CSC 112 MIDTERM KEY
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** (24 points). Simulate execution of the following program. What would be its output?

```
public class Sample {
    static int x;
    int y[];

    public Sample(int x) {
        y = new int[] {x};
    }

    public static void negate(int x) {
        x = -x;
    }

    public static void addOne(int y[]) {
        y[0] = y[0]+1;
    }

    public void addTwo(int y[]) {
        y[0] = y[0]+2;
    }

    public void addThree(int y[]) {
        this.y[0] = this.y[0]+3;
    }

    public static void main(String[] args) {
        int z = 7;
        int[] w = {8};
        negate(z);
        addOne(w);
        System.out.println(z);
        System.out.println(w[0]);
        addOne(new int[] {z});
        negate(w[0]);
        System.out.println(z);
        System.out.println(w[0]);

        Sample a = new Sample(2);
        Sample b = new Sample(4);
        System.out.println(b.y[0]);
        a.x++;
        a.y[0]++;
        System.out.println(b.x);
        System.out.println(b.y[0]);
        negate(a.x);
        System.out.println(a.x);

        a.y[0] = 2;
        b.y[0] = 3;
        b.addTwo(a.y);
        System.out.println(a.y[0]);
        System.out.println(b.y[0]);
        b.addThree(a.y);
        System.out.println(a.y[0]);
        System.out.println(b.y[0]);
    }
}
```

Result:

7
9
7
9
4
1
4
1
4
3
4
6

2. **Merge Sort** (12 points). Consider the list of numbers below. Draw all the intermediate lists that would be created during execution of the merge sort algorithm, as presented in class. You may use the simplified list representation shown below in your drawings.

5 → 8 → 7 → 7 → 5 → 4 → 1 → 6 → 3 → 9 → 3 → 5

Answer:

5 → 8   7 → 7   4 → 5   1 → 6   3 → 9   3 → 5

5 → 7 → 7 → 8   1 → 4 → 5 → 6   3 → 3 → 5 → 9

1 → 4 → 5 → 5 → 6 → 7 → 7 → 8

1 → 3 → 3 → 4 → 5 → 5 → 5 → 6 → 7 → 7 → 8 → 9

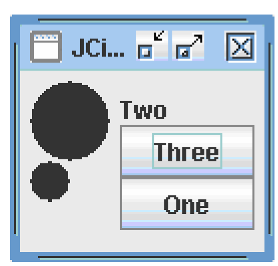
3. **Programming Practice** (12 points). Java includes a number of qualifiers that can be added to fields and methods, including `private`, `static`, and `final`. These qualifiers have specific effects within a program, are provided to help programmers achieve certain goals. Four each of the three qualifiers just mentioned, describe (a) the practical effect of including it before a field of a class, and (b) what motivation might cause a programmer to use include such a qualifier. (In other words, what does the qualifier do, and why would the programmer want to do that?)

private: means that a field may only be directly referenced from within a method of that class. This can be used to prevent programs using the class from modifying sensitive data.

static: dissociates a field from any specific instance of the class. Static fields belong to the class as a whole rather than any one instance, and their value is accessible without an instantiated instance of the class. This is helpful in avoiding multiple copies of metadata or other information that relates to the class but does not vary from instance to insyance.

final: Declares that the value of the variable will not change once it has been initialized. This is a contract enforced by the compiler. The keyword may be used to prevent accidental modification of a quantity that is supposed to remain constant.

