**Wellesley College ◊ CS111 Computer Programming and Problem Solving ◊ Fall 2007**

# FINAL EXAM REVIEW PROBLEMS

The CS111 final exam is a self-scheduled exam held during the normal final exam period. It is an open book exam: you may refer to any books, your notes, and your assignments, but you may not use anyone else's notes. You may not talk to other people about the exam before or after taking it (including students who took CS111 in previous semesters), nor may you use a computer during the exam.

Here is a list of topics covered by the course that may be tested on the final exam:

- **problem solving patterns:** divide/conquer/glue, recursion, iteration (tail recursion, **`while`** loops, **`for`** loops);
- **abstraction:** method abstraction, data abstraction, abstraction barriers/contracts/APIs.
- **modularity:** constructing programs out of mix and match parts (e.g., list and array methods).
- **control structures:** sequencing, method invocation, conditionals (**`if`**/**`else`**), loops (**`while`**, **`for`**), **`return`**.
- **data structures:** objects, lists, arrays, strings, files.
- **models:** execution diagrams, object diagrams, box-and-pointer-diagrams for lists
- **Java methods:** declaration vs. invocation; parameter declaration and use; formal vs. actual parameters; scope of parameter names; **`void`** vs. non-**`void`** return types; using **`return`** to return result; invocation model (create a frame in Java execution model).
- **Java class declarations:** instance variables, class (static) variables; constructor methods, instance methods, class (static) methods; inheritance.
- **Java statements:** local variable declarations, method invocations, assignments, **`if`**/**`else`** conditional statements (**if** (*test*) *then_stmt*; **if** (*test*) *then_stmt* **else** *else_stmt*;) **`while`** loops, **`for`** loops, **`return`**; accessibility keywords (**`public`**, **`private`**)
- **Java expressions:** literals (numbers, booleans, characters, strings), variable references (instance variables [e.g., `foo.x`], array subscripts [e.g., `foo[i]`], **`this`**), constructor method invocations (**`new`**), array creation expressions (e.g. **`new`** `int[3]` and **`new`** `int [] {7,3,2}` ), non-**`void`** method invocations, primitive operator applications (arithmetic, relational, logical).
- **microworlds:** BuggleWorld, TurtleWorld, PictureWorld, Lists, Java Graphics, AnimationWorld.

Below are some problems intended to help you review material for the final exam. The problems do **not** cover all of the topics listed above, so you should also review your notes and assignments.

The problems range in difficulty. Some problems are from previous final exams. Others (with some rewriting) could be turned into reasonable final exam problems. Others are too long or complex for a final exam problem, but review material that is covered on the exam.

The problems are not in any particular order, so you should not feel compelled to do them in order. Rather, you should first work on those problems that cover material in which you think you need the most practice. To help you decide which problems to work on, each problem lists the concepts that the problem covers.

There is a handout of solutions that accompanies this collection of problems.

**Problem 1: Sales Statistics** *(Data Abstraction, Arrays, Lists, Objects, Object Diagrams)*

The management of the Decelerate Clothing Store (specializing in "clothes that slow you down") wants to track certain statistics about customer purchases. In particular, they want to track the amount of each purchase and whether it was made with cash or credit card. Later, they want to be able to calculate statistics based on this information, such as the largest cash purchase amount, the average amount of a credit card purchase, and the percentage of credit card purchases.

The management has hired Abby Stracksen of Simplistic Statistics to implement a Java program for tracking the purchase information and calculating the desired statistics. Abby begins by designing a contract for a `History` class that maintains a history of customer purchase:

### Contract for the History Class

*Constructor method for the History class*

```
public History (int maxEntries);
```
Returns a new `History` object that can store up to `maxEntries` purchase entries.

*Instance methods for the History class*

```
public void add (int amount, boolean cash);
```
Adds a new purchase entry to this history: amount is the amount of the purchase; `cash` value `true` indicates a cash purchase, `cash` value `false` indicates a credit card purchase. An attempt to add an entry is ignored if `maxEntries` entries have been stored.

```
public int size ();
```
Returns the number of entries in this history.

```
public int min (boolean cash);
```
Returns the minimum amount of a purchase made with the given `cash` value.
I.e. if `cash` is true, return the minimum purchase made with cash, otherwise return the minimum purchase made with credit card. If there are no purchases with the given `cash` value, returns the largest integer.

```
public int max (boolean cash);
```
Returns the maximum amount of a purchase made with the given `cash` value. If there are no purchases with the given `cash` value, returns the smallest integer.

```
public int average (boolean cash);
```
Returns the average amount of a purchase made with the given `cash` value. If there are no purchases with the given `cash` value, returns 0.

```
public int number (boolean cash);
```
Returns the number of purchases made with the given `cash` value.

```
public double percentage (boolean cash);
```
Returns the percentage (by number of purchases) of all purchases made with the given `cash` value. Returns 0 if there have been no purchases.

