MIDTERM EXAMINATION
CSC 112 ♦ SPRING 2011

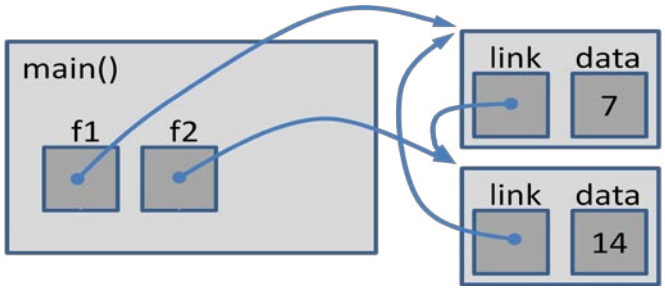
You will have 110 minutes to complete this exam. All work should be written in the exam booklet. Start with the questions that you know how to do, and try not to spend too long on any one question. Partial credit will be granted where appropriate. Good luck!

1. **Program Simulation** (16 points)

Consider the short program shown below.

```
public class Foo {

   public Foo link;

   public int data;

   public Foo(Foo link, int data) {
      this.link = link;
      this.data = data;
   }

   public void addOne() {
      data = data+1;  // Checkpoint E
   }

 public void copyUp() {
      data = link.data;
   }

   public static void main(String[] args) {
      Foo f1 = new Foo(null,7);
      Foo f2 = new Foo(f1,14);
      f1.link = f2;  //Checkpoint A

      f2.link = new Foo(null,21);
      f2.link.link = new Foo(null,28);
      f2.link.link.link = f1;  // Checkpoint B

      Foo f3 = f1;
      for (int i = 0; i < 4; i++) {
         f3.copyUp();
         f3 = f3.link;
      }  // Checkpoint C

      f1.link = f1;
      f1.copyUp();
      f2.link.link.data = 35;  // Checkpoint D

      f2.link.addOne();
   }
}
```

The diagram at right describes the state of memory at checkpoint A. Draw an equivalent diagram for the memory at checkpoint B, at checkpoint C, at checkpoint D, and at checkpoint E.



2. **Vocabulary** (12 points)

In the program below, identify a line that contains an example of each of the vocabulary terms that follow, or indicate that the requested item is not present in this program. If you do identify a line, please also indicate **which part of the line** corresponds to the term. (Some comments have been deliberately omitted or altered to avoid giving away the answers.)

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

/**
 *  A simple component that draws a color-changing circle.
 *
 *  @author Nicholas R. Howe
 *  @version CSC 112, 9 March 2011
 */
class JDisk extends JComponent {
   // The color of the disk
   private Color color;

   JDisk() {
      super();
      addMouseListener(new Clicker());
       color = Color.RED;
   }

   public Color getColor() {
       return color;
   }

   public void setColor(Color c) {
            color = c;
   }

   public void paintComponent(Graphics g) {
      g.setColor(color);
            g.fillOval(0,0,20,20);
   }

   public Dimension getMinimumSize() {
            return new Dimension(20,20);
   }

   public Dimension getPreferredSize() {
            return new Dimension(20,20);
   }

