CIS 120 Final Exam 19 December 2014

Name (printed): ____________________________

Pennkey (login id): ____________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: ______________________________________ Date: _________________________

1	/10
2	/18
3	/24
4	/12
5	/16
6(a)	/8
6(b)	/8
6(c)	/12
7	/12
Total	/120

- Do not begin the exam until you are asked to do so.
- You have 120 minutes to complete the exam.
- There are 120 total points.
- There are 12 pages in this exam, plus an Appendix.

1. **Java Concepts: True or False** (10 points)

All of the following questions pertain to Java.

a. T F If `s` and `t` are variables of type `String` such that `s == t`, then `s.equals(t)` is guaranteed to return **`true`**.

b. T F The call `m(new Object())` may throw a `NullPointerException` when `m` is declared with the type signature shown below:

```
public void m(Object o)
```

c. T F The call `m(new Object())` may throw an `IOException` when `m` is declared with the type signature shown below:

```
public void m(Object o)
```

d. T F The keyword `synchronized`, when applied to a method, ensures that at most one thread at a time is executing in method's body.

e. T F Whenever you override the `equals` method of a class, you should be sure to override the `hashCode` method compatibly.

f. T F The `Circle` class below demonstrates how *inheritance* can be used to avoid code duplication:

```
public class Circle {
  Point center;
  public Circle(int x, int y) {
    center = new Point(x,y);
  }
}
```

g. T F The variables `p` and `q` are *aliases* when the following program's execution reaches the line marked "HERE". (Assume that `ColoredPoint` is a subclass of `Point`.)

```
Point p = new Point(1, 2);
Point q = new ColoredPoint(1, 2, Red);
p = q;
// HERE
```

h. T F A recommended way to improve the performance of Java programs that use file I/O is to use a *buffered* stream, like this:

```
InputStream f = new new FileInputStream("filename.dat");
InputStream fin = new BufferedInputStream(f);
```

i. T F The **`this`** reference is always guaranteed to be non-**`null`**.

2. **Binary Search Trees** (18 points)

Recall the type definitions for binary trees from Homework 3:

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
```

Also recall that binary search trees (BST) are trees with an additional invariant (which hopefully you remember).

a. Draw *all* of the possible binary search trees that contain exactly the integers 1, 2, and 3.

b. Recall that the *height* of a binary tree is the maximum number of nodes from the root to any leaf, while the *size* of binary tree is the total number of nodes in the tree. Note that `(height Empty) = 0`. Below is the standard lookup function for binary search trees:

```
let rec lookup (x: 'a) (t: 'a tree) : bool =
  begin match t with
  | Empty -> false
  | Node(lt, y, rt) ->
     if x < y then lookup x lt
     else if x > y then lookup x rt
     else true
```

i. Does the `lookup` function take time proportional to the *height* or the *size* of the tree?

height size

ii. Is it possible to construct a tree `t` such that the size of `t` is equal to its height? In one sentence, explain why or why not.

iii. Is it possible to construct a tree `t` such that the size of `t` is 1000 but the height of `t` is 10? In one sentence, explain why or why not.

3. OCaml Programming: Abstraction, Lists, Higher-order Functions (24 points)

This problem refers to the OCaml code in Appendix A, which gives a signature `Encoding` that provides operations for representing compressed representations of lists.

The module `RLE` implements this signature. It compresses a list using *run-length encoding*: sequences of identical consecutive elements of the list are represented by a pair of the element and the length of the sequence. The encoding of a whole list is a list of such pairs. For example, the list consisting of four `"a"`s followed by three `"b"`s and one `"c"` has the (module-internal) representation computed by the `encode` function as shown below:

```
(encode ["a"; "a"; "a"; "a"; "b"; "b"; "b"; "c"]) = [("a", 4); ("b", 3); ("c", 1)]
```

For each of the following multiple choice questions, circle *all* correct answers; *there may be more than one.*

a. The type `'a encoding` is *abstract*. Circle all of the code snippets that illustrate a legal way in which a client of the `RLE` module may use its interface.

i.
```
;; open RLE
let l : int list = decode (encode [1;2;3])
```

ii.
```
;; open RLE
let l : int list = decode [(1,2)]
```

iii.
```
;; open RLE
let l : (int * int) list = encode [1]
```

iv.
```
;; open RLE
let l : int encoding = encode(decode(encode []))
```

b. Suppose that the `RLE` representation of a compressed list is:

