CS61B - Exam 2
March 17, 2003

Problem 0 (1 point): Identification
Fill out you name, you neighbor's names and other information in the grid on the solution sheet.

Problem 1: (4 points) Inheritance

Consider the following example of a program that uses inheritance.

```
abstract class Pet{
public pet() { System.out.println("Pet.Pet"); }
public void chase (Pet p) {
System.out.println("Pet.chase"); }
abstract public void snarl (Pet p);
}

class Cat extends Pet {
public Cat () { System.out.println("Cat.Cat"); }
public void snarl (Pet p) {
System.out.println("Cat.snarl"); }
}

class Dog extends Pet {
public Dog() {
     this(0);
     System.out.println("Dog.Dog");
     }
     public Dog (int x) {
System.out.println("Dog.Dog(int)"); }
     public void chase (Cat c) {
System.out.println("Dog.chase"); }
     public void snarl (Pet p) {
System.out.println("Dog.snarl"); }
}

public class Animals {
     public static void main (String [] args) {
            Pet p1 = new Cat();
     Pet p2 = new Dog();
     Dog d = new Dog();
     Cat c = new Cat();

     p2.chase(c);   // Part B
```

```
        p2.snarl(p1);  // Part C
        d.chase(c);    // Part D
        }
}
```

Part A: How many times will the Pet constructor be executed in the above main method?

Part B-D: What will each of these lines print? (Your answers may include: “Compile time error” or “runtime exception” if appropriate).

Remember to write your answer on the separate

Problem 2: (6 points) Lists

Consider a simplified version of the Ring class from Lab6, with code in the appendix. You are to add a “get” method to the class that will return the item at a given position. In general, your implementation may run in time O(n) for a list of size n. however, you must optimize for a common usage pattern I which the user calls the get operations in order, starting at the beginning of the list and moving forward by a constant increment; in this case each call to get should run in O(1). For example, given a Ring r of size n, the following loop:

```
for(int i=0; i < r.size(); i++) {
        System.out.println(r.get(i));
}
```

should run in O(n). A loop that runs for high to low may still run in time $O(n^2)$. Your solution may add variables to the Ring class and may include a small number of additional statements existing methods.

Your solutions should modify as few existing methods as possible. Overly complex solutions will not receive full credit, and solutions that do not meet the complexity requirements will receive no partial credit.

Part A) What variables, if any, do you wish to add to the Ring class?

Part B) What invariants are there on your new variables, if any? (Your solution should be precise enough that another CS61B student could understand the constraint.)

Part C) What changes to existing methods need to be made? You should tell us what method(s) are being changed, and what code is being added. Ignore repOk for this problem.

Part D) Give an implementation of the get method.

```
/** Get an element of this list.
*   @param pos is a position in the list.
*   @return the element in the given position in this Ring.
*   @exception IndexOutOfBoundsException if pos less than 0
*                or greater than size()-1
*/
public Object get (int pos)
{
     FILL IN
}
```

Problem 3: (5 points)
In homework 4 you stored a set of Squares in a linked list. The code to handle clicks in a Region was linear in the number of square in the Region. Assume that the number of Squares, s, can grow quite large as can the size of each Region, r. Use a hash table in each region to store the set of Squares, so that given a click at position(x,y), you can find all the clicked-on squares in time O(c), where c is the number of clicked-on squares. (The code for the Squares class is included at the end of the exam.)

Part A) What will you store in the hash table? Describe both keys and values. (You need to tell us how the keys and values relate to the Squares. Don't just give types.)

Part B) What is the Big-Oh complexity of adding a k-by-k square to the hash table using your strategy, assuming squares can grow arbitrarily?

Part C) If you assume that all squares are at the 20-by-20, as they were in the homework, how can you modify your strategy to ensure that at most 4 entries will be inserted into the hash table when adding a square?

Problem 4: (4 points) Enumerations

Consider the two Enumerations, NumbersFrom and Mystery, defined below.

```
class NumbersFrom implements Enumeration {
     NumbersFrom (int start) {
          myCurrent = start – 1;
     }

     public Object nextElement() {
          return true;
     }

     public Object nextElement () {
```

```
          myCurrent += 1;
          return new Integer(myCurrent);
     }

     private int myCurrent;
}

class Mystery implements Enumeration {

     Mystery (Enumeration e, int v)
     {
          myValue = v;
          myEnum = e;
     }

     public Object nextElement() {
          Integer possible = (Integer)
myEnum.nextElement();
          while(possible.intValue() % nyValue == 0) {
               possible = (Integer) myEnum.nextElement();
          }
     return possible;
     }
     private int myValue;
     private Enumeration myEnum;
}
```

Part A) What will the following loop print?

```
int stopAt = 10;
Enumeration mystery = new Mystery ( new NumbersFrom(1), 2);
while (mystery.hasMoreElements()) {

     int next = ((Integer)
mystery.nextElement()).intValue();
     if (next > stopAt) {
          break;
     }
     System.out.print(next + " ");
}
```

Part B-D) Define an Enumeration called Primes that will produce prime numbers starting with 2. For example, if Primes were used in place of Mystery in part A, the loop would print the prime numbers less than stopAt.

```
public class Primes () {

     public Primes() {
          FILL IN FOR PART C
     }
     public boolesn hasMoreElements() {
          return true;
     }

     public Object nextElement() {
     FILL IN FOR PART D
     }

     /* Internal variables */
     FILL IN FOR PART B
}
```