

## CIS 120

### Midterm I — September 24, 2008

Name: \_\_\_\_\_

Student ID: \_\_\_\_\_  
(from your PennCard)

Email \_\_\_\_\_ @seas.upenn.edu

Lab:(circle one)

201	202	203	204	205	206
W 3:00 PM	W 12:00 PM	W 4:00 PM	R 10:00 AM	R 3:00 PM	R 4:00 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		10
3		12
4		9
5		9
Total		50

1. (10 points)

What result would the DrJava interpreter return after each of the following interactions? Assume that the interactions occur in the order below and the interactions pane is *not* reset between interactions. None of the interactions below produce errors.

Welcome to DrJava.

```
> int[] x = new int[] { 1, 2, 3 }
```

```
> int[][] y = new int[][] { { 1, 2, 3}, {4, 5, 6} }
```

```
> x [0]
```

*Answer:1*

```
> x [ x [0] ]
```

*Answer:2*

```
> y[0].length
```

*Answer:3*

```
> x == y[0]
```

*Answer:false*

```
> x[0] == y[0][0]
```

*Answer:true*

```
> y[0] = x;
```

```
> x[0] = 5;
```

```
> y[0][0]
```

*Answer:5*

```
> y[1][0]
```

*Answer:4*

```
> y[1][0] = y[1][2];
```

```
> y[1][0]
```

*Answer:6*

```
> y[1][2]
```

*Answer:6*

```
> x == y[0]
```

*Answer:true*

2. (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.)

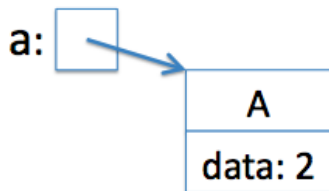
```
class A {
  private int data;
  public A(int data) {
    this.data = data;
  }
  public int getData() { return this.data; }
}
```

```
class B extends A {
  private A[] arr;
  public B(int data, A[] arr) {
    super(data);
    this.arr = arr;
  }
}
```

For example, the interaction:

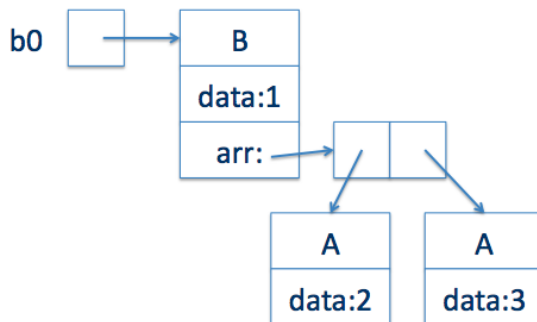
```
> A a = new A(2)
```

produces this picture of the heap:



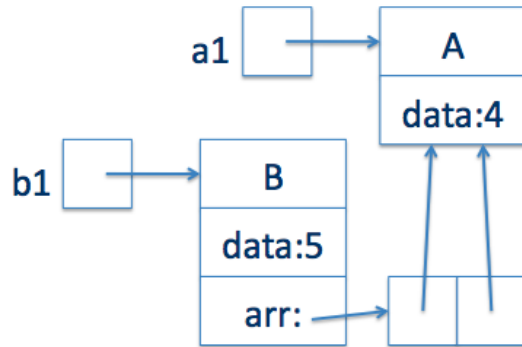
(a) Draw the heap after this interaction:

```
> B b0 = new B(1, new A[] { new A(2), new A(3) } );
```



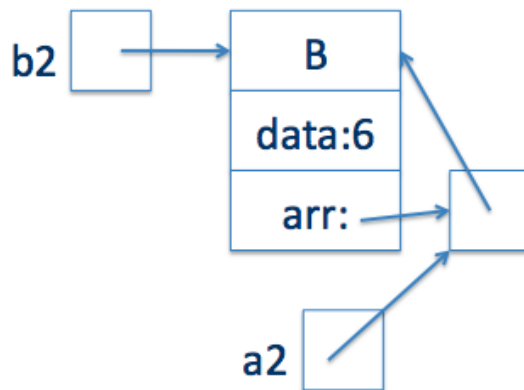
*Answer:*

```
(b) > A a1 = new A(4);
    > B b1 = new B(5, new A[] { a1, a1 });
```



*Answer:*

```
(c) > A[] a2 = new A[] { null };
    > B b2 = new B(6, a2);
    > a2[0] = b2;
```



*Answer:*

3. (12 points) In this problem, we will add some functionality to the `TicketFactory` class presented in the lectures and provided in the code handout.

(a) The `Ticket` nested class should keep track of whether the ticket has been used. Once a ticket has been used, it remembers that fact and cannot be used again. To do this, you need to implement a method

```
void use()
```

that returns `true` the first time it is invoked on a particular `Ticket`, and `false` on any subsequent invocation on that same `Ticket`. Write in the space below the `use` method preceded by declaration(s) of any instance variable(s) it may need.

