Final, CS1110, Spring 2011
Name (last name ALL CAPS): **NetID**:

Grades for the final will be posted on the CMS as soon as they are completed. Course grades will be posted next week. You can look at your final when you return in the fall.

HAVE A GOOD SUMMER!

Submit all regrade requests by 8PM TONIGHT. Use the CMS where possible. Regrades for prelims will not be considered.

You have 2.5 hours to complete the questions in this exam, which are numbered 0..7. Please glance through the whole exam before starting. The exam is worth 100 points.

Question 0.	_________	(out of 02)
Question 1.	_________	(out of 15)
Question 2.	_________	(out of 15)
Question 3.	_________	(out of 13)
Question 4.	_________	(out of 13)
Question 5.	_________	(out of 14)
Question 6.	_________	(out of 14)
Question 7.	_________	(out of 14)
Total	___________	(out of 100)

Question 0 (2 points). Print your name and Cornell **net id** at the top of each page. Please make them legible.

The first few questions deal with an abstract class `Repository` and subclasses of it, for maintaining information about libraries. When answering these questions, read carefully, look at the pertinent class invariants and specifications of methods, and deal with one issue at a time. Classes `Repository` and `Book` appear below. They will be referred to later.

```
/** An instance represents part of a library of books,
     CDs, etc. perhaps kept in electronic form but
      modeling a real library*/
public abstract class Repository {
   private int cap; // max no. of books this repository
   /** Constructor: an instance that can hold c items */
   public Repository(int c) {
      cap= c;
   }
   /** = no. of items this repository can hold */
   public int capacity() {
      return cap;
   }
   /** = no. of items in this repository */
   public abstract int noItems();
}
```

```
/** An instance is a book and its title */
public class Book {
   private Object book; // the book, as an object
   /** Constructor: A book b.  */
   public Book(Object b) {
      book= b;
   }
   /** = this book */
   public Object getBook() {
     return book;
   }
}
```

Some operations of Vector v and String s

s.endsWith(s1) = “s ends with String s1”

v.size() = number of elements of v
v.get(i) = element i of vector v
v.add(ob); Add object ob to v
v.clear(); Remove all elements from v
v.contains(ob) = v contains Object ob
v.indexOf(ob) = index of first occurrence of ob in v (-1 if not in)
v.lastIndexOf(ob) = index of last occurrence of ob in v (-1 if not in)

Question 1 (15 points) Exceptions, abstract classes, and subclasses.

(a) On the back of the previous page, write a class `OutOfSpaceException`, whose objects can be thrown. You determine which class it should extend.

(b) Abstract classes and subclasses. Below (use the back of the previous page if necessary), write a subclass `Shelf` of class `Repository` (see page 1). *In addition to any components required from class* `Repository`, `Shelf` should contain the following fields and methods; make them private or public according to standard practice. You do not have to put a class invariant or specs on the methods because they are given here:

- `Vector<Book> contents` (a field): the books that are on this shelf. The **location** of a book on this shelf is its index in this vector. The shelf may not hold more books than the capacity given when this instance was created.
- A constructor that helps create an empty shelf (nothing on it) that can hold c books (a parameter).
- `add(Book b)`: A function to put book `b` on this shelf and return its location. Throw an `OutOfSpaceException` if there is not enough space on the shelf for `b`.
- `get(int i)`: a function that returns the book at location `i` of this shelf (null if not there)

Question 2 (15 points). Subclasses, etc.

(a) Below, write a subclass `WebBook` that extends class `Book` (see page 1). An instance is a `Book` that is also on the web, so that in addition to the book itself, the URL of the book is maintained. Provide the following methods and any fields that are necessary. There is no need to specify these methods since we give the specs here.

- A constructor that has two parameters: the book and its URL (as a String). Throw an `IllegalArgumentException` if the url does not end in ".pdf".
- a getter function for the url.

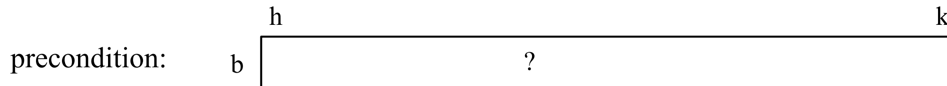
(b) Write the following method, which is *not* in class Book or WebBook but in some other class.

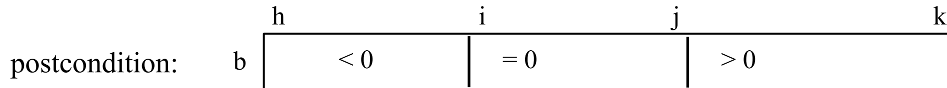
```
/** = the url for b (null if it doesn't have one) */
public static String url(Book b) {



}
```

Question 3 (13 points). Algorithms. Write the body of function DNF. Use a Dutch National Flag algorithm, which we can think of as swapping not colored balls but an array of ints, putting the negative ones first, then the 0's, and then the positive ints. You **must** write an invariant first, in the place given below, and the code you write **must** use the precondition, postcondition, and the invariant that you write. Few points will be given if your code has nothing to do with the invariant. To swap two elements, you may write something like "swap b[v] and b[w]".

precondition: 

postcondition: 

invariant:

/** Use a Dutch National Flag algorithm to arrange the elements of b[h..k] and produce a Point object Point (i, j). The precondition and postcondition are given above. */
public static Point DNF(**int**[] b, **int** h, **int** k) {

return ____;

}

```
/** An instance is an immutable point (x,y) */
public class Point {
   public final int x;
   public final int y;

   /** Constructor: an instance for (x, y) */
   public Point(int x, int y) {
      this.x= x;  this.y= y;
   }
}
```

Question 4 (13 points) Two-dimensional arrays and loops. Recall drawing bricks, or rectangles, in A7. Write method `placeBricks`, whose specification and header appears below. It creates a 2-dimensional array of elements of class `Brick` and adds them to the GUI, as shown to the right. Do not use recursion.