```
[(v_1, cnt_1); (v_2, cnt_2); (v_3, cnt_3); ... ; (v_N, cnt_N)]
```

Circle the statements that are *invariants* that the `RLE` module enforces about the compressed representation:

i. `N` is greater than `0`, that is, the representation is a non-empty list.

ii. None of `cnt_1`, `cnt_2`, . . . , `cnt_N` is `0`.

iii. Adjacent values `v_i` and `v_(i+1)` are not equal, i.e. `v_1 <> v_2` and `v_2 <> v_3` and `v_3 <> v_4`, etc.

iv. Adjacent counts `cnt_i` and `cnt_(i+1)` are not equal, i.e. `cnt_1 <> cnt_2` and `cnt_2 <> cnt_3` and `cnt_3 <> cnt_4`, etc.

c. Circle the programs that implement the same function as `RLE` module’s `decode`? The definitions of the higher-order functions `transform` and `fold` are given in Appendix A.

i.
```
let decode (l:(’a * int) list) : ’a list =
  fold (fun (x,cnt) r -> (replicate x cnt)@r) [] l
```

ii.
```
let decode (l:(’a * int) list) : ’a list =
  transform (fun (x,cnt) -> replicate x cnt)) l
```

iii.
```
let decode (l:(’a * int) list) : ’a list =
  fold (fun xs r -> xs @ r) []
      (transform (fun (x,cnt) -> replicate x cnt) l)
```

iv.
```
let decode (l:(’a * int) list) : ’a list =
  fold (fun (x,cnt) r -> transform
                    (fun l -> replicate x cnt) r)
      [] l
```

d. Suppose that you wanted to add an operation that transforms an encoded list analogously to the way that `transform` works on ordinary lists. Which function signature should you add to the `Encoding` signature?

i. `val transform : (’a encoding -> ’b encoding) -> ’a -> ’b`

ii. `val transform : (’a -> ’b) -> ’a encoding -> ’b encoding`

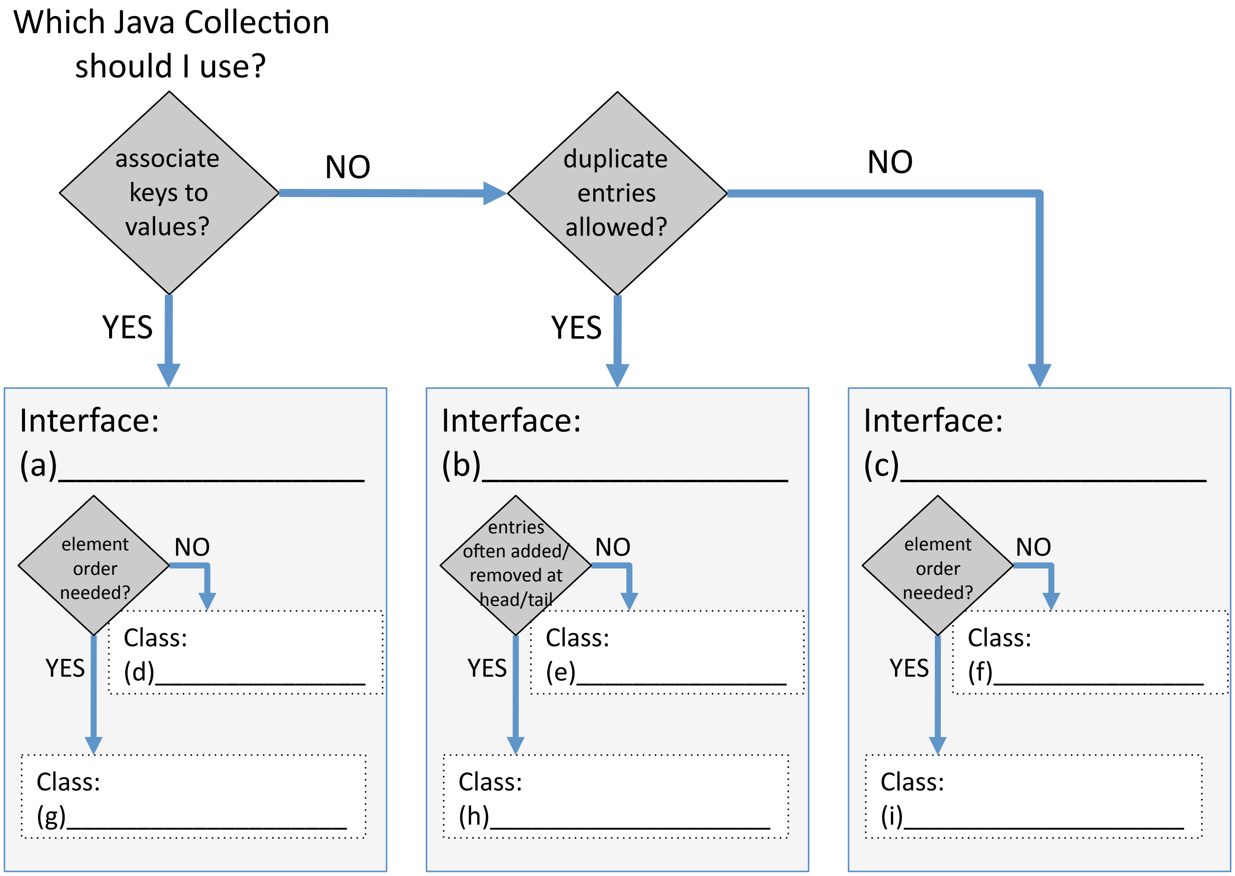
iii. `val transform : ’a encoding -> ’b encoding`

iv. `val transform : (’a -> ’b) -> ’b encoding -> ’a encoding`

4. Java Collections (12 points)

(a) The flow chart below is designed to help programmers determine which of Java's collections to use. There are nine blanks labeled (a) through (i). Fill in the blanks using the names of the appropriate interfaces or classes drawn from those given below. *Hint: only nine of the names below will be used.*