Abby has also defined the contract for a `Purchase` class that models an individual purchase:

**Contract for the Purchase class:**

*Constructor method for the Purchase class*

**`public Purchase (int amount, boolean cash);`**
Returns a new `Purchase` object that with amount `amount` and cash/credit mode `cash`. A `cash` value true indicates a cash purchase, `cash` value false indicates a credit card purchase.

*Instance methods for the Purchase class*

**`public int getAmount ();`**
Returns the amount of this purchase.

**`public void setAmount (int newAmount);`**
Sets the amount of this purchase to be `newAmount.`

**`public boolean getCash ();`**
Returns the cash/credit mode of this purchase.

**`public void setCash (boolean newCash);`**
Set the cash/credit mode of this purchase to be `newCash`.

Abby's co-worker Emil P. Mentor has begun to implement Abby's contract. Here is his implementation of the Purchase class:

```
public class Purchase {

  // Instance Variables
  private int amount;
  private boolean cash;

  // Constructor Method
  public Purchase (int i, boolean b) {
    amount = i;
    cash = b;
  }

  // Instance Methods
  public int getAmount () {
    return amount;
  }

  public void setAmount (int newAmount) {
    amount = newAmount;
  }

  public boolean getCash () {
    return cash;
  }

  public void setCash (boolean newCash) {
    cash = newCash;
  }
}
```

Emil also started to implement the `History` class, but was called away on a business trip. Here's how far he got:

```
public class History {

  // Instance Variables:
  private Purchase [ ] purchases;
  private int size;

  // Constructor Method:
  public History (int maxEntries) {
    purchases = new Purchase[maxEntries];
    size = 0;
  }

  // Instance Methods:
  public void add (int amount, boolean cash) {
    if (size < purchases.length) {
      purchases[size] = new Purchase(amount, cash);
      size = size + 1;
    }
  }

  // I still need to finish the other methods! - Emil -
}
```

**Part a.** Based on Emil's implementation, draw an object diagram that shows the result of executing the following statements. Your diagram should include the local variable `h` and all objects that are accessible from `h` via some sequence of pointers.

```
History h = new History(5);
h.add(82, false);   // credit purchase
h.add(53, true);    // cash purchase
h.add(178, false);  // credit purchase
```

**Part b.** Finish Emil's implementation by fleshing out the missing instance methods from his `History` class.

**Part c.** Your colleague Bud Lojack believes that Emil could also have written the constructor method for `Purchase` in either of the two ways below:

```
public Purchase (int amount, boolean cash) {
  amount = amount;
  cash = cash;
}

public Purchase (int a, boolean c) {
  int amount = a;
  int cash = c;
}
```

Is Bud right? Explain.

**Part d.** On his desk, Emil left the following notes about alternative implementations of the `Purchase` class:

> Many ways to implement `Purchase` instance. E.g.
> 1) As an array of two integers. Slot 0 = amount; Slot 1 = cash (use 0 for `false`, 1 for `true`).
> 2) As an integer list with two elements. First element = amount; second = cash (use 0 for `false`, 1 for `true`)
> 3) As a single positive/negative integer. Amount is the absolute value. Positive indicates cash; negative indicates credit.
> 4) As a single positive integer `n`. Amount = `n / 2`; cash = `n % 2`, where 0 is `false`, 1 is `true`.

Based on Emil's notes, provide four alternative implementations of the `Purchase` class that all satisfy the `Purchase` contract.

**Part e.** Bud Lojack thinks that Emil should have made the `amount` and `cash` instance variables of the `Purchase` class **`public`** rather than **`private`**. Explain to Bud why this is a bad idea.

**Part f.** Emil returns from his trip, and says that he had an epiphany about an alternative representation of `History` instances that does not involve `Purchase` objects. Instead, Emil thinks a history instance can be implemented as an object with three instance variables:

1. The `maxEntries` integer.
2. An integer list `cashes` holding the amounts of the cash purchases.
3. An integer list `credits` holding the amounts of the credit purchases.

**i.** Repeat part (a) using this representation of the `History` class.

**ii.** Write an alternative implementation of the `History` class based on this representation.

**Part g.** The implementations of the `History` class considered above are only two of many possible implementations. What are some other implementations? For each implementation you can think of, draw an object diagram of the example from part (a).