4. **GUI Building** (12 points). Draw the component layout that would be created by the following code.




```
Container pane = frame.getContentPane();
pane.setLayout(new FlowLayout());
JPanel panelA = new JPanel();
```

```
panelA.setLayout(new GridLayout(2,1));
JPanel panelB = new JPanel();
panelB.setLayout(new BorderLayout());
panelA.add(new JCircle(20));
panelA.add(new JCircle(10));
pane.add(panelA);
panelB.add(new JButton("One"),BorderLayout.SOUTH);
panelB.add(new JLabel("Two"),BorderLayout.NORTH);
panelB.add(new JButton("Three"));
pane.add(panelB);
```

5. **Generic Programming** (12 points). Convert the following class to a generic class that represents a pair of some arbitrary class.

```
class Pair {
    private int left;
    private int right;

    public Pair(int left, int right) {
        this.left = left;
        this.right = right;
    }

    public void swap() {
        int temp = left;
        left = right;
        right = temp;
    }
}
```

```
class Pair<E> {
    private E left;
    private E right;

    public Pair(E left, E right) {
        this.left = left;
        this.right = right;
    }

    public void swap() {
        E temp = left;
        left = right;
        right = temp;
    }
}
```

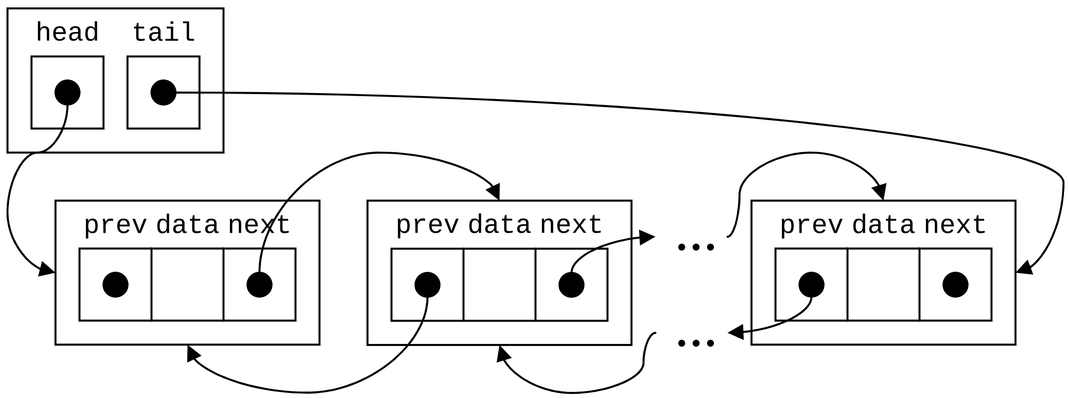
6. **Style** (16 points). This is your chance to look at programs with an instructor's eye! Below is a version of `GuessGame.java` that includes multiple instances of poor style, or style that does not adhere to the standards for this course. Identify instances of bad style below, explaining what is wrong in each case, and how to fix it. You don't need to rewrite the entire program; only enough to show how to fix style errors.

Summary of errors:
1. *Missing @version in class Javadoc comment*
2. *Missing Javadoc comment for field x*
3. *Missing @param and @guess in method2() Javadoc comment*
4. *Uninformative names: method1(), method2(), x, X*
5. *Braces misplaced in method2() if/else*
6. *Braces missing on main() if/else*
7. *Indentation wrong in several places*
8. *Three statements in main() need to be on separate lines*
9. *correct == true is redundant*

```
import java.io.*;
import java.util.Random;

/**
 *  Plays a simple guessing game with the user.
 *  The user must guess a number between 1 and 1024 in ten tries.
```

```
 *
 *  @author  Knott Mee
 *  @version  March 14, 2006  // needs version comment
 */
public class BadStyle {
    /** The number to guess */  // needs Javadoc comment
    public static int number = (int)Math.ceil(1024*Math.random());
    // also needs better name

    /** Needed to read input typed by the user */
    private static BufferedReader stdin =
    new BufferedReader(new InputStreamReader(System.in));

    /**
     *  Reads in an integer
     *
     *  @param prompt  Prompt for the user
     *  @return Integer entered
     */
    public static int method1(String prompt)
        throws NumberFormatException, IOException {
        System.out.print(prompt);
        String line = stdin.readLine();
        return Integer.parseInt(line);
    }

    // Insufficient Javadoc:  requires @param and @return
    /**
     * Gives feedback on a guess.
     *
     *  @param guess  The number guessed by the user
     *  @return T/F:  Whether the guess was correct
     */
    // change nondescriptive method name & argument name
    private static boolean respondToGuess(int guess) {
      if (guess < number) {
        System.out.println("That number is too small.");
        return false;
      } else if (guess > number) {  // Closing brace in wrong place.
        System.out.println("That number is too big.");
        return false;
      } else {  // Closing brace in wrong place.
        System.out.println("You guessed it!  Congratulations!");
        return true;
      }  // Closing brace in wrong place.
    }

    /** Runs the game. */
    public static void main(String[] args)
        throws NumberFormatException, IOException {
       int guesses = 0;
       boolean correct = false; // begin new line
       int X = 0; // begin new line
       System.out.println("I have a number between 1 and 1024.");
       do {
           guesses++;
           X = method1("Please enter a guess:  ");
           correct = method2(X);
       } while ((guesses < 10)&&(!correct));
       if (correct) {  // missing braces on if/else; simplify test
           System.out.println("You took "+guesses+" guesses.");
       } else {
           System.out.println("Out of guesses.  Better luck next time!");
       }
    }
}
```