```
Set         Map         List
ArraySet    LinkedMap   ArrayList
HashSet     HashMap     LinkedList
TreeSet     TreeMap     TreeList
```



(b) (multiple choice) Suppose that you were going to use Java Collections to implement a program that simulates a simplified version of the Java Abstract Stack Machine that does not need to push workspaces on to the stack. Assuming that you have classes `Identifier` and `Value`, which type below would be the best fit for representing the *stack* part of the Abstract Stack Machine?

(a) `Set<Identifier>`
(b) `LinkedList<Identifier>`
(c) `LinkedList<Map<Identifier, Value>>`
(d) `ArrayList<Value>`
(e) `Map<Identifier, Set<Value>>`
(f) `Map<Value,Identifier>`

5. Java Types (16 points)

Consider the classes and interfaces `Iterable`, `Collection`, `List`, `Iterator`, `ArrayList` and `LinkedList` from the Java Collections Framework. (More information about these interfaces and classes is shown in Appendix B.)

Write down a type for each of the following Java variable definitions. Due to subtyping, there may be more than one type that would work—you should put the **most specific type** permissible in that case. (For example, if `C implements I` then you should put `C` instead of `I`). Write **static error** if the compiler would flag an error anywhere in the code snippet for any reason.

Hint: read through the Appendix carefully. Even though you may have used these methods before, you may not have given much thought to their types.

The first two have been done for you as a sample and are used in the remaining definitions.

________`ArrayList<String>`___________ `x = new ArrayList<String>();`

_____`ArrayList<List<String>>`________ `y = new ArrayList<List<String>>();`

a. ______________________________ `a = y.get(0).get(0);`

b. ______________________________ `b = x.add("CIS120");`

c. ______________________________ `c = x.contains(y);`

d. ______________________________ `d = y.add(x.get(0));`

e. ______________________________ `e = y.iterator()`

f. ______________________________ `f = ((List<String>)null).get(0);`

g. ______________________________ `g = new Collection<String>();`

Note: This last code snippet is larger than a single expression: it assigns to h twice.

h. ______________________________ `h = x;`
`h = new LinkedList<String>();`

6. Java Data Structures Programming (28 points total)

Appendix C shows (part of) the implementation of a new Java `Collection` class called `Deque<E>`, which implements the `List<E>` interface. The code provided is a Java version of the doubly-linked queues (a.k.a. deques) mutable data structure that you implemented for HW05 in OCaml. There is a type `DQNode`, which stores an element, along with references to the next and previous elements of the deque. A deque object has references to the head and tail of the deque.

Your task in this problem is to implement some missing functionality of the `Deque<E>` class.

a. (8 points) First, implement the `contains` method, the JavaDoc for which is given below:

> **`boolean contains(Object o)`**
> Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element `e` such that either `o` and `e` are both `null` or `o` is non-`null` and `o.equals(e)`.

```
@Override
public boolean contains(Object o) {





}
```

b. (8 points) Because it implements the `List<E>` interface, the `Deque<E>` class must be able to provide an `Iterator<E>` object via the `interator()` method. (See Appendix B for the relevant interfaces.) Appendix D contains the code for a class called `DequeIterator`, which is an inner class of `Deque<E>`. We have given you the `hasNext()` and the `next()` methods. Your next task is to implement some a test case for the `remove()` method of the iterator, the JavaDoc for which is given below:

> **`void remove()`**
> Removes from the underlying collection the last element returned by this iterator. This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.
> **Throws:** `IllegalStateException` - if the next method has not yet been called, or the remove method has already been called after the last call to the next method

The test case below checks one desired behavior:

```
@Test public void removeOnlyElement() {
  List<Integer> l = new Deque<Integer>();
  l.add(3);
  Iterator<Integer> i = l.iterator();
  i.next();
  i.remove();
  assertTrue(l.isEmpty());
}
```

Complete the following test which should check that the `remove()` method properly raises an `IllegalStateException` when called before `next`.

Hint: You may want to use the `fail()` *method, which causes a test to fail.*

```
@Test public void removeFailsWhenCalledBeforeNext() {
  List<Integer> l = new Deque<Integer>();
  Iterator<Integer> i = l.iterator();



}
```

c. (12 points) Lastly, complete the `remove()` method to match the description given above. Recall the invariants for the `Deque` datatype (suitably translated to Java):

- The deque is empty, and the head and tail are both `null`,
- or, the deque is non-empty, and:
  - `head != null` and `tail != null` and
    - · `tail` is reachable from `head` by following `next` pointers
    - · `tail.next == null` (there is no element after the tail)
    - · `head` is reachable from `tail` by following `prev` pointers
    - · `head.prev == null` (there is no element before the head)
  - for every `DQNode n` in the deque: if `n.next != null` then `n.next.prev == n`
  - for every `DQNode n` in the deque: if `n.prev != null` then `n.prev.next == n`

Your implementation of `remove` should maintain these invariants.

Hint: the field `lastReturned` *is set by the* `next()` *method. If it is non-*`null`*, then it refers to the node to be removed. We have started implementing this method for you.*

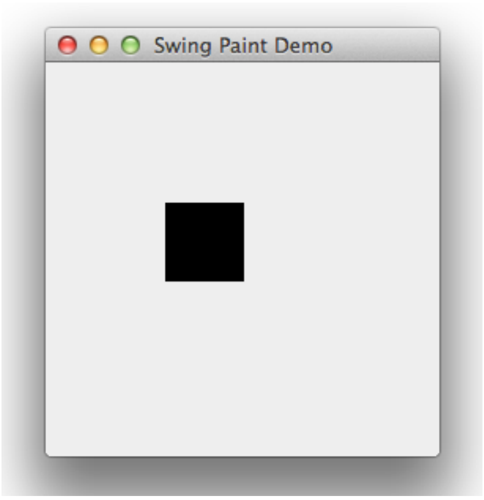
```
@Override
public void remove() {
  // check that it is valid to call remove
  if (lastReturned == null) {
    throw new IllegalStateException();
  }
  // write your code below