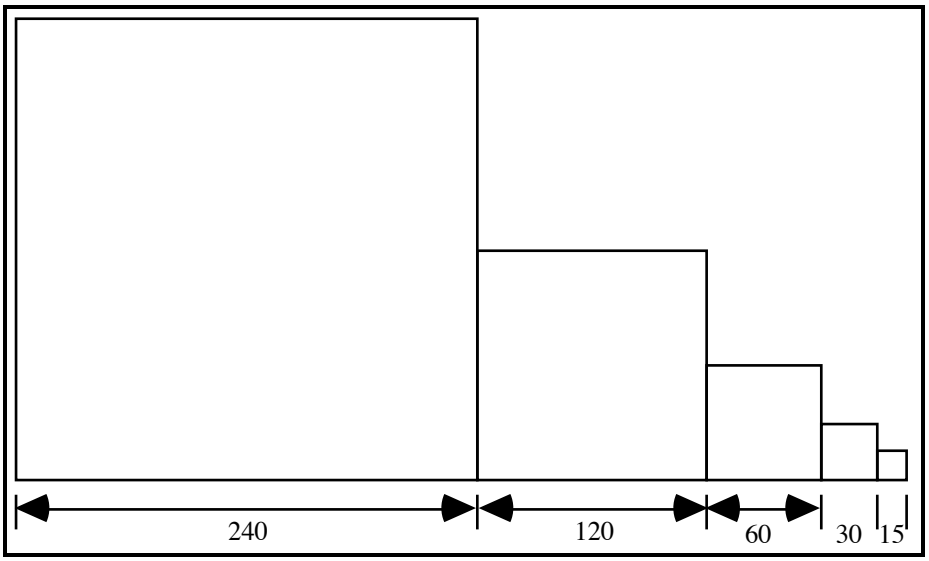
**Problem 2: Squares** *(Recursion, Iteration, TurtleWorld, BuggleWorld, PictureWorld, Java Graphics)*

Below are four parts that implement a similar problem in four different microwolds that we have studied. In all parts, you should write any auxiliary methods that simplify the definition of the requested method.

**Part a.** Write the following instance method for a `SquareTurtle` subclass of `Turtle`:

```
public void squares (int n, int len)
```

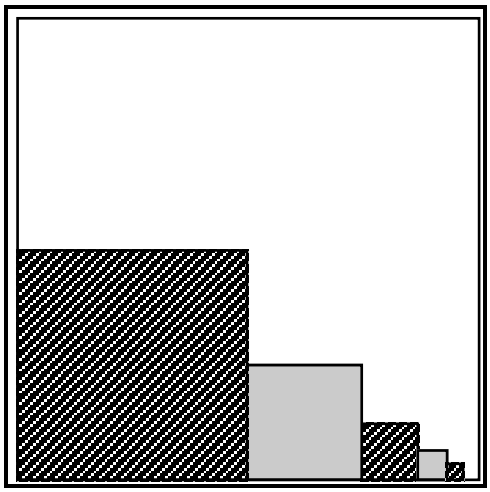
Draws `n` adjacent squares, the first of which has side length `len`, and the rest of which have a side length that is one half the side length of the previous square. After drawing the squares, the position and heading of the turtle should be the same as it was before drawing the squares. For instance, if `sara` is a `SquareTurtle` facing `EAST`, then `sara.squares(5, 240)` should draw the following picture and return `sara` to the same position and heading.




**Part b.** Write an instance method for a `SquareBuggle` subclass as `Buggle` that has the same interface as the `squares()` method from Part a in which each square of side length `len` is drawn as a `len` by `len` square of grid cells filled with bagels.

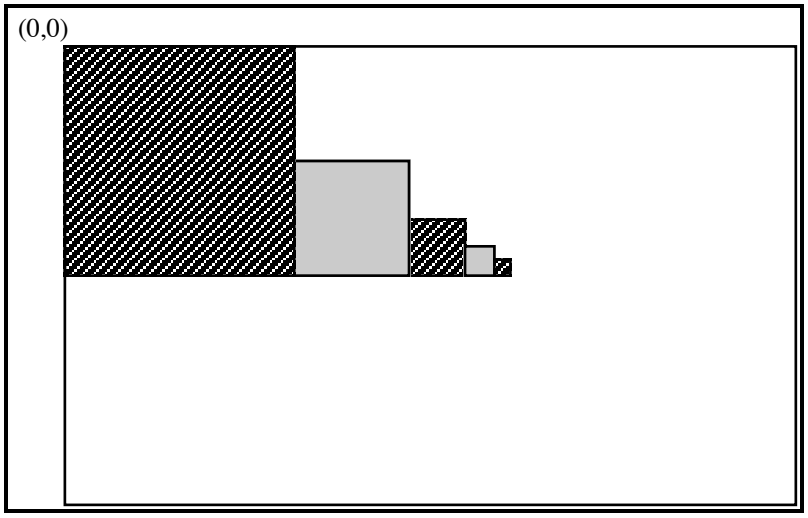
**Part c.** Write the following `PictureWorld` method:

`public Picture squares (int n, Color c1, Color c2)`
Returns a picture with `n` adjacent squares sitting at the bottom of the frame. The leftmost square should fill the lower left quadrant of the frame. Each subsequent square should be one-half the size of the square to its left. The colors of the squares should alternate between `c1` and `c2` from left to right. Assume that `public Picture patch (Color c)` returns a rectangular picture with color `c` that fills whole frame.



**Part d.** Write the following instance method squares() for a `SquareApplet` subclass of `Applet`.

`public void squares (Graphics g, int n, int len, Color c1, Color c2)`
Draws in this canvas a picture with `n` adjacent colored squares as shown below. The leftmost square has side length `len` and an upper left corner at (0,0). Each successive square has a side length that is half the side length of the square to its left. The colors of the squares should alternate between `c1` and `c2` from left to right.



