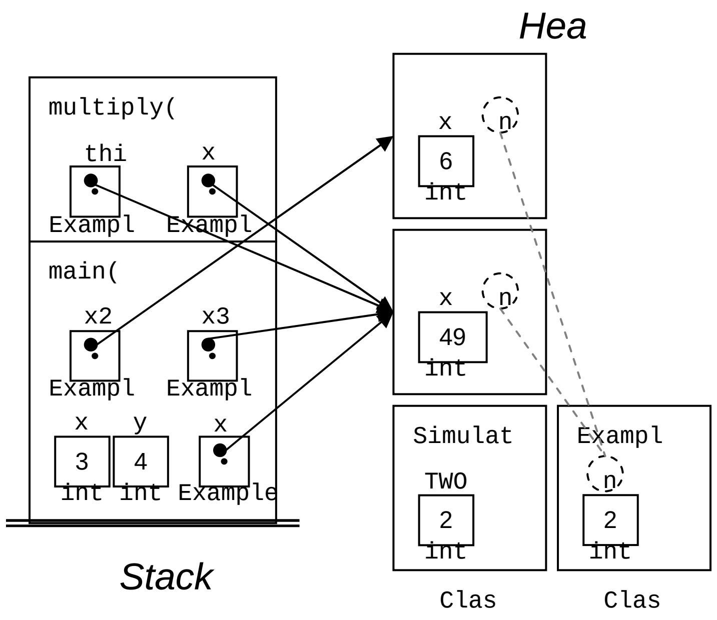
# MIDTERM EXAMINATION ANSWER KEY
# CSC 112 ♦ SPRING 2006

*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)

Simulate execution of the following program. Draw the state of memory at the point labeled Checkpoint B, showing all local variables on the stack and heap-allocated structures with their names and types. As an example of the desired format, the state at Checkpoint A is shown below.

```
public class Simulate {
    public static final int TWO = 2;

    public static void main(String[] args) {
      int x = 3;
      int y = 4;
      Example x1 = new Example(y);
      // Checkpoint A
      Example x2 = new Example(y);
      Example x3 = x1;
      x1.add(x);
      x2.add(TWO);
      x3.multiply(x1);
    }

    public static class Example {
      private static int n = 0;
      private int x;

      public Example(int x) {
          this.x = x;
          n++;
      }

      public void add(int x) {
          this.x = this.x+x;
      }

      public void multiply(Example x) {
          this.x = this.x*x.x;
          // Checkpoint B
      }
    }
}
```



## 2. **Sorting** (16 points)

Consider the array of numbers below, which are to be sorted in increasing order from left to right. Simulate the array version of the algorithms specified, and show the state of the array after **each** swap performed.

3, 1, 6, 5, 2, 4

a.) Selection sort, array implementation, growing the sorted region from left to right.

*3, 1, 6, 5, 2, 4*
*1, 3, 6, 5, 2, 4*
*1, 2, 6, 5, 3, 4*
*1, 2, 3, 5, 6, 4*
*1, 2, 3, 4, 6, 5*
*1, 2, 3, 4, 5, 6*

b.) Insertion sort, array implementation, growing the sorted region from left to right.

*3, 1, 6, 5, 2, 4*
*1, 3, 6, 5, 2, 4*
*1, 3, 5, 6, 2, 4*
*1, 3, 5, 2, 6, 4*
*1, 3, 2, 5, 6, 4*
*1, 2, 3, 5, 6, 4*
*1, 2, 3, 5, 4, 6*
*1, 2, 3, 4, 5, 6*

## 3. **Programming Style** (12 points)

Programming languages contain many features designed to eliminate the need for redundant code (i.e., code that is essentially the same except for minor differences). Give examples of two such mechanisms in Java, explaining exactly how they help avoid redundant code. Cite at least three advantages that result from eliminating redundancy in your programs.

