NAME: _____

CMPSCI 187
Programming With Data Structures
Second Midterm Exam Fall 2012

D. A. M. Barrington 7 November 2012

DIRECTIONS:

- Answer the problems on the exam pages.
- There are seven questions on pages 3-11, most with multiple parts, for 100 total points. Probable scale is somewhere around A=93, C=69, but will be determined after I grade the exam.
- If you need extra space use the back of a page or the overflow page 12. The code handout on page 13 may be separated from the exam and need not be handed in.
- No books, notes, calculators, or collaboration.

1	/10
2	/15
3	/15
4	/15
5	/15
6	/10
7	/20
Total	/100

Question 1 – Java Concepts (10): Briefly explain the difference between the two concepts in each pair (2 points each): indicated:

- (a) recursive and iterative methods to compute the factorial of an integer

- (b) the class `LLNode<T>` and the class `DLLNode<T>`

- (c) serializable and unserializable classes

- (d) synchronized and unsynchronized methods

- (e) sorted and unsorted lists

Question 2 – Software Engineering (15): Briefly discuss (in English) how you would make the following modifications to the code for the specified program, with specific reference to the code (5 points each):

- (a) In Project #4, we initially determined the continent of each land square by adapting DJW's `markBlobs` method, which was recursive and thus implicitly used a stack. Could we have done this job instead with a non-recursive method using queues? Explain how we could or why we could not.

- (b) In Project #4 you used a queue to determine the centrality of a land square `X` in a `MapGrid` object, which was the distance from `X` to the farthest square on `X`'s continent. How could you use a similar queue-based method to determine the *number of squares* on `X`'s continent?

- (c) In the case study of Chapter 5, DJW use queues to simulate the physical queues of customers waiting for service. Suppose we want to change that simulation so that every ten seconds while waiting in the queue, a customer abandons the queue with a 1% probability. How could we implement this change while still using queue objects to simulate each queue?

Question 3 – Tracing Code (15): Determine the output value of the following blocks of code. In each case, assume that the `Dog` class from the page 13 handout is present. Include a brief justification of your answer.

- (a)

```
// give the return value of recurse (3, 3)
public int recurse (int j, int k) {
   if ((j == 0) && (k == 0))
      return 7;
   if (j == 0) return (1 + recurse (0, k - 1));
   return recurse (j - 1, k + 1);}
```

- (b)

```
Dog cardie = new Dog("Cardie", 5);
Dog duncan = new Dog("Duncan", 3);
LLNode<Dog> list = new LLNode<Dog>(null);
LLNode<Dog> cNode = new LLNode<Dog>(cardie);
LLNode<Dog> dNode = new LLNode<Dog>(duncan);
list.setLink(cNode);
cNode.setLink(dNode);
dNode.setLink(list);
list.setLink(list.getLink( ).getLink( ));
cNode.setLink(dNode.getLink( ));
dNode.setLink(dNode.getLink( ).getLink( ));
LLNode<Dog> cur = cNode;
for (int i = 0; i < 4; i++)
   cur = cur.getLink( );
System.out.println (cur.getInfo( ).getName( ));
```

- (c)

```
QueueInterface<String> q = new ArrayQueue<String> ( );
QueueInterface<String> r = new ArrayQueue<String> ( );
q.enqueue ("Cardie");
r.enqueue ("Duncan");
q.enqueue ("Biscuit");
r.enqueue ("Ace");
for (int i = 0; i < 3; i++) {
   String first = q.dequeue( );
   String second = r.dequeue( );
   if (first.compareTo(second) < 0) {
      q.enqueue (first);
      r.enqueue (second);}
   else {
      r.enqueue (first);
      q.enqueue (second);}}
while (!q.isEmpty( ))
   System.out.println (q.dequeue( ));
```

Question 4 – Finding Errors (15): Each of the following code fragments has a specific error that either prevents it from compiling, will cause an exception if it is run, or will cause it to produce a *clearly* unintended output. Find the error and explain what will happen (5 points each):

- (a)

```
public class DogPound {
  public DogPound ( ) {
     ListInterface<Dog> list = new ArrayUnsortedList<Dog> ( );}
  public static void main (String [ ] args) {
     Dog cardie = new Dog("Cardie", 5);
     list.add (cardie);
     System.out.println (list);}}
```

- (b)

```
public int multiply (int j, int k) {
// precondition: j and k are positive or 0
// postcondition: return value is j * k
   if (k == 0)
      return 0;
   if (j == 1)
      return k;
   return (k + multiply (j - 1, k);}
```

- (c)

```
Dog cardie = new Dog("Cardie", 5);
Dog duncan = new Dog("Duncan", 3);
LLNode<Dog> list = new LLNode<Dog>(null);
LLNode<Dog> cNode = new LLNode<Dog>(cardie);
LLNode<Dog> dNode = new LLNode<Dog>(duncan);
cNode.setLink(list);
dNode.setLink(cNode);
list = dNode;
LLNode cur = list;
while (cur.getInfo( ) != null) {
   int totalAge += cur.getInfo( ).getAge( );
   cur = cur.getLink( );}
System.out.println (totalAge);
```

Question 5 – Timing Analysis (15): Find the worst-case asymptotic running time of each block of code, as a function of the input size N (5 points each):