**Problem 3: Greatest Common Divisor** *(Iteration)*

Wyla Lupe has been experimenting with ways to calculate the **greatest common divisor** (GCD) of two integers. The GCD of two integers A and B is the largest integer that evenly divides into both A and B. For example, the GCD of 30 and 18 is 6, the GCD of 28 and 16 is 4, and the GCD of 17 and 11 is 1.

A clever algorithm for computing GCDs was developed by Euclid in 300 B.C. (In fact, it is considered by many to be the oldest non-trivial algorithm!) Wyla has expressed Euclid's algorithm in Java as the following tail recursive `GCDTail` method. (You do **not** have to understand **why** the algorithm works!)

```
public static int GCDTail(int A, int B) {
  if (B == 0) {
    return A;
  } else {
    return GCDTail(B, A % B);
  }
}
```

Recall that `A % B` (pronounced "`A` mod `B`") calculates the remainder of `A` divided by `B`. For example, `10%3` is `1`, `10%4` is `2`, `10%5` is `0`, and `10%6` is `4`.

**Part a** In the following table, show the sequence of values that the parameters `A` and `B` take on in the iterative calculation of `GCDTail(95,60)`. **Important:** You have been provided with more rows than you need, so some rows should remain empty when you are done.

| A | B |
|---|---|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

**Part b .** In the following `GCDWhile` code skeleton, implement an alternative version of Euclid's GCD algorithm that uses a **`while`** loop to express the same iteration that is expressed by Wyla's `GCDTail` method. You may wish to introduce one or more local variables.

```
public static int GCDWhile (int A, int B) {
   // Flesh out this skeleton
}
```

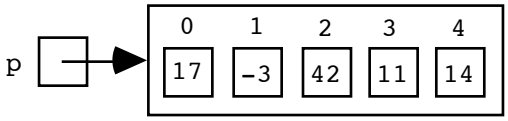
**Part c.** Would it be easy to re-express Wyla's `GCDTail` program as a **`for`** loop? **Briefly** explain your answer.

**Problem 4: Array Reversal** *(Arrays, Iteration)*

**Part a.** Implement the following `reverseCopy()` method on integer arrays:

> **`public static int [] reverseCopy (int [] a);`**
> Returns a new array that has the same length as `a` and all the elements of `a` in reverse order.

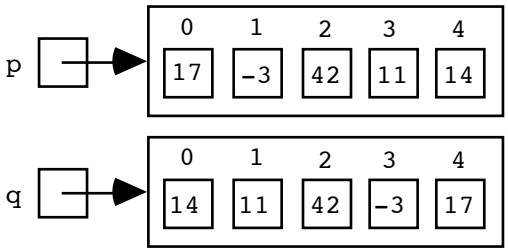
For example, suppose that `p` is the following array:




Then executing the statement

```
int [] q = reverseCopy (p)
```

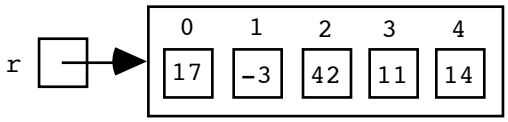
gives rise to the following diagram:




**Part b.** Implement the following `reverseInPlace()` method on integer arrays:

> **`public static void reverseInPlace (int [] a);`**
> Modifies `a` so that its elements are in the reverse of their original order. You should not create any intermediate arrays.

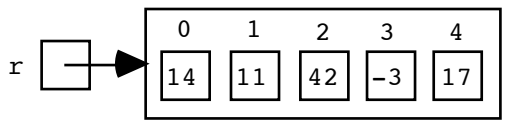
For example, suppose that `p` is the following array:




Then executing the statement

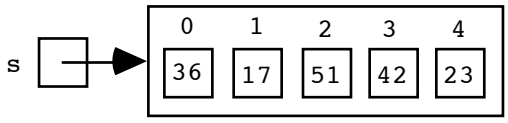
```
reverseInPlace (r)
```

changes the diagram to be:



**Problem 5: Inversions** *(Arrays, Iteration, Lists)*

In an integer array A, an inversion is defined to be a pair of indices (i, j) such that i < j and A[i] > A[j]. For instance, the following array `s` has five inversions: (0, 1), (0, 4), (2, 3), (2, 4), and (3, 4).




**Part a.** Implement the following method:

```
public static int countInversions (int [] a);
```
Returns the number of inversions in a.

For instance, `countInversions(s)` should return 5.

**Part b.** Implement the following method:

```
public static PointList listInversions (int [] a);
```
Returns a list of all the inversions in a. Each inversion (i, j) should be represented as a Point instance whose x field is i and y field is j. The order of inversions in the resulting list is immaterial.