*Loops allow you to repeat similar commands multiple times with variations controlled by the loop parameters. Functions allow you to repeat similar commands with variations controlled by the function arguments. Classes allow you to create similar data structures with variations contained within the field values. Generic classes allow you to create similar classes with variations specified by the class variables.*

*Redundancy can make programs harder to understand, develop, debug, and maintain. Eliminating redundancy will make each of these tasks easier. Furthermore, the process of eliminating redundancy often provides deeper insight into the nature of a problem.*

## 4. **GUI Building** (8 points)

The following code is intended to create a button and add a listener that will print a message to the console whenever the button is clicked. Identify two problems with the code as written that will prevent the button from working. (Assume that the surrounding code to create the frame and display it is working properly.)

Code in **`createAndShowGUI()`** to add the button and its listener:

```
JButton button = new JButton("Test button");
ButtonListener blistener = new ButtonListener();
frame.getContentPane().add(button);
```

Nested class definition for the listener:

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

*The class **`ButtonListener`** must implement the **`ActionListener`** interface.*
*The listener must be registered to the button via its **`addListener()`** method.*

5. **Java Core** (12 points)

Using a for-each loop write Java code to compute the sum of a preexisting **`ArrayList<Float>`** named **`samples`**. Declare a variable of appropriate type called **`result`** to hold your answer.

```java
float result = 0;
for (Float f : samples) {
    result += samples;
}
```

6. **Lists** (12 points)

You plan to implement a singly-linked list that will hold **`int`** values. Consider the starter code below. Write an implementation of **`Node.insertAfter()`** that will create a new **`Node`** containing the integer data and insert it into the singly linked list structure following the mark, updating all links as necessary. Account for any special cases that could arise, although you may assume that mark is not **`null`**. You may find it helpful to draw a picture.

```java
public class SL_IntList {
    private Node head;

    public SL_IntList() {
        head = null;
    }

    // Additional methods not relevant to this question

    private class Node {
       int data;
       Node next;

       private Node(int data) {
           this.data = data;
       }

       private void insertAfter(Node mark, int data) {
           Node toInsert = new Node(data);
           toInsert.next = mark.next;
           mark.next = toInsert;
       }
    }
}
```

## 7. **Generic Classes** (12 points)

Rewrite the portions of the `SL_IntList` class shown above as a generic class that may contain a singly-linked list `SL_List` of objects of an arbitrary class.

```
public class SL_IntList<E> {
    private Node head;

    public SL_IntList() {
        head = null;
    }

    // Additional methods not relevant to this question

    private class Node {
       E data;
       Node next;

       private Node(E data) {
           this.data = data;
       }

       private void insertAfter(Node mark, E data) {
           Node toInsert = new Node(data);
           toInsert.next = mark.next;
           mark.next = toInsert;
       }
    }
}
```

## 8. **Stacks and Queues** (12 points)

Suppose that you have an `IntStack s` with operations `push()` and `pop()`, and an `IntQueue q` with operations `in()` and `out()`. Suppose further that at some moment in time the contents of the stack are (listed from the top) 1, 2, and 3, while the contents of the queue (listed from head to tail) are 4, 5, and 6. Write a sequence of operations using no additional storage that would end in a configuration where the stack would contain 4, 5, and 6 (again listed starting from the top) and the queue would contain 3, 2, and 1 (again listed from head to tail). For example, `s.push(q.out())` would move the 4 from the head of the queue to the top of the stack, and `q.in(s.pop())` would move it back from the stack to the tail of the queue. Hint: slow and methodical is the ticket. Be careful!

```
q.in(s.pop())
q.in(s.pop())
q.in(s.pop())
q.in(q.out())
q.in(q.out())
s.push(q.out())
q.in(q.out())
q.in(q.out())
q.in(q.out())
q.in(q.out())
s.push(q.out())
q.in(q.out())
q.in(q.out())
q.in(q.out())
s.push(q.out())
s.push(q.out())
s.push(q.out())
q.in(s.pop())
q.in(s.pop())
```

*There are many ways to do this problem; one example is given above.*