   public class Clicker extends MouseAdapter {
      public void mouseClicked(MouseEvent e) {
         color = Color.BLACK;
      }
   }
}
```

a.) A **Javadoc** comment

b.) A class **member**

c.) A local variable **declaration**

d.) An **initialization** of an instance of some class

e.) A **nested class** (sometimes called a **local class**)

f.) An **event handler**

g.) A **constructor**

h.) An **external method call** (i.e., a call to a method of some other class than that defined here)

i.) Use of a **static member** of some class.

j.) A **qualifier**

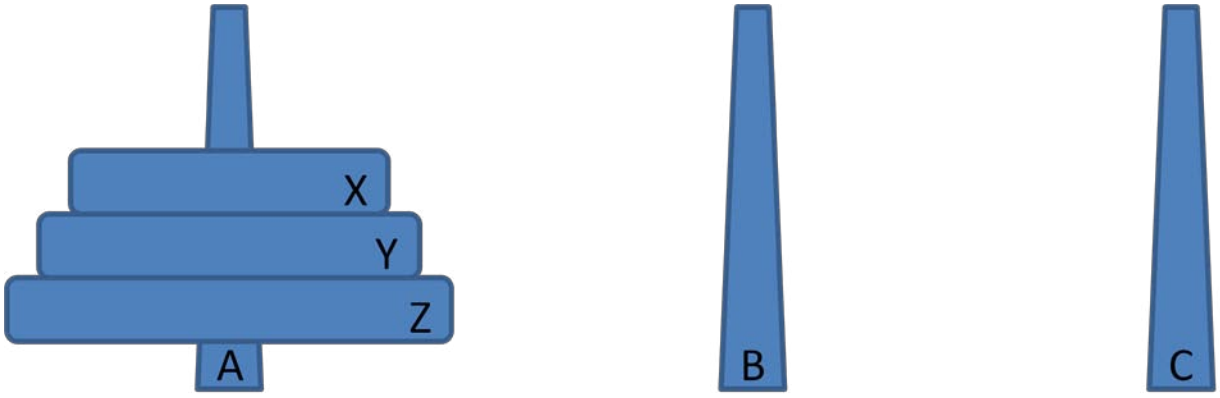
k.) A method **parameter**

l.) A method **argument**

3. **Stacks** (12 points)

There is a child's toy called the Towers of Hanoi, consisting of a central spire around which colored rings may be stacked. The rings come in various sizes and colors, and the usual stacking order is to have the largest rings at the bottom and the smallest at the top. The toy acts as a stack: the last ring added must be removed before any of the rings under it.

Suppose that you have three rings and three spires, starting in the configuration shown below (all rings on A, with the largest at the bottom and the smallest at the top). Your goal is to move all rings to B, while maintaining the constraint that a larger ring may never be placed onto a spire on top of a smaller ring. Write a set of stack operations that would achieve this objective. For example, C.push(A.pop()) would move the smallest ring from A to C. At this point, a second attempt to do C.push(A.pop()) would not be allowed, because ring Y cannot go on top of ring X.



4. **Queues** (16 points)

In most languages, the array is a basic type. However, there is no absolute requirement that a language should provide arrays as a predefined part of the language. One early hardware design for computer memory, called **delay-line memory**, used a column of mercury to store bits. Electronic signals were introduced into the mercury column at one end, and could be read some time later at the other end. (They would then be reintroduced at the head of the column so that no data were lost.) In sum, this form of memory functioned more as a queue than as an array. If computers had continued to use this form of memory, it might have been more natural to design a language around the queue. For this problem, you are to imagine that Java provides a **BasicQueue** class as a basic type, and your job is to implement class Array. To be more precise, assume that Queue provides methods **in(x)**, **out()**, and **size()** – note that **out()** returns the element removed, and **size()** tells the number of elements stored. The array datatype must implement **get(i)** and **set(i,x)**; we will write only set for this problem, plus a constructor that allocates space and sets all elements to 0. Elements of the array will be stored in the queue, with the convention that index 0 appears at the head, etc. (Of course, implementing get or set will require cycling through the elements once, but should end up with each element in its original position.) To get you started, a framework appears below.

```
/** Defines an array using a queue for storage */
public class Array {
   /** Here is the queue that will store the array values */
   private BasicQueue q;

   /** Allocate the storage and put N zeros into the array */
   public Array(int N) {
      q = new Queue();
      // FILL IN THE REST
   }

   /** Set the array element at index i to value x */
   public void set(int i, int x) {
      // FILL IN
   }
}
```

5. **Lists & Iterators** (12 points)

Suppose that a particular **list** begins with the contents shown below.

For each of the diagrams that follow, write a series of iterator method calls that will transform the preceding state into the one shown, using the list iterator provided and using the smallest number of method calls possible. The first is done for you as an example.

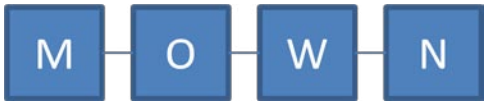
a.) Begin with ListIterator mark = list.listIterator(4);

Answer: mark.previous(); mark.remove();

b.) Begin with ListIterator mark = list.listIterator();

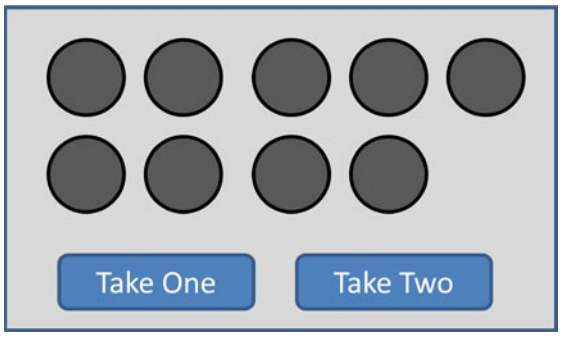
c.) Begin with ListIterator mark = list.listIterator();

d.) Begin with ListIterator mark = list.listIterator(2);



6. **Program Design & Classes** (16 points)

Suppose that you are designing a program that will play the game of Nim against the computer using a graphical interface. The game is very simple: at the beginning, a random number of stones (i.e., circles) is displayed in the window. The players take turns removing either one or two stones at a time, and the one forced to take the last stone is the loser. A mock-up of the proposed user interface appears at right.



Describe the major new classes you would expect to have to write to implement this program in a well-designed Java application. Give concise description of the role of each class, and how you would expect

it to interact with the other classes in your design. You do not need to write code, and you do not need to describe classes that Java already provides (such as JButton) although you may refer to them if necessary. Do include a description of any nested classes, and specify in which class execution would begin. If you can, relate your design to the MVC software pattern.

7. **Sorting** (16 points)

Consider the sequence of numbers below, which are to be sorted in increasing order from left to right. Simulate the sorting algorithms as specified.

10 20 15 18 5 30 0 25

a.) Insertion sort, array implementation, growing the sorted region from left to right. Show the state of the array after **every** swap.

b.) Merge sort, list implementation. Show the state of the lists after **each** merge operation.