You may assume `PointList` is like `IntList`, except stores a list of `Point`s rather than `int`s. For instance, `System.out.println(listInversions(s))` might (among many possible orderings) display:

```
[java.awt.Point[x=3,y=4], java.awt.Point[x=2,y=4], java.awt.Point[x=2,y=3],
java.awt.Point[x=0,y=4],java.awt.Point[x=0,y=1]]
```

**Problem 6: Converting Betweeen Different Forms of Iteration** *(Lists, Arrays, Iteration)*

We saw in class that iterations could be expressed as tail recursions, **`while`** loops, and **`for`** loops. Each of the following parts contains a method that uses one of these forms of iteration. For each part, write two equivalent methods that use the other two forms of iteration.

**Part a.**

```
public static int weightedSum (IntList L) {
  return weightedSumTail (L, 1, 0);
}

public static int weightedSumTail (IntList L, int index, int total) {
  if (isEmpty(L)) {
    return total;
  } else {
    return weightedSumTail(tail(L), index + 1, (index*head(L)) + total);
  }
}
```

**Part b.**

```
public static boolean isMember (int n, int [] a) {
  int i = a.length - 1;
  while ((i >= 0) && (a[i] != n)) {
    i = i - 1;
  }
  return (i >= 0); // Will only be true if n is in a.
}
```

**Part c.**

```
public static void partialSum (int [] a) {
  int sum = 0;
  for (int i = 0; i < a.length; i++) {
    sum = sum + a[i];
    a[i] = sum;
  }
}
```

**Part d.**

```
public static void squiggle (Graphics g, int x1, int y1, int x2, int y2) {
  if ((x1 > 0) || (y1 > 0) || (x2 > 0) || (y2 > 0)) {
    g.drawLine(x1, y1, x2, y2);
    squiggle(g, x2, y2, y1/4, x1*2);
  }
}
```

---

**Problem 7: Converting Betweeen Arrays and Lists** *(Lists, Arrays, Iteration)*

Implement the following two methods for converting between lists and arrays of integers:

> **`public static int [] listToArray (IntList L);`**
> Returns an array of integers whose length is the same as the length of `L` and whose elements, from low to high index, are in the same order as the elements of `L`.
>
> **`public static IntList arrayToList (int [] a);`**
> Returns a list of integers whose length is the same as the length of `a` and whose elements are in the same order as the elements of a (from low to high index).

---

**Problem 8: File Reversal** *(Tests File I/O, Iteration, Lists)*

Implement the following method:

> ```
> public static void reverseLines (String inFile, String outFile)
>                   throws IOException;
> ```
> Reads all the lines from the file named `inFile` and writes them to the file named `outFile` in reverse order. Propagates any `IOException` encountered when processing either file.

Your method should work for an input file with any number of lines. The approach that you should follow is to store all the lines from the input file in a `StringList` before you write any lines to the output file.

**Problem 9: Iterative List Reversal** *(Tests Object Diagrams, Recursion, Iteration, Lists,)*

In class we studied the following recursive method for reversing a list:

```
public static IntList reverse (IntList L) {
  if (isEmpty(L)) {
    return empty();
  } else {
    return postpend(reverse(tail(L)), head(L));
  }
}

public static IntList postpend (IntList L, int n) {
  if (isEmpty(L)) {
    return prepend(n, empty());
  } else {
    return prepend(head(L), postpend(tail(L), n));
  }
}
```

This is not an efficient way to reverse a list. Each call to `postpend()` creates a new list whose length is one more than the length of its first argument. Furthermore, postpend() is called once for each element in the list being reversed. So lots of intermediate list nodes are created that do not appear in the final result.

**Part a.** Assume that `A` is the list whose printed representation is `[1,2,3,4]`. Assuming that `reverse()` is implemented as shown above, draw object diagrams (i.e. box-and-pointer list representations) for all the list nodes created by the invocation `reverse(A)`.

**Part b.** An alternative technique for reversing a list is to follow the strategy one would use in reversing a pile of cards: form a new pile by iteratively removing the top card of the original pile and putting it on the new pile. When there are no more cards in the original pile, the new pile contains the cards in reverse order from the original pile.

Based on this idea, here is a table corresponding to an iterative reversal of the list `[1,2,3,4]`:

| list | result |
|---|---|
| [1, 2, 3, 4] | [] |
| [2, 3, 4] | [1] |
| [3, 4] | [2, 1] |
| [4] | [3, 2, 1] |
| [] | [4, 3, 2, 1] |

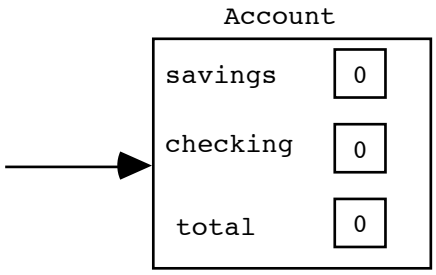
Implement this iterative list reversal strategy in a `reverse()` method in three ways: (1) using an auxiliary tail recursive `reverseTail()` method to implement the iteration; (2) using a **`while`** loop to implement the iteration; and (3) using a **`for`** loop to implement the iteration.

