CS 61B, Spring 1999
MT3
Professor M. Clancy

Problem #1

One approach to producing debugging output is to use inheritance to create objects that print any changes to themselves. For instance, instead of a Interval in homework assignment 4, one might have a TracedInterval that inherits from Interval, in which:
(1) each constructor is overridden so that after initializing, it prints the Interval object;
(2) the extendThrough method is overridden so that after updating, it prints the new value of the Interval object.

Suppose for the purposes of this problem that the Interval class from homework assignment 4 is defined as follows:

```
public class Interval {
  private double myLeft;
  private double myRight;

  // Constructor.
  public Interval (double left, double right) . . .

  // Return this interval's left endpoint.
  public double left ( ) . . .

  // Return this interval's right endpoint.
  public double right ( ) . . .

  // Return a hash value for this interval.
  public int hashCode ( ) . . .

  // Return true when this interval represents the same interval
  // as the argument; return false otherwise.
  public boolean equals (Interval intvl) . . .

  // Return true exactly when this interval contains x;
  // return false otherwise.
  public boolean contains (double x) . . .

  // Return true when this interval overlaps the argument,
  // i.e. contains integers in common with it;
```

```
    // returns false otherwise.
    public boolean overlaps (Interval intvl) . . .

    // Extend this interval to include all the points in the argument.
    // Precondition: overlaps (intvl).
    public void extendThrough (Interval intvl) . . .
  }
```

In the space below, define the TraceInterval class described above. The main program:

```
  public static void main (String [ ] args) {
    TracedInterval intvl = new TracedInterval(1.0, 3.0);
    intvl.extendThrough(2.5, 6.0);
  }
```

should produce something like the following as output:

```
  new interval is [1.0, 3.0]
  extended interval is [1.0, 6.0]
```

Your TracedInterval class should not have any private data of its own; it should rely as much as possible on information stored and methods provided by the Interval class.

Problem #2

Recall from lab assignment 9 that the Amoeba class was defined as follows.

```
  public class Amoeba {
    . . .                              // various methods
    private String myName;             // this amoeba's name
    private Amoeba myParent;           // this amoebas parent
    private Amoeba myOlderSibling;     // this's most recently added sibling
    private Amoeba myYoungestChild;    // this's most recently added child
    }
```

On the nect page, write a public Amoeba method named amoebasNamed that, given a String as argument, returns a list of all Amoeba objects among this and its descendants whose names are the same as the argument. The sequence of the Amoeba objects within the list is not important. Assumbe that a List class has been defined that supplies methods given below.

```
  public class List {

    // Initialize and empty list.
    pubic List  ( ) . . .
```

```
    //Initialize a list containing only obj.
    public List (Object obj) . . .

    // Return the result of inserting obj at the front of this list.
    // In Scheme, this would be (cons obj this) .
    public List cons (Object obj) . . .

    // Return the result of appending the argument to this list.
    // In Scheme, this would be (append this list2) .
    public List append (List list2) . . .

    // Return true if this list is empty, false otherwise.
    // In Scheme, this would be (null? this) .
    public boolean isEmpty ( ) . . .

    // Return the first element in this list.
    // Precondition:  !isEmpty ( )
    public Object car ( ) . . .

    // Return a list containing all but the first element in this list.
    // Precondition:  !isEmpty ( )
    public List cdr ( ) . . .

    // Various private data fields.
       . . .
  }
```

You may supply auxilary methods, but do not use any of the existing Amoeba methods. In particular, do not use or reimplement the enumeration methods; your method or one of the methods it calls must be recursive.

```
Your solution to problem 2:
     public List amoebasNamed (String name) {
```

```
    }
```

Problem #3

The "8 puzzle" uses eight square blocks numbered 1 to 8 in a three-by-three tray. The goal is to slide the blocks aroudn in the tray until the blocks are in numerical order when read row by row. For example, the four moves below solve the problem for the given initial configuration.

```
   initial      after moving   after moving   after moving   after moving
configuration     7 down         5 down         2 left         3 up
-------------  -------------  -------------  -------------  -------------
| 1 | 5 | 2 |  | 1 | 5 | 2 |  | 1 |   | 2 |  | 1 | 2 |   |  | 1 | 2 | 3 |
-------------  -------------  -------------  -------------  -------------
| 4 | 7 | 3 |  | 4 |   | 3 |  | 4 | 5 | 3 |  | 4 | 5 | 3 |  | 4 | 5 |   |
-------------  -------------  -------------  -------------  -------------
| 6 |   | 8 |  | 6 | 7 | 8 |  | 6 | 7 | 8 |  | 6 | 7 | 8 |  | 6 | 7 | 8 |
-------------  -------------  -------------  -------------  -------------
```

the lst page of this exam contains the framework of a program to solve this puzzle. Its primary class, named Tray, represents the tray and its blocks with an array of the integers 0 through 8; o represents the space in the tray. Rows and columns are numbered from the top left corner of the tray:

```
          column
             0   1   2
           -------------
row 0      | 1 | 5 | 2 |
           -------------
        1  | 4 | 7 | 3 |
           -------------
        2  | 6 |   | 8 |
           -------------
```

Part a:
Supply a pseudocode body for the processUpMove method, in detail sufficient for another CS 61B student to translate it immediately into Java code (essentially one pseudocode statement for each Java statement in processUpMove).

```
    // Process the tray (if any) that results from moving
    // a block up in the arguement tray by including it
    // in the enumeration if it hasn't been seen before.
    public void processUpMove (Tray current) {
```

```
    }
```

Part b:
Given below and on the next page is most of the Collection class used by the TrayEnumeration methods. Fill in the missing code so that trays are enumerated in depth first order, that is, so that the trays reachable by moving a block in one direction from the initial tray are returned before any of the trays reachable only by moving a block in any of the other directions. Assume that a ListNode has a public Object item field and a public ListNode next field, plus a one-argument constructor that initializes item to the argument and next to null.

```
public class Collection {
    private ListNode myFirst;
    private ListNode myLast;

    // Initialize an empty collection.
    public Collection ( ) {
       myFirst = myLast = null;
    }

    // Initialize a collection with one element (the argument).
    public Collection (Object obj) {
       myFirst = myLast = new ListNode (obj);
    }

    // Add the given element to the collection.
    public void put (Object obj) {
       if (myLeft == null) {
          myLeft = myRight = new ListNode (obj);
       }
       else { //you fill this in so that trays are enumerated depth first




       }
    }

    // Remove an element from the collection,
    // and return the element removed.
    // Precondition: !isEmpty ( ) .
```

```
    public Object get ( ) {
        Object obj;
        if (myFirst == myLast) {
           obj = myFirst.item;
           myFirst = myLast = null;
        }
        else { //you fill this in so taht trays are enumerated depth first



        }
        return obj;
    }

    // Return true if teh collection is empty; return false otherwise.
    public boolean isEmpty ( ) {
       return myFirst == null;
    }
}
```