- Variables: *Answer:*

```
boolean used = false;
```

- use method: *Answer:*

```
public boolean use() {  
    if (this.used) {  
        return false;  
    } else {  
        this.used=true;  
        return true;  
    }  
}
```

(b) Next, we want to keep track of how many tickets issued by a `TicketFactory` have been used. The number of tickets used should be returned by the `TicketFactory` method

```
int getUsed()
```

Write in the spaces below the declaration(s) of any instance variable(s) that may be needed, followed by the implementation of `getUsed`, followed by the revised `use` method. (The `use` method should also perform the functionality of the previous question.)

- Variables: *Answer:*

```
private int numUsed = 0;
```

- `getUsed` method: *Answer:*

```
public int getUsed() { return numUsed; }
```

- Revised `use` method: *Answer:*

```
public boolean use() {  
    if (this.used) {  
        return false;  
    } else {  
        this.used=true;  
        TicketFactory.this.used++;  
        return true;  
    }  
}
```

- (c) Finally, we want to be able to reissue a `Ticket`, meaning that a new `Ticket` is created by the same factory, we a new number, the old `Ticket` is marked as used, but the number of used tickets is not increased. You will achieve this by adding a new method to the `Ticket` class

```
public Ticket reissue()
```

Write the `reissue` method any any variable declarations you may need, if any, in the spaces below

- Variables: *Answer: None*

- reissue method: *Answer:*

```
public Ticket reissue() {  
    this.used = true;  
    return TicketFactory.this.issue();  
}
```

4. (9 points) This problem refers to the shape classes that were discussed in lectures and are provided in the code handout. 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) > Point p = new Point (1,1)  
> Circle c = new Circle (p, 2)  
> Symmetric s = c

**ok**    **error**

*Answer:ok*

(b) > c.reflect()

**ok**    **error**

*Answer:ok*

(c) > s.reflect()  
> Area a = s

**ok**    **error**

*Answer:Error: Bad types in assignment*

(d) > Area a = c

**ok**    **error**

*Answer:ok*

(e) > a.move(2,2)

**ok**    **error**

*Answer: Error: No 'move' method in 'Area' with arguments: (int, int)*

(f) > p.getArea()

**ok**    **error**

*Answer: Error: No 'getArea' method in 'Point' with arguments: ()*

(g) > (s instanceof Area)

**ok**    **error**

*Answer:ok: true*

(h) > if (p instanceof Displaceable) p.move(1,2);

**ok**    **error**

*Answer:ok*

(i) > if (s instanceof Area) s.getArea();

**ok**    **error**

*Answer:Error: No 'getArea' method in 'Symmetric' with arguments: ()*

5. (9 points) Give *three* different short examples of a *valid* definition of a class A such that the line

```
A a = new A();
```

produces a compile-time error. Explain the errors.

All examples should compile by themselves and only cause an error when used for object creation. All examples must produce a different sort of error, and so must differ in their explanations.

*Answer:*

- A may be an enumeration class.

```
public enum A {  
    NORTH, SOUTH, EAST, WEST;  
}
```

- A may be an abstract class:

```
public abstract class A {  
}
```

- A's constructor may be private

```
public class A {  
    private A() { }  
}
```

- A may not have a constructor with no arguments:

```
public class A {  
    public A(String x) { }  
}
```

- A may be a dynamic nested class that cannot be created without an enclosing instance.

```
public class B {  
    public class A {  
    }  
}
```

- A may be a package-scoped class defined in another package.

```
package other;  
class A {}
```

## TicketFactory.java

```
1 public class TicketFactory {
2     private int issued;
3     private String name;
4     public TicketFactory(String name) {
5         this.name = name;
6     }
7     public Ticket issue() { return new Ticket(this.issued++); }
8     public int getIssued() { return this.issued; }
9
10    public class Ticket {
11        private final int number;
12        private Ticket(int number) { this.number = number; }
13        public int getNumber() { return this.number; }
14        public String getFactoryName() { return TicketFactory.this.name; }
15    }
16 }
```

## Displaceable.java

```
1 public interface Displaceable {
2     public double getX();
3     public double getY();
4     public void move(double dx, double dy);
5 }
```

## Area.java

```
1 public interface Area {
2     public double getArea();
3 }
```

## Circle.java

```
1 public class Circle extends Symmetric implements Displaceable, Area {
2     private Point center;
3     private double radius;
4     public Circle(Point center, double radius) {
5         this.center = center;
6         this.radius = radius;
7     }
8     public double getX() { return this.center.getX(); }
9     public double getY() { return this.center.getY(); }
10    public void move(double dx, double dy) {
11        this.center.move(dx, dy);
12    }
13
14    public double getArea() {
15        return Math.PI * radius * radius;
16    }
17
18 }
```

## Point.java

```
1 public class Point extends Symmetric implements Displaceable {
2     private double x,y;
3     public double getX() { return this.x; }
4     public double getY() { return this.y; }
5     public void move(double dx, double dy) {
6         this.x += dx;
7         this.y += dy;
8     }
9
10    public Point(double x, double y) {
11        this.x = x;
12        this.y = y;
13    }
14 }
```

## Symmetric.java

```
1 public abstract class Symmetric
2     implements Displaceable {
3     public abstract double getX();
4     public abstract double getY();
5     public abstract void move(double dx, double dy);
6
7     public void reflect() {
8         move(-2 * getX(), -2 * getY());
9     }
10 }
```