Name (NEATLY): _____								
University ID#: _____								
TA's name:	Richard	Ahmed	Joan	Jyna	Jesse	Manaswi	Steve	
Discussion time:	8	9	10	11	12	1	2	4

CMSC 132 Exam #1 practice questions Spring 2016

Do not open this exam until you are told. Read these instructions:

1. This is a closed book exam. **No calculators, notes, or other aids are allowed.** If you have a question during the exam, please raise your hand. Each question's point value is next to its number.
2. **You must turn in your exam immediately when time is called at the end.**
3. In order to be eligible for the most partial credit, show all of your work for each problem, **write legibly**, and **clearly indicate** your answers. Credit **cannot** be given for illegible answers.
4. You will **lose credit** if ***any*** of your identifying information above is incorrect or missing.
5. You will **lose credit** if your name is not written at the top of each odd–numbered page where indicated.
6. In any code you have to write: (a) **Do not** use exceptions for normal processing, (b) Use a **minimal** number of `return` statements, (c) You may **not** use any Java library collection classes anywhere, unless a problem specifically allows it, (d) `break` and `continue` may **not** be used in any loops, and (e) Inefficient code may earn reduced credit.
7. Parts of some pages are for scratch work. If you need extra scratch paper after you have filled these areas up, please raise your hand. Scratch paper must be turned in with your exam, with your name and ID number written on it. Scratch paper **will not** be graded.
8. To avoid distracting others, **no one** may leave until the exam is over.
9. The Campus Senate has adopted a policy asking students to include the following handwritten statement on each examination and assignment in every course: "*I pledge on my honor that I have not given or received any unauthorized assistance on this examination.*" Therefore, **just before turning in your exam**, you are requested to write this pledge **in full** *and* **sign it** below:

_____

_____

Good luck!

1. Consider the classes below and on the right, which are used by the code fragments below them and on the next page. Give the **output** of each code fragment. However, if any code fragments are syntactically incorrect, **or** would cause any type of error when executed, don't give their output, just write ***invalid*** instead (justification not necessary). (There are no errors in the classes themselves.) Each code fragment would be located **in the `main()` method in the class `Shapes`**, where the comment appears. (Because the fragments are in that method, they can use the variables declared there.) Note that `Circle` is the only class that has an explicit constructor.

   The parts are all independent, so each part only uses the initial declarations in the `main()` method, and does not depend on any parts above it.

Shape.java

```
interface Shape {
  public void draw(int color);
}
```

Rectangle.java

```
class Rectangle implements Shape {

  public void draw(int color) {
    System.out.println("#1");
  }

  public void draw(double size) {
    System.out.println("#2");
  }

  public void rotate() {
    System.out.println("#3");
    draw(1.25);
  }

}
```

Square.java

```
class Square extends Rectangle {

  public void draw(double size) {
    System.out.println("#4");
  }

  public void draw(String animation) {
    System.out.println("#5");
  }

  public void resize() {
    System.out.println("#6");
  }

}
```

Circle.java

```
class Circle implements Shape {

  Circle() {
    System.out.println("#7");
  }

  public void draw(int color) {
    System.out.println("#8");
  }

  public void draw(String animation) {
    System.out.println("#9");
  }

  public void rotate(double diam) {
    System.out.println("#10");
  }

}
```

Shapes.java

```
public class Shapes {

  public static void
         main(String[] args) {
    Shape aShape;
    Rectangle aRect;
    Square aSquare;
    Circle aCirc;
    Rectangle[] rectangles;
    Square[] squares;

    // the code fragments would
    // appear exactly right here
  }

}
```

a.
```
aShape= new Shape();
System.out.println(aShape
                   instanceof Shape);
```

b.
```
aShape= new Rectangle();
System.out.println(aShape
                   instanceof Shape);
```

c.
```
aShape= new Square();
aShape.resize();
```

d.
```
aSquare= new Square();
aSquare.draw(3);
```

e.
```
aSquare= new Circle();
aSquare.draw("fast");
```

f.
```
aRect= new Rectangle();
((Square) aRect).resize();
```

g.
```
aSquare= new Square();
((Rectangle) aSquare).draw(1.23);
```

h.
```
aRect= new Square();
aRect.draw(2.5);
```

i.
```
squares= new Square[100];
rectangles= squares;
rectangles[0]= new Rectangle();
rectangles[0].draw(2.5);
```

j.
```
aSquare= new Rectangle();
aSquare.rotate();
```

k.
```
aRect= new Square();
aRect.rotate();
```

2. Consider the classes below. The code fragments following them, and on the next page, use these classes. Give the output of each code fragment. However, if any code fragments are incorrect or would cause any type of error when running, don't give their output, just write "invalid" instead (justification not necessary). The classes themselves are valid Java. **Each code fragment would be located in the `main()` method in the class `Horse`, where the comment appears.** (Because the fragments are in that method, they use things declared there.)

   The parts are all independent, so each part only uses the initial declarations in the `main()` method, and does not depend on any parts above it.