- (a)

```
BoundedQueueInterface<String> q = new ArrayQueue<String> (3*N);
BoundedQueueInterface<String> r = new ArrayQueue<String> (3*N);
for (int i = 0; i < N;, i++)
   q.enqueue("Cardie");
boolean flag = false;
while (!q.isEmpty( )) {
   if (flag)
      r.enqueue (q.dequeue( ));
   else r.enqueue ("Duncan");
   flag = !flag;}
```

- (b)

```
BoundedQueueInterface<String> q = new ArrayQueue<String> (N);
BoundedQueueInterface<String> r = new ArrayQueue<String> (N);
for (int i = 0; i < N;, i++)
   q.enqueue("Cardie");
while (!q.isEmpty( )) {
   while (!q.isEmpty( ))
      r.enqueue(q.dequeue( ));
   r.dequeue( );
   while (!r.isEmpty( ))
      q.enqueue(r.dequeue( ));}
```

- (c)

```
UnboundedQueueInterface<String> q = new LinkedQueue<String> ( );
UnboundedQueueInterface<String> r = new LinkedQueue<String> ( );
q.add("Cardie");
for (int i = 0; i < N; i++) {
   while (!q.isEmpty( )) {
      r.enqueue(q.dequeue( ));
      r.enqueue("Duncan");}
   while (!r.isEmpty( ))
      q.enqueue (r.dequeue( ));}
```

Question 6 – Short Code Writing (10): In Project #2 you used an explicit stack to implement a backtrack search to solve Sudoku puzzles. This search is actually much easier to implement using recursion.

Remember that you had a Board class whose objects have 9 by 9 arrays of numbers, with 0's representing unassigned cells. To make things simple and uniform for this question, we will revise the `Board` class to have the following methods, which you may assume to be already written:

```
public int getEntry (int row, int col)
// returns array entry from that row and column

public Board move (int row, int col, int value)
// returns a new Board with entry at that row and column changed to "value"

public boolean isBad ( )
// returns true if calling Board has a positive number
// twice in any row, column, or box
```

Write a method `public Board solution ( )` to be added to the `Board` class. This method should return `null` if the `Board` is not solvable (if it is not possible to replace the zeros with positive numbers to get a valid completed puzzle). If there is a solution, the method should return the first one in lexicographic order (just as `SudokuSolver` did in Project #2).

Your strategy should be to move toward a solution by finding the first unassigned cell, trying all nine possible digits in that position, and for each one using a recursive call to see whether there is a solution. There are two base cases for your recursion – one where you know that no solution is possible and one where you know that the current Board `is` the solution.

Question 7 – Long Code Writing (20): A small Massachusetts town has asked you to design a database for its dog licenses. The record of each license has three components: a license number (which should be an `int`), a `Person` object representing the owner, and a `Dog` object representing the dog to be licensed. Write the header and field declarations of the `License` class. You may assume that you have standard getters and setters for each field, and a constructor that takes a parameter to set each field. *But* if you need this class to have any *other* methods, you must write them.

You also need to write a class `LicenseDatabase` such that a `LicenseDatabase` object stores a set of `License` objects in a sorted list, where the objects are sorted by license number. You will need to make sure that there are never two licenses with the same license number in the list. Thus you must write the `equals` and `compareTo` methods for `License` that will make this work.

`LicenseDatabase` will have the following methods:

```
public boolean addNewLicense (int number, Person owner, Dog d)
// adds new license with given data to database,
// unless another license with that number is there
// return value tells whether the addition was successful

public boolean removeLicense (int number)
// removes license with that number if it is there
// return value tells whether the removal had any effect

public License findDog (Dog d)
// returns the first License object with d.equals (dog) if any, otherwise null
```

The code handout has the `ListInterface` interface from DJW, and you may assume that the class `ArraySortedList` implements this interface with the intended behavior for a sorted list. The handout also has the `Dog` class from the first exam (with an `equals` method added), and you may assume that there is a `Person` class that also has an `equals` method implemented.

Remember that DJW lists do not use iterator objects, but get the same functionality with `reset` and `getNext`. If `getNext` is called when the current position is at the end, it goes to the beginning.

OVERFLOW PAGE

Code Handout for CMPSCI 187 Second Midterm

The Dog class, slightly augmented:

```
public class Dog {
     private String name;
     private int age;

     public Dog (String newName, int newAge) {
          name = newName; age = newAge;}

     public String getName {return name;}
     public void setName (String newName) {name = newName;}
     public int getAge (return age;}
     public void setAge (int newAge) {age = newAge;}
     public boolean equals (Dog other) {
        if (!this.name.equals(other.name)) return false;
        return (this.age == other.age);}
```

Several questions use these generic interfaces from DJW:

```
public interface QueueInterface<T> {
   T dequeue( );
   boolean isEmpty ( );}

public interface BoundedQueueInterface<T> extends QueueInterface<T> {
   void enqueue (T element);
   boolean isFull( );}

public interface UnboundedQueueInterface<T> extends QueueInterface<T> {
   void enqueue (T element);
```

Question 7 uses this generic interface from DJW:

```
public interface ListInterface<T> {
     int size( );                  // number of elements in the list
     void add (T element);         // put "element" into the list
     boolean contains (T element); // is "element" in the list?
     boolean remove (T element);   // returns whether element was there
     T get (T element);            // return item equal to "element" or return null
     String toString( );           // returns string describing list
     void reset ( );               // sets current position to beginning of list
     T getNext ( );                // returns item at current position,
                                   // advances current position
     }
```