  // ensure that next must be called before remove can be called again
  lastReturned = null;
}
```

7. Java Swing Programming and Debugging (12 points)

The code in Appendix E implements a simple Java GUI program in which a 50x50 black box follows the mouse cursor around the window. It looks like this (the mouse cursor is not shown):



a. Write down the line number(s) on which the keyword **`new`** is used to create an instance of an anonymous inner class, or “none” if there are none (there may be zero or more than one).

b. During the development of this program, you encountered several buggy behaviors, caused by bugs in your use of the Swing library.

For each of the following coding errors, indicate what buggy behavior the program might exhibit. The buggy behaviors are numbered (1) through (3), and they are described on the next page. Each coding error leads to exactly one buggy behavior

Behavior Coding Error

______________ Omitted the call to `f.pack()` on line 16.

______________ Accidentally used `addMouseListener` instead of `addMouseMotionListener` on line 27.

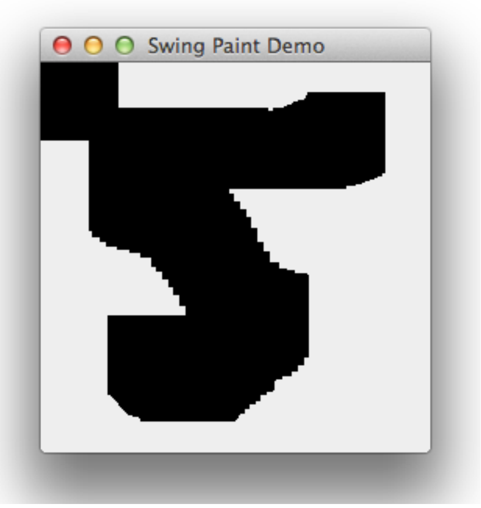
______________ Omitted the call to `repaint()` on line 31.

______________ Forgot to override the method `getPreferredSize` (lines 36–39 were omitted)

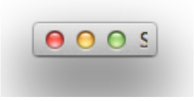
______________ Omitted the call to **`super`**`.paintComponent(g)` on line 43.

Buggy behaviors:

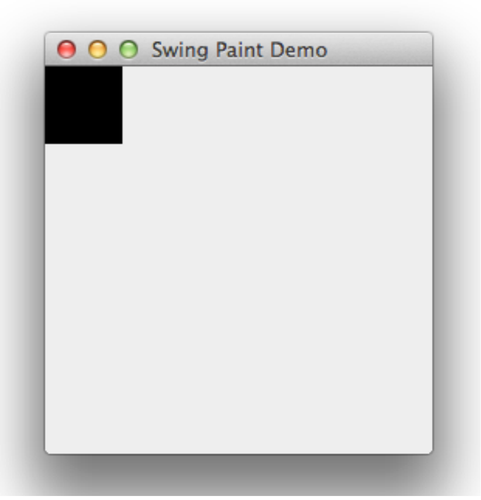
Behavior (1): The mouse movement causes the screen to look like this, which is cool, but not what you wanted. The square should follow the mouse, not “draw” on the screen.



Behavior (2): The window was tiny like this, which isn’t even cool:



Behavior (3): The window looked like this, and no matter how you moved the mouse, the square didn’t do anything:



A OCaml Compressed List

```
(* A signature for list encodings *)
module type Encoding = sig
  type ’a encoding
  val encode : ’a list -> ’a encoding
  val decode : ’a encoding -> ’a list
end

(* run-length encoding for compressing lists *)
module RLE : Encoding = struct
  type ’a encoding = (’a * int) list

  (* run-length encodes a list *)
  let rec encode (l:’a list) : (’a * int) list =
    begin match l with
      | [] -> []
      | x::xs ->
        begin match encode xs with
          | [] -> [(x, 1)]
          | ((y,cnt)::ys) ->
           if x = y then (y,cnt+1)::ys else
             (x,1)::(y,cnt)::ys
        end
    end

  (* produces a list containing cnt copies of the element x *)
  let rec replicate (x:’a) (cnt:int) : ’a list =
    if cnt = 0 then [] else x::(replicate x (cnt-1))

  (* decodes a run-length encoded list *)
  let rec decode (l:(’a * int) list) : ’a list =
    begin match l with
      | [] -> []
      | (x,cnt)::xs -> (replicate x cnt)@(decode xs)
    end
end
```