7. **Lists** (12 points). One can implement the `list_append` operation by repeatedly taking the first element off one list and adding it to the end of the second list. This is the best we could do with the list methods defined in class, but is inefficient because the number of operations grows with the number of items in the list. If we are allowed to make arbitrary changes to the links on individual nodes, we can accomplist `list_append` in constant time. Suppose that we have two lists of the form shown in the diagram below, called `list1` and `list2`. What minimal changes to the link structure would be necessary to make `list1` contain all the elements of both lists, and `list2` empty? (Write pseudocode or Java. Assume for purposes of this question that all fields are `public` and may be accessed directly.) Make sure your answer handles any special cases, such as when either input list is empty.

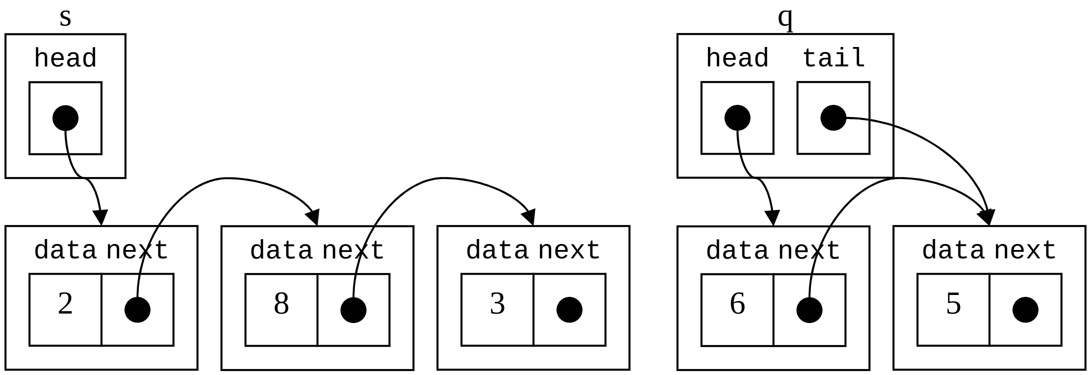


if (list2.head != null)
    list2.head.prev = list1.tail;
    if (list1.tail != null)
        list1.tail.next = list2.head;
    else
        list1.head = list2.head;
    end;
    list1.tail = list2.tail;
    list2.head = list2.tail = null;
end;

8. **Stacks and Queues** (8 points). Draw the state of the two data structures below at the end of the following sets of instructions. Assume list-based implementations as presented in class, and be sure to show all links/references.

```
Stack<Integer> s = new Stack<Integer>();
Queue<Integer> q = new Queue<Integer>();
s.push(3);
s.push(8);
s.push(5);
q.in(2);
q.in(6);
q.in(s.pop());
```

```
s.push(q.out());
```


s
head
data next
2
data next
8
data next
3
q
head
tail
data next
6
data next
5