**Problem 10: Bank Accounts** *(Data Abstraction, Lists)*

Suppose that there is a Java class `Account` that represents bank accounts. A straightforward way to represent a simple bank account is to keep track of three integer instance variables representing, respectively, the savings account balance, the checking account balance and the total amount of money in the bank (equal to the sum of the savings and checking account balances). Assume further that the constructor method for the class `Account` has zero parameters. With this straightforward approach, the constructor method is

```
public Account ( ) {
   this.savings = 0;
   this.checking = 0;
   this.total = 0;
}
```

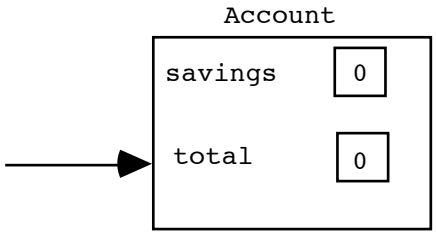
resulting in the following object diagram representation:




Below is a Java class that implements this straightforward representation.

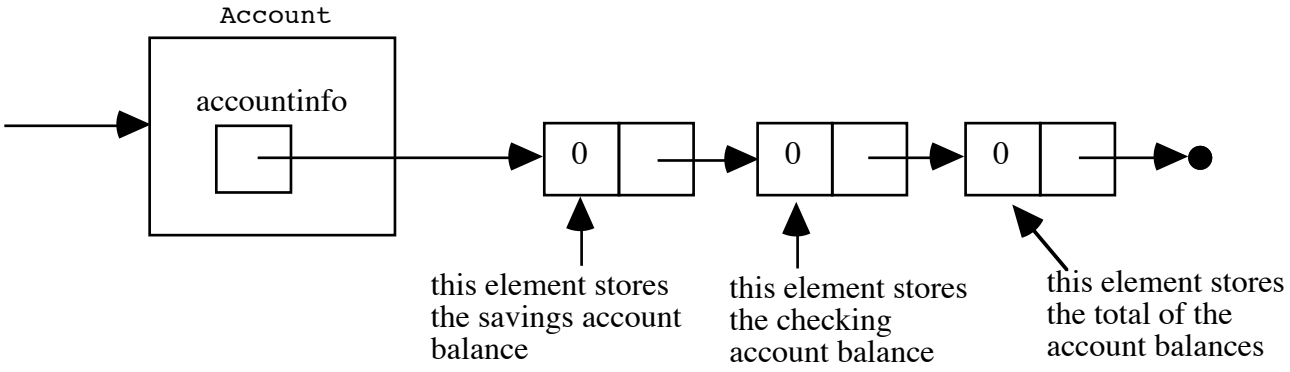
```
class Account {

  private int savings;
  private int checking;
  private int total;

  public account () {
    this.savings = 0;
    this.checking = 0;
    this.total = 0;
  }

  public int getSavings() {return this.savings;}
  public int getChecking() {return this.checking;}
  public int getTotal() {return this.total;}

  public void depositToSavings(int amountToAdd) {
    this.savings = this.savings + amountToAdd;
    this.total = this.total + amountToAdd;
  }

  public void transferFromSavingsToChecking(int transferAmount) {
    this.savings = this.savings - transferAmount;
    this.checking = this.checking + transferAmount;
  }

  public void withdrawFromChecking(int withdrawalAmount) {
    this.checking = this.checking - withdrawalAmount;
    this.total = this.total - withdrawalAmount;
  }
}
```

**Part a.** An alternative representation for bank accounts is to keep track of only two integer instance variables representing, respectively, the savings account balance and the total amount of money in the bank (equal to the sum of the savings and checking account balances). This approach results in the following object diagram representation:




Write an implementation of the `Account` class that uses this alternative representation. It is important to note that it is only the *internal representation* of the bank account that has changed. The *external interface* to the bank account itself remains the same. Additionally, every instance method should behave the same as previously.

**Part b.** Yet another representation for bank accounts is to use only one instance variable: an integer linked list that has three elements. The elements store, respectively, the savings account balance, the checking account balance and the total amount of money in the bank (equal to the sum of the savings and checking account balances). The resulting object diagram representation is:




Write an implementation of the `Account` class that uses this representation. As in Part a of this problem, it is important to note that it is only the *internal representation* of the bank account that has changed. The *external interface* to the bank account remains the same. Additionally, every instance method should behave the same as previously.

Your solution will use the `IntList` class discussed in class to represent an integer linked list. When writing your code, you may assume, that `IntList.head()` may be abbreviated by `head()` and similarly for the other `IntList` methods.

**Problem 11: Triangulators** *(Tests Objects, Animations)*

Below is a class declaration for the `Triangulator` subclass of `Sprite`. Note that the `resetState()` method is empty; you will modify this in Part b.

```
import java.awt.*;

class Triangulator extends Sprite {
  private Point p;
  private int width;
  private int height;
  private int delta;

  public Triangulator(Point p, int length, int delta) {
    this.p = p;
    width = length;
    height = length;
    this.delta = delta;
  }

  public drawState (Graphics g) {
    Polygon poly = new Polygon();
    p.addPoint(p);
    p.addPoint(new Point(p.x + width, p.y));
    p.addPoint(new Point(p.x, p.y + height));
    g.setColor(Color.blue);
    g.fillPoly(poly);
  }

  public void updateState () {
    p.x = p.x + 2*delta;
    p.y = p.y + delta;
    width = width + delta;
    height = height - delta;
  }

  public void resetState () {
    // You will flesh out this method in Part b.
  }
}
```

**Part a.** Suppose that an animation contains the single sprite created by the invocation

```
new Triangulator(new Point(20,30), 25, 10)
```

Draw the first four frames of the animation.

**Part b.** Flesh out the `resetState()` method so that it returns the sprite to its initial state. You should *not* any any new instance variables to the Triangulator class.

**Part c.** Change the representation of Triangulator by replacing the width and height instance variables by two new integer instance variables: (1) `length`, which is the initial length provided to the Triangulator constructor; and (2) `n`, which is the number of times `updateState()` has been called. Modify the constructor method and three instance methods so that your modified implementation has the same behavior as the original one.

**Problem 12: Matrices** *(Tests Data Abstraction, Arrays, Iteration, Lists, Recursion, File I/0)*

*Warning: this is a **really** big problem. However, you can learn a lot by doing even a small part of it. For instance, focus on only one of the many concrete representations considered and/or focus on only a subset of the methods in the contract.*

**Definitions and Notation**

Many engineering applications use two-dimensional rectangular collections of numbers called **matrices**. For simplicity, we will consider the case where all the numbers are integers. An ***m* by *n* integer matrix** is a collection of *m* * *n* integers conceptually arranged into *m* rows and *n* columns. Here is a sample 3 by 4 integer matrix that we will call M1:

```
 17  82  9 -60
-23  19 42  57
  6 -73 11 -14
```

We assume that the rows and columns of the matrix are indexed starting at 1. For example the first row of M1 is the four integers 17, 82, 9, and –60, and the second column of M1 contains the three integers 82, 19, and –73. If M is an m by n integer matrix, we use the notation M(i,j) to stand for the element in row i and column j, where i must be in the range 1..m and j must be in the range 1..n. The **transpose** of an m by n integer matrix M is an n by m integer matrix N such that N(i,j) is M(j,i). For example, the transpose of the sample matrix M1 is the following 4 by 3 integer matrix:

```
 17 -23   6
 82  19 -73
  9  42  11
-60  57 -14
```

The elements of an integer matrix are typically enumerated in two ways. In **row major order**, all the elements of the first row are listed first, followed by the elements of the second row, and so on. Here is the row major listing of M1:

```
17 82 9 -60 -23 19 42 57 6 -73 11 -14
```

In **column major order**, all the elements of the first column are listed first, followed by the elements of the second column, and so on. Here is the column major listing of M1:

```
17 -23 6 82 19 -73 9 42 11 -60 57 -14
```

Note that the column major listing of a matrix is the row major listing of its transpose, and vice versa.

Integer matrices are an excellent example for studying data abstraction, because there are many interesting ways to represent them in a programming language. The rest of this problem gives a contract for integer matrices and considers several concrete representations that can be used to implement that contract.

**A Java Contract for Mutable Integer Matrices**

Here we present a Java contract for a class `Matrix` of mutable integer matrices. The "mutable" means that the dimensions and contents of a matrix object may change once it is created. For simplicity, we will require that matrices are always non empty --- that is, they must always contain at least one row and one column.

*Constructor Methods*

**`public`** `Matrix (int m, int n, int [ ] elts)`
Creates a new `m` by `n` matrix whose elements are taken from `elts` in row major order. If `elts` contains too few elements, the missing elements are assumed to be 0. If `elts` contains too many elements, the extra elements are ignored. For instance, suppose that A is the integer array `{5, 1,8,3,6}`. Then `Matrix(3,2,A)` creates the matrix

```
5 1 8
3 6 0
```

and `Matrix(2,2,A)` creates the matrix

```
5 1
8 3
```

It is assumed that `m` and `n` are positive numbers. If not, the behavior of this constructor is unspecified. (This means that the implementor can do whatever is convenient in this situation.)

*Instance Methods*

**`public int`** `rows ()`
Returns the number of rows in this matrix.

**`public int`** `cols ()`
Returns the number of columns in this matrix.

**`public int`** `elementAt (int i, int j)`
Returns the integer at row i and column j in this matrix. If i or j are out of bounds, the behavior of this method is unspecified. (This means that the implementor can handle this error case in any way that's convenient.)

**`public void`** `setElementAt (int i, int j, int val)`
Changes the integer at row i and column j in this matrix to be `val`. If i or j are out of bounds, the behavior of this method is unspecified.

**`public void`** `transpose ()`
Transposes this matrix.

**`public int`** `[] toArray ()`
Returns an array of the elements in the matrix in row major order. Changes to this array should *not* affect the matrix.