0	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	2	3	4	5	6
3	3	3	3	4	5	6
4	4	4	4	4	5	6
5	5	5	5	5	5	6
6	6	6	6	6	6	6

Here are important points, assuming the name of the array is `b`:

- `b` has `m` rows and `m` columns of `Brick`s; precondition: `0 < m`.
- `b[0][0]` is placed at the upper-left corner `(0,0)` of the GUI.
- Each `Brick` has side length `20`, and there is no space between them.
- The color of a square depends on `r` (`r` is the number in each Brick):
  if `r` is even, bricks `b[r][0..r]` and `b[0..r-1][r]` are `Color.red`; otherwise, `Color.green`.
- You will write nested for-loops, each of which processes a range of integers. The previous point should help you determine how to write them. You need not write loop invariants.
- Class `Brick`, which extends `GRect`, has this additional constructor:

```
/** Constructor: a new square Brick at (x,y) of the GUI with
    side length s, color color, and label lab.*/
public Brick(double x, double y, double s, Color color, int lab)
```

- Add a `Brick d` to the GUI using `add(d);` .

```
/** = an array of m x m squares, as specified above, all of which have been added to the GUI. */
public Brick[][] placeBricks(int m) {




}
```

Question 5 (14 points). Recursion.

(a) Consider class Rhino —the only fields necessary for this question are given to the right. Complete the bodies of the two functions declared below. The first function, which is static and is defined in class Rhino, finds the weight of the heaviest Rhino in an ancestry tree. It may help to use Math.max in this one, but you don't have to. The second appears in every Rhino object and find the heaviest Rhino in an ancestry tree.

These are two separate, independent questions. Do not solve the second one by calling the first or vice versa.

```
/** an instance maintains info
    about a Rhino */
public class Rhino {
  double weight; // rhino's
                 // weight, > 0
  Rhino mother;  // rhino's mother
                 // (null if unknown)
  Rhino father;  // rhino's father
                 // (null if unknown)
}
```

Note: The rhino at the top or root of the ancestry tree is part of the tree and therefore may be the heaviest.

```
/** = the weight of the heaviest Rhino in r's ancestry tree.
      (if r is null, use 0 as the weight) */
public static double heaviest(Rhino r) {

}
```

```
/** = the heaviest Rhino in this rhino's ancestry tree. */
public Rhino heaviest() {

}
```

Question 6 (14 points) Miscellaneous.

(a) Explain how to execute a while-loop "**`while`** `(` expression `)` statement"

(b) Consider the statement below. Draw variable `bb`. Then execute the assignment statement, drawing the objects that are created. The goal is to see whether you know how ragged arrays look. You may show what is in an array object in any reasonable way, but you must draw all objects.

```
int[][] bb= new int[][]{{3, 2}, {4}, {}};
```

(c) We have forgotten how to find the length of an array `b`, and we are in a hurry. We do remember that evaluation of `b[k]` throws an `ArrayIndexOutOfBoundsException` if `k` is not the index of an array element of `b`. So we (meaning you) write the function below, using a loop (with initialization) that successively evaluates `b[0]`, `b[1]`, `b[2]`, ... until the exception is thrown, at which time `k` will be the length! Write the body of the function. You will need a try-statement. You may not use `b.length`.

```
/** = number of elements in b */
public static int size(Object[] b) {

}
```

Question 7 (14 points) new-expressions, assignments, and debugging. Consider the two classes at the bottom of this page. When drawing an object, don't draw the partition for class `Object`.

(a) Consider writing a JUnit testing class to test the classes at the bottom of the page. Write a test procedure to test the constructor of class `M`, making sure that there is complete test coverage —each part of the constructor is tested in at least one test case. We made field `f` public to make this easier.

(b) Evaluate the new-expression `new M(5)`. As you do it, draw any objects that are created and draw the frames for any method calls that are executed. Don't erase the frames; just cross them out. Write the value of the expression here: ____________________.

(c) Below is a sequence of three statements. First, draw the variables declared in the statements. Second, execute the sequence, drawing any objects that are created. Do not draw frames for calls. Do not attempt to show us the state of affairs after each statement by drawing the variables again and again. Just execute the sequence of statements, changing the values of variables as execution requires.

```
B b= new B(3);
M c= new M(2);
M a= b;
```

```
public class B extends M {

   public B(int n) {
     super(2*n);
   }

   public int lead() {
     return f + 2;
   }
}
```

```
public class M {
   public int f;

   public M (int n) {
     f= (n is odd ? 2 : n);
   }

   public int lead() {
     return f / 2;
   }
}
```