MIDTERM EXAMINATION ANSWER KEY
CSC 112 ♦ FALL 2008

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

Which line(s) of the following program incorrectly call(s) a method from an improper context?

```
1   public class Context {
2       public static void staticMethod() {
3           nonstaticMethod();
4           Context c = new Context();
5           c.nonstaticMethod();
6       }
7
8       public void nonstaticMethod() {
9           staticMethod();
10          Context c = new Context();
11          c.staticMethod();
12      }
13
14
15      public static void main(String[] args) {
16          staticMethod();
17          nonstaticMethod();
18      }
19  }
```

Lines 3 and 17 call a nonstatic method from a static context. These are the only errors; they would be caught by the compiler. (Note: line 5 calls a nonstatic method from a static method, but the context c is specified so this is ok. Calling a static method from a nonstatic context is always ok.)

2. **Programming Style** (12 points)

Give at least three advantages or disadvantages of using classes from the Java Collections framework instead of your own custom-designed classes that perform the same function. Is there ever a situation where your own class would be preferred?

The classes in the Java Collections framework are familiar to most programmers, so they make your programs more understandable. They have already been debugged, so they reduce the possibility of error. They save development time, since you do not need to reimplement basic data structures. Although it is usually preferable to use a Collection class, there are some reasons that might prompt you not to. You may wish to implement something yourself as an educational experience. Or you may wish to make an implementation more efficient in some way, usually at the expense of safety. Finally, the Collections class may not do exactly what you want.

3. **Programming Efficiency** (16 points)

The following code defines several listener classes that print out various messages when the associated button is pressed. Rewrite these three classes as a single class that can fulfill all three roles (i.e., printing the appropriate message). Then rewrite the middle part of the **`createButtons`** method so that it works with your new class. (The full program is not shown; assume that the surrounding code to create the GUI and display it is working properly.)

Nested class definition for the listeners:

```
private class ButtonOneListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clicked button one.");
    }
}

private class ButtonTwoListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clicked button two.");
    }
}

private class ButtonThreeListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clicked button three.");
    }
}
```

Code in **`createButtons()`** to create and add the buttons and their listeners:

```
JButton button1 = new JButton("Button One");
JButton button2 = new JButton("Button Two");
JButton button3 = new JButton("Button Three");

// middle part:
ButtonOneListener listener1 = new ButtonOneListener();
ButtonTwoListener listener2 = new ButtonTwoListener();
ButtonThreeListener listener3 = new ButtonThreeListener();

frame.getContentPane().add(button1);
frame.getContentPane().add(button2);
frame.getContentPane().add(button3);
```

The solution is to create a class with a field that holds the button name. This name will be assigned by the constructor.

```
private class ButtonListener {
    private String name;
    public ButtonListener(String name) {
        this.name = name;
    }
```

```
    public void actionPerformed(ActionEvent e) {
        System.out.println("Clicked button "+name+".");
    }
}
```

The middle part needs to change slightly to provide different names to each listener instance:

```
// middle part:
ButtonOneListener listener1 = new ButtonListener("one");
ButtonTwoListener listener2 = new ButtonListener("two");
ButtonThreeListener listener3 = new ButtonListener("three");
```

4. **Generic Programming** (12 points).

Write a single generic method that would replace the two shown below.

```
static void shift(Integer[] x) {
    Integer tmp = x[0];
    for (int i = 1; i < x.length; i++) {
        x[i-1] = x[i];
    }
    x[x.length-1] = tmp;
}
static void shift(String[] x) {
    String tmp = x[0];
    for (int i = 1; i < x.length; i++) {
        x[i-1] = x[i];
    }
    x[x.length-1] = tmp;
}
```

You simply need to make the class generic on a type E, and replace Integer (or String) with E wherever it appears.

```
static <E> void shift(E[] x) {
    E tmp = x[0];
    for (int i = 1; i < x.length; i++) {
        x[i-1] = x[i];
    }
    x[x.length-1] = tmp;
}
```

5. **Program Simulation** (12 points)

Simulate execution of the following program, and show its output.

The output is shown at right. Note that Sim.x and s.x are two names for the same variable, so they will always have the same value.

```
public class Sim {
    private static int x = 3;
```

```
    public int method(int x) {
        this.x = -this.x;
        x = -x;
        return x;
    }

    public static void main(String[] args) {
        int x = 4;
        Sim s = new Sim();

        System.out.println(x);                    4
        System.out.println(Sim.x);                3
        System.out.println(s.x);                  3

        x = s.method(x);

        System.out.println(x);                    -4
        System.out.println(Sim.x);                -3
        System.out.println(s.x);                  -3

        s.x = s.method(Sim.x);

        System.out.println(x);                    -4
        System.out.println(Sim.x);                3
        System.out.println(s.x);                  3
    }
}
```

6. **Stacks and Queues** (16 points)

A **deque** is a list data structure that allows items to be added and removed from either end. Suppose that class **`Deque<E>`** is already defined and has methods **`isEmpty`**, **`addLeft`**, **`addRight`**, **`removeLeft`**, and **`removeRight`**. Write implementations of **`push`** and **`pop`** for **`Stack<E>`** implemented using the deque methods. Similarly, write implementations of **`offer`** and **`poll`** for **`Queue<E>`**. Your methods should throw an exception if the user attempts a forbidden action.

The key here is that each stack or queue method can be implemented as a single call to a deque method. The exception handling adds a further wrinkle. Methods shown below.

```
public class Stack<E> {
   Deque<E> d = new Deque<E>();

   public void push(E item) {
         d.addLeft(item);
   }
   public E pop() {
         if (d.isEmpty()) {
               throw new RuntimeException("Popped empty stack.");
         }
         return d.removeLeft();
```

```
   }
}

public class Queue<E> {
   Deque<E> d = new Deque<E>();

   public void offer(E item) {
         d.addLeft(item);
   }
   public E poll() {
         if (d.isEmpty()) {
               throw new RuntimeException("Polled empty queue.");
         }
         return d.removeRight();
   }
}
```

7. **Linked Lists** (16 points)

You are trying to implement an **`append`** method for the **`DL_IntList`** class we developed. The code below will not accomplish this. Assuming that the picture below shows the state of the data structures involved before the method below has executed, draw a picture of the result after it has executed. Then, *in words*, describe the error(s) that have been made, and what should be done instead. Do not attempt to fix the code.

```
public void append(DL_IntList suffix) {
   tail.next = suffix.head;
   tail = suffix.tail;
   suffix.head = tail;
   suffix = null;
}
```

There are two problems. First, there is a missing prev link where the two lists are joined. Second, the head and tail of the suffix were not properly set to null. The first line is fine. The second is fine, but should be delayed until the prev link has been set up. The third line could have done that, if it referred to the prev field of suffix.head. The last line should set the head and tail fields to null, not suffix itself. Final picture: