Read and fill in this page now.
Do NOT turn the page until you are told to do so

Your name: _____

Your login name: _____

Your discussion section day and time: _____

Your discussion section t.a.: _____

Name of the person sitting to your left: _____

Name of the person sitting to your right: _____

Problem 0: _____ Total : _____ / 50

Problem 1: _____ _____

Problem 2: _____

Problem 3: _____

Problem 4: _____ _____ _____

This is an open-book test. You have approximately one hour and fifty minutes to complete it. You may consult any books, notes, or other paper-based inanimate objects available to you. To avoid confusion, read the problems carefully. If you find it hard to understand a problem, ask us to explain it. If you have a question during the test, please come to the front or the side of the room to ask it.

This exam comprises 12.5% of the points on which your final grade will be based. Partial credit may be given for wrong answers. Your exam should contain five problems (numbered 0 through 4) on eleven pages. Please write your answers in the spaces provided in the test; in particular, we will not grade anything on the back of an exam page unless we are clearly told on the front of the page to look there. A separate supplementary document will also be handed out at the exam.

You need not rewrite a function that appears in any of the textbooks or handouts; merely cite the page in the textbook, the reading material, or the handout in which the function appears. In problems where we supply code or functions for you, you may lose points for not using the supplied code or functions. Some students are taking this exam late. Please do not talk to them, mail them information, or post anything about the exam to news groups or discussion forums until after Friday.

Relax—this exam is not worth having heart failure about.

Problem 0 (1 point)

Put your login name on each page. Also make sure you have provided the information requested on the front page.

Problem 1 (12 points)

Part a

For each statement indicate whether it is true or false (please spell out the word "true" and "false"; it is sometimes hard to distinguish a T from an F).

1. A child class can call private methods inherited from its parent class. **F**
2. A static class variable cannot be changed. **F**
3. Once an exception is thrown, the program necessarily terminates. **F**
4. The default behavior of `equals` is to compare references. **T**
5. The default scope of class variables and class methods is `public`. **F**
6. A subclass can extend from multiple abstract base classes. **F**
7. A single subclass can both extend a base class and implement an interface. **T**
8. Interfaces may have fields. **T**
9. An object can be assigned to a variable of its base class without casting. **T**
10. It is always illegal to define two class methods with the same name. **F**

Part b

Consider the following classes (Dog is in Dog.java, Dalmatian is in Dalmation.java).

```
public class Dog {
      protected String name;
      public Dog (String aName) {
            name = aName;
      }
}

public class Dalmatian extends Dog {
      private int spots;
      public Dalmatian (String aName, int nSpots) {
            super(aName);
            spots = nSpots;
      }

      public int countSpots () {
            return spots;
      }
}
```

The following statements are typed in sequence in the `main` method for `Dog`. For each statement, indicate whether it is OK or will it result in an error. For each error, say if it is a compile time or runtime error and provide a brief explanation.

Statement	Error or OK?	(If error) compile-time or run-time?	(If error) Explanation
`Dog d1 = new Dalmatian("lucky", 101);`	**OK**		
`d1.countSpots();`	**ERROR**	**Compile-time**	**countSpots() is not defined for Dog class**
`Dalmatian d2 = (Dalmatian)(new Dog("pongo"));`	**ERRRO**	**Runtime**	**Casting an object of super class to subclass**

Problem 2 (14 points)

Consider the following class and interface definitions contained in the default package:

```
abstract class Foo {
      static int x = 2;

      public void f() {
            System.out.println("foo");
      }
}
```

```
interface Bar {
      static int y = 3;

      void f();
}
```

```
public class Puzzles extends Foo implements Bar {
      int z = 1;

      public void f() {
            System.out.println("Puzzles");
      }

      public void superF() {
            super.f();
      }

      private void oddOrEven(int x){
            //Finds the remainder of x/2.
            x = x % 2;
      }

      private int bar(int x){
            x = x * 2;
            return x;
      }

      public void mystery() {
            int x = 0;
            int y = 5;
            x = x + this.x;
            y = y + this.y;
            System.out.println("Mystery 1: x = " + x);
            System.out.println("Mystery 2: y = " + y);
            x = bar(x);
            oddOrEven(y);
            oddOrEven(z);
            System.out.println("Mystery 3: x + y + z = " + (x + y + z));
      }
```

```
    public void mystery2() {
        int[][] y = new int[3][3];
        System.out.print("Mystery 4: ");
        for(int n = 0; n < 3; n++){
            for(int m = 0; m < n; m++){
                y[n][m] = n + m;
                System.out.print(y[n][m]);
            }
        }
        System.out.println();
        System.out.print("Mystery 5: ");
        for(int n = 0; n < 3; n++){
            for(int m = 0; m < 3; m++){
                System.out.print(y[n][m]);
            }
        }
        System.out.println();
    }

    public static void main(String[] args){
        Bar b = new Puzzles();
        System.out.print("f prints: ");
        b.f();
        System.out.print("superF prints: ");
        ((Puzzles)b).superF();
        ((Puzzles)b).mystery();
        ((Puzzles)b).mystery2();
    }
}
```

Complete the following table showing the output that would be produced on the console when the `main` method defined above is executed:

`f prints:`	**puzzles**
`superF prints:`	**foo**
`Mystery 1:`	**x = 2**
`Mystery 2:`	**y = 8**
`Mystery 3:`	**x + y +z = 13**
`Mystery 4:`	**123**
`Mystery 5:`	**000100230**

Problem 3 (8 points)

This question will be based on the `Account` class from lab 2 (see the supplemental hand out). You will provide code to allow iteration over ancestor accounts. Your code should enable the following function to work correctly:

```
public static double maximumWithdrawal(Account a) {
      a.initIterator();
      double total = 0.0;
      while(a.hasNext())
            total += a.next().balance();
      return total;
}
```

This function should return the maximum amount that can be withdrawn from the account, including overdraft protection, i.e. the maximum amount `x` such that calling `withdraw(x)` on the account returns true (recall that if `balance < x`, the difference is withdrawn from `parentAccount`, if it exists).

The methods you will need to implement are:

`void initIterator()`**:** initializes the iteration to the first account, namely the one on which `initIterator()` is called.

`boolean hasNext()`**:** returns a `boolean` indicating whether there are any more accounts to iterate over.

`Account next()`**:** returns the next `Account` in the sequence.

You should also add whatever instance variables are necessary to record the state of the iterator. Your code should not interfere with the ordinary working of an `Account` object (it should not change its state).

```
//Instance variables for the iteration

Account next;
//initialize an iteration of the ancestor accounts of this account
public void initIterator() {

      next = this;

}

//returns true if there are more accounts to be returned, false otherwise
public boolean hasNext() {

      if (next == null)

            return false;

      else

            return true;



}

//return the next Account in the iteration sequence.
//precondition: hasNext( );
public Account next() {

      Account toReturn = next;

      next = this.parentAccount;

      return toReturn;

}
```

Problem 4 (15 Points)

Consider the following definition of a `IntList` class:

```
public class IntList {
      private class IntListNode {
            public int item;
            public IntListNode next;
      }

      private IntListNode myHead;
      private IntListNode myTail;
      private int size;
}
```

Part a

Write a `rotateRight` method of the `IntList` class that, given a nonnegative int k, rotates the nodes in this list circularly to the right by k positions. That is, it should unlink the last k nodes in the list and move them to the front of the list, as shown below. Assume that, if N is the number of elements in this list, then k < N. The list should be manipulated destructively: you should not create any new `IntListNode` and `IntList` objects. Do not use any other methods other than those you explicitly define yourself.

Examples:

k	list before rotation	list after rotation
0	(3 2 9 0)	(3 2 9 0)
2	(3 2 9 0)	(9 0 3 2)
3	(3 2 9 0)	(2 9 0 3)

Solution:

```
public void rotateRight (int n) {

      //write your solution below
      if (n == 0) return;

      myTail.next = myHead;

      for (int i = 0; i < size - n; i++) {
            myHead = myHead.next;
            myTail = myTail.next;
      }

      myTail.next = null;
}
```

A less efficient different solution is to rotate the list to the right once using an inner loop and then apply an outer loop to achieve the desired number of rotations

Part b

Write a `recurReverse` method of the `IntList` class that reverses the order of the elements using recursion. The list should be manipulated destructively. For example, calling `recurReverse` on the `IntList` (3 2 9 0) should change the list to (0 9 2 3). You should not create any new `IntListNode` and `IntList` objects. Do not use any other methods other than those you explicitly define yourself.

Solution a:

```
public void recurReverse() {

      //write your solution below

      IntListNode newHead = myTail;
      myTail = reverseHelper(myHead);
      myHead = newHead;

}

private static IntListNode reverseHelper(IntListNode node) {
      if (node == null) return null;

      if (node.next == null) {
            return node;
      } else {
            IntListNode n = reverseHelper(node.next);
            n.next = node;
            node.next = null;
            return node;
      }
}
```

Solution b :

```
void recurReverse() {
      if(myHead == null || myHead.next == null)
            return;
      IntListNode temp = myHead;
      myHead = myHead.next;
      recurReverse();
      myTail.next = temp;
      myTail = myTail.next;
      myTail.next = null;
}
```

Part c

Given an integer list we are interested in *runs* of values with the same parity (negative or non-negative). A *run* is a consecutive sequence of negative values that's as long as possible, or a consecutive sequence of non-negative values that's as long as possible (0 is non-negative). For instance, there are three *runs* in the sequence -3 -21 0 15 0 -38 -9 -22: a run of two negatives (-3 -21) starting at position 0, a run of three non-negatives starting at position 2 (0 15 0), and a run of three negatives starting at position 5 (-38 -9 -22) . Write a `countRuns` method of the `IntList` class that returns the number of runs in an `IntList` object. If the `IntList` contains no elements, the number of runs is defined to be zero.

```
public int countRuns() {

      //write your solution here
      int count = 0, prev = currVal = 0;
      IntListNode curr = myHead;
      //empty case
      if (curr == null) return count;
      else { //at least one run
            prev = currVal = curr.item;
            curr = curr.next;
            count++;
      }

      for (; curr != null; curr = curr.next) {
            currVal = curr.item;
            if (prev == 0 && currVal < 0) count++; //switch from 0 to -
            else if (currVal == 0 && prev < 0) count++; //switch from - to 0
            else if (prev * currVal < 0) count++; //switch from + to -
            prev = currVal;
      }
      return count;

}
```

```
public class Account {


    // This class represents a bank account whose current
    // balance is a nonnegative amount in US dollars.

    // Initialize an account with the given balance.
    public Account (int balance) {
        . . .
    }

    // Initialize an account with the given balance and a parent account
    public Account (int balance, Account parent) {
        . . .
    }

    // Add the given amount to the account.
    public void deposit (int amount) {
       . . .
    }

    // Subtract the given amount from the account
    // if possible.  If the amount > balance, then the difference
    // is drawn from the parentAccount, if it exists
    public void withdraw (int amount) {
        . . .
    }

    // Return the number of dollars in the account.
    public int balance ( ) {
        . . .
    }

    private int myBalance;
    private Account parentAccount;
}
```