**`public`** `IntList toList ()`
Returns a list of the elements in the matrix in row major order.

**`public`** `String toString ()`
For an `m` by `n` matrix, returns a string with `m` lines in which the *i*th line (starting with *i* = 1) has the `n` elements of *i*th row of the matrix, separated by spaces.

**Concrete Representations of Integer Matrices**

Here we consider a few of the many concrete representations for mutable integer matrices.

(a) *Array of arrays*. One representation that jumps to most people's minds is representing the matrix as an object with a single instance variable `elts` that contains an array of arrays of integers. Each inner array represents one row of the matrix and the outer array is a collection of these rows. If an *m* by *n* matrix M is represented in this manner, then M(i,j) is `elts[i-1,j-1]`. (The subtraction of one is necessary because array indices start at 0 while our matrix indices start at 1.) Note that *m* and *n* don't need to be explicitly stored in the object because they can be determined from `elts`.

(b) *Row major array*. A popular representation of a matrix is as an object with three instance variables:
(1) `rows`, the number of rows in the matrix;
(2) `cols`, the number of columns in the matrix;
(3) `elts`, an integer array containing the `rows` * `cols` elements of the array in row major order.
In this representation, M(i,j) is `elts[(cols * (i-1)) + (j-1)]`. Note that in this representation the number of rows and columns needs to be stored explicitly, since both cannot be determined simply from the length of `elts`.

(c) *Column major array*. This representation is just like the row major array representation except that `elts` lists the integers in column major order. In this representation, M(i,j) is `elts[(rows * (j-1)) + (i-1)]`.

(d) *Flexible array*. In the flexible array representation, the elements are stored in an integer array that may be either in row major order or column major order. As in representations (b) and (c), there are instance variables named `rows`, `cols`, and `elts`, but there is also a fourth boolean instance variable `isRowMajor` that indicates whether the array elements are in row major order or column major order. In this representation, it is very easy to transpose a matrix: just negate `isRowMajor` and swap the values of `rows` and `cols`; it is not necessary to modify `elts`.

(e) *Array of lists*. In this representation, the matrix is represented as an array of rows, where the elements of each row are stored in an IntList. This is not an efficient way to represent a matrix: looking up an element in a row requires searching through a list; setting an element requires copying list structure; and transposition requires creating lots of new list structure. However, we consider it as a possible representation for two reasons: (1) to emphasize that data abstractions can be implemented by many different concrete representations; and (2) to provide a context for doing some list manipulation.

There are many more concrete representations than the ones considered above. For example:

- An array of lists, where the lists represent columns rather than rows;
- A list of arrays, where the arrays represent lists or columns;
- A list of lists, where the inner lists represent rows or columns.

Additionally, the `isRowMajor` boolean used in the flexible array representation could be used in many other representations as well to simplify transposition

**Matrix Problems**

**Part a.** Consider the sample 2 by 3 matrix M2:

```
8 3 5
1 9 6
```

For each of the five concrete representations (a) through (e), draw an object diagram for the matrix M2 using that representation.

**Part b.** For each of the five concrete representations (a) through (e), define a class declaration for `Matrix` that uses that representation. Define whatever auxiliary methods you find helpful. (Auxiliary methods are especially helpful for manipulating the lists representation (e).)

**Part c.** We can extend the `Matrix` contract by adding some new constructor and instance methods. Extend the implementations in Part b by adding the following three constructor methods and one instance method:

**`public`** `Matrix (int [ ] [ ] elts)`
Creates a new `m` by `n` matrix where m is the length of `elts` and n is the length of the smallest element array of `elts` (this last specification is necessary because not all element arrays of `elts` may have the same size.) The element at index (i,j) in the resulting matrix is `elts[i+1,j+1]`. (The addition of one is necessary because matrix indices start at 1 while array indices start at 0.)

**`public`** `Matrix (int m, int n, IntList elts)`
Creates a new `m` by `n` matrix whose elements are taken from `elts` in row major order. If `elts` contains too few elements, the missing elements are assumed to be 0. If `elts` contains too many elements, the extra elements are ignored.

**`public`** `Matrix (String inFile)` **`throws`** `IOException`
Assume that `inFile` is the name of a file whose first line contains the two integers `m` and `n` (separated by a space); this line should be followed by `m` × `n` space-separated integers arranged on any number of lines. If the file contains too few elements, the missing elements are assumed to be 0. Creates a new `m` by `n` matrix whose elements are specified by the file contents. If the file contains too many elements the extra elements are ignored. For example, the 2x3 matrix example from above could be represented by any of the following files:

```
2 3
8 3 5 1 9 6
```

```
2 3
8 3 5
1 9 6
```

```
2 3
8 3
5 1
9 6
```

```
2 3
8
3 5 1
9 6
```

**`public void`** `(String outFile)` **`throws`** `IOException`
Writes to the file named `outFile` a representation of the matrix that can be read by the previous constructor method.