i] = array1[i] - array2[i];
        }
        return diffArray;
    }
}
```

(a) (3 points)

What is the result of the following interaction:

```
> int[] a = {1,2,3,4}
> int[] b = {5,5,5,5}
> TwoIntArray a = new TwoIntArray(a, b)
> a.arrayDiff()
>
```

(b) (4 points)

Explain under what circumstances the method `arrayDiff` will trigger an `IndexOutOfBoundsException`.

(c) (5 points)

Rewrite the method `arrayDiff` so that it will not trigger such an exception while still computing an appropriate output.

```
public int [] arrayDiff() {
    int minLength;
    if ( array1.length <= array2.length) {
        minLength = array1.length; }
    else {
        minLength = array2.length;
    }
    // equivalently: minLength = Math.min(array1.length, array2.length);
    int [] diffArray = new int[minLength];
    for (int i = 0; i < minLength; i++) {
        diffArray[i] = array1[i] - array2[i];
    }
    return diffArray;
}
```

(d) (3 points)

This code might also trigger another class of exception at runtime when run on certain inputs. Explain briefly.

3. (5 points) Consider the following abstract class `Counter`, which compiles without error:

```
abstract class Counter{
    int count = 0;

    public Counter(int initial) {
        count = initial;
    }

    abstract void incrementCount();
    abstract void decrementCount();

    public int getCount(){
        return count;
    }
}
```

Now consider the concrete class `ByTwoCounter`, which fails to compile:

```
class ByTwoCounter extends Counter{

    public void incrementCount(){
        count += 2;
    }

    public void decrementCount(){
        count -= 2;
    }
}
```

Change the code for `ByTwoCounter` to allow it to compile without error. (Hint: The compiler gives an error that says something about not being able to find some symbol and something about `Constructor Counter()''`.)

```
class ByTwoCounter extends Counter{
    public ByTwoCounter(int initial) {
        super(initial);
    }

    public void incrementCount(){
        count += 2;
    }

    public void decrementCount(){
        count -= 2;
    }
}
```

4. (10 points)

Here's most of the code for a simple 2D integer array class, with an incomplete method numElements that is to compute the number of elements in the array. A sample interaction with a completed version of this method is shown below.

```
class Int2DArray {
    private int[][] array;

    public Int2DArray(int[][] ip) {
        array = ip;
    }

    public int[][] getArray() {
        return array;
    }

    public int numElements() {
        // code to compute and return the total number of elements in the array.
        //Remember all rows need not be of the same dimensions
        int size = 0;
        [MISSING CODE - YOUR CODE GOES HERE]
        return size;
    }
}

> int [][] a = { {1, 2, 3, 4}, {1, 2, 3}, {1, 2}, {1}};
> Int2DArray array2d = new Int2DArray(a)
> array2d.numElements()
10
```

Complete the code for numElements.

```
public int numElements() {
    // code to compute and return the total number of elements in the arrayay array. R
    int size = 0;

    for(int i =0; i< array.length; i++)
        for(int j = 0; j < array[i].length; j++)
            size++;

    return size;
}
```

5. (10 points) Given the following class definitions, draw a picture of what the heap looks like at the **end** of each of the following interaction sequences. (You can assume that the interactions pane is reset between each sequence.)

```
class D {
    public int data;

    public D(int newInt) { data=newInt; }

    public void setData(int newInt) { data=newInt; }
}

class A {
    public D l;
    public D r;

    public A(D newL, D newR){ l=newL; r=newR; }

    public void setL(D newL){ l=newL; }
    public void setR(D newR) { r=newR;}

    public D getL() { return l; }
    public D getR() { return r; }
}
```

For example, the interaction:

```
> D a = new D(5)
```

produces this picture of the heap:



(a) Draw the heap after this interaction:

```
> D d2 = new D(1)
> A a2 = new A (d2, d2)
```

```
(b) > A a3 = new A (new D(3), new D(4))
    > A b3 = new A (a3.getL(), a3.getR())
    > a3.getR().setData(6)

(c) > D c = new D(3)
    > A a4 = new A(c, new D(4))
    > D c = new D(3)
    > A b4 = new A(c, new D(5))
```