```
(* Higher-order functions *)

let rec transform (f:’a -> ’b) (l:’a list) : ’b list =
  begin match l with
    | [] -> []
    | h::tl -> (f h) :: (transform f tl)
  end

let rec fold (combine:’a -> ’b -> ’b) (base:’b) (l:’a list) : ’b =
  begin match l with
    | [] -> base
    | h::tl -> combine h (fold combine base tl)
  end
```

B Excerpt from the Collections Framework

```
interface Iterator<E> {
  public boolean hasNext();
  // Returns true if the iteration has more elements.
  public E next();
  // Returns the next element in the iteration .
  public void remove();
  // Removes from the underlying collection the last element
  // returned by this iterator
}

interface Iterable<E> {
  public Iterator<E> iterator();
  // Returns an iterator over a set of elements of type E.
}

interface Collection<E> extends Iterable<E> {
  public boolean add(E o);
  // Ensures that this collection contains the specified element
  // Returns true if this collection changed as a result of the call .
  // (Returns false if this collection does not permit duplicates
  // and already contains the specified element.)
  public boolean contains(Object o);
  // Returns true if this collection contains the specified element.
}

interface List<E> extends Collection<E> {
  public E get(int i);
  // Returns the element at the specified position in this list
}

class ArrayList<E> implements List<E> {
   public ArrayList();
   // Constructs an empty list with an initial capacity of ten

   // ... methods specified by interface
}

class LinkedList<E> implements List<E> {
   public LinkedList();
   // Constructs an empty list

   // ... methods specified by interface
}
```

C Deque<E> class (partial implementation)

```
import java.util.Iterator;
import java.util.List;
import java.util.NoSuchElementException;

public class Deque<E> implements List<E> {

  // The type of doubly-linked nodes in the queue
  private final class DQNode {
    public final E v;
    public DQNode next;
    public DQNode prev;
    public DQNode(E v, DQNode next, DQNode prev) {
      this.v = v;
      this.next = next;
      this.prev = prev;
    }
  }

  private DQNode head;
  private DQNode tail;

  @Override
  public boolean isEmpty() {
    return (head == null);
  }

  // Insert an element at the tail of the deque
  @Override
  public boolean add(E e) {
    DQNode newNode = new DQNode(e, null, tail);
    if (tail == null) {
      head = newNode;
      tail = newNode;
    } else {
      tail.next = newNode;
      tail = newNode;
    }
    return false;
  }

  @Override
  public Iterator<E> iterator() {
    return new DequeIterator();
  }

  @Override
  public boolean contains() {
    // see exam problem 6 (a)
  }
  // ... code for DequeIterator ...
}
```

D DequeIterator (inner class of Deque<E>)

```
// An inner class of Deque<E>, it appears at the location
// marked ... code for DequeIterator ... on the previous page
private final class DequeIterator implements Iterator<E> {
  private DQNode curr;
  private DQNode lastReturned;

  DequeIterator() {
    this.curr = head;
  }

  @Override
  public boolean hasNext() {
    return (curr != null);
  }

  @Override
  public E next() {
    if (curr == null) {
      throw new NoSuchElementException();
    }
    lastReturned = curr;
    curr = curr.next;
    return lastReturned.v;
  }

  @Override
  public void remove() {
    // see exam problem 6 (c)
  }
}
```

E An Example Java GUI Program

```java
// library imports omitted to save space (this code compiles)

public class GUI {
  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
  }

  static void createAndShowGUI() {
    JFrame f = new JFrame("Swing Paint Demo");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.add(new MyPanel());
    f.pack();
    f.setVisible(true);
  }
}

@SuppressWarnings("serial")
class MyPanel extends JPanel {
  private int x;
  private int y;

  public MyPanel() {
    addMouseMotionListener(new MouseAdapter() {
      public void mouseMoved(MouseEvent e) {
        x = e.getX();
        y = e.getY();
        repaint();
      }
    });
  }

  @Override
  public Dimension getPreferredSize() {
    return new Dimension(250, 250);
  }

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.fillRect(x, y, 50, 50);
  }
}
```