Animal.java

```
interface Animal {
  public void eat();
}
```

Mammal.java

```
class Mammal implements Animal {

  private int weight;

  Mammal() {
    System.out.println("Mammal()");
    weight= 0;
  }

  Mammal(Mammal other) {
    weight= other.weight;
  }

  public void setWeight(int weight) {
    this.weight= weight;
  }

  public void eat() {
    System.out.println("Mammal eat()");
  }

  public void eat(int i) {
    System.out.println("Mammal eat(" +
                       i + ")");
  }

  public void feed() {
    eat();
  }

}
```

Horse.java

```
class Horse extends Mammal {

  private String name;

  Horse(String name, int weight) {
    System.out.println("Horse()");
    this.name= name;
    setWeight(weight);
  }

  Horse(Horse other) {
    super(other);
    name= other.name;
  }

  void setName(String name) {
    this.name= name;
  }

  String getName() {
    return name;
  }

  public void eat() {
    System.out.println("Horse eat()");
  }

  public void eat(String s) {
    System.out.println("Horse eat(" +
                       s + ")");
  }

  public static void main(String[]
                          args {
    Mammal m= new Horse("Silver", 800);
    Animal a= new Horse("Trigger",
                        1500);
    Horse h= new Horse("Mr. Ed", 1250);

    // each fragment would be here
  }

}
```

a. `Horse e2= new Horse("Misty", 850);`

b. `Mammal m2= new Horse("Star", 1750);`

c. `h.eat(10);`

d. `m.eat("grass");`

e. `((Animal) h).eat();`

f. `m.feed();`

g. `System.out.print(h instanceof Mammal)`

h. `System.out.print(a instanceof Mammal)`

i.
```
Horse h2= new Horse(h);
h.setName("My Little Pony");
System.out.print(h2.getName());
```

j.
```
Horse h2= h;
h.setName("My Little Pony");
System.out.print(h2.getName());
```

3. Consider the Java classes (and interface) below and on the right, which are used by the code fragments below them and on the next page. Note that one class is abstract. Give the output of each code fragment. However, if any fragments are syntactically incorrect, **or** would have any type of error when executed, just write ***invalid***. (There are no errors in the classes and interface themselves.)

   The parts are all independent, so parts do **not** depend on the results of any parts above them.

Marsupial.java

```
package question1;

class Marsupial {

  private String name;

  Marsupial(String name) {
    this.name= name;
  }

  void eats() {
    System.out.println("Eats #1.");
  }

  String getName() {
    return name;
  }

}
```

Herbivore.java

```
package question1;

abstract class Herbivore
         extends Marsupial {

  Herbivore(String name) {
    super(name);
  }

  void eats() {
    System.out.println("Eats #2.");
  }

  abstract void chews(boolean b);

}
```

Australian.java

```
package question1;

interface Australian {
  void greets(Koala k);
}
```

Koala.java

```
package question1;

class Koala extends Herbivore
            implements Australian {

  Koala(String name) {
    super(name);
  }

  public void greets(Koala k) {
    System.out.println("G'day mate!");
    System.out.println(getName() + " "
                       + k.getName());
  }

  void chews(boolean b) {
    System.out.println("Yum, yum.");
  }

  void chews(int i) {
    System.out.println("Delicious!");
  }

  void likes(Koala k) {
    greets(k);
    k.greets(this);
  }

}
```

a.
```
Koala k= new Koala("Kourtney");
// the next statement prints a boolean
System.out.println(k instanceof
                   Marsupial);
```

b.
```
Marsupial m= new Marsupial("Marsha");
Marsupial m2= new Koala("Katelyn");
// the next statement prints a boolean
System.out.println(m.getClass() ==
                   m2.getClass());
```

c.
```
Herbivore h= new Marsupial("Mary");
System.out.println(h.getName());
```

d.
```
Marsupial m= new Marsupial("Molly");
m.eats();
```

e.
```
Marsupial m= new Koala("Kate");
m.eats();
```

f.
```
Marsupial m= new Herbivore();
m.eats();
```

g.
```
Herbivore h= new Koala("Kora");
((Marsupial) h).eats();
```

h.
```
Koala k= new Koala("Kourtney");
Koala k2= new Koala("Katie");
k.likes(k2);
```

i.
```
Australian a= new Australian();
a.greets(new Koala("Kim"));
```

j.
```
Australian a= new Koala("Khloe");
a.chews(true);
```

k.
```
Koala k= new Koala("Krystal");
k.chews(5);
```

l.
```
Herbivore h= new Koala("Kia");
h.chews(true);
```

m.
```
Herbivore h= new Koala("Karli");
h.chews(6);
```

4. Multiple choice– circle the numbers of all correct answers. There may be **one or more** correct answers for each question. **Note**: incorrect answers will lose credit, so it is not to your advantage to randomly guess. **Clearly indicate** what numbers you are circling (circle boldly).

   a. Which of the following are correct regarding an `equals()` method in Java?

      (1) An `equals()` method written in a class named `C` must have a parameter that is declared to be of type `C`, in order to override the `equals()` method in the `Object` class.

      (2) An `equals()` method must have a parameter that is declared to be of type `Object`, in order to override the `equals()` method in the `Object` class.

      (3) An `equals()` method written in a class named `C` must cast its parameter to `Object`, even if its parameter is declared to be of type `C`, otherwise it will not be able to properly compare the fields of current object to those of the parameter object.

      (4) A well–designed `equals()` method should check for and handle the case where its current object may be null.

      (5) An `equals()` method in a subclass cannot refer directly to private fields in a superclass in order to compare them, so it should invoke the superclass `equals()` method to do this.

   b. Which of the following are reasons why an initialization block would be useful? (Everywhere that it is used in this question the term "initialization block" refers to a **non–static** initialization block.)

      (1) An initialization block would avoid having to repeat code in multiple constructors.

      (2) An initialization block would be useful because it is always executed after the constructor executes.

      (3) An initialization block would be useful because a class can have multiple constructors, but only one initialization block.

      (4) An initialization block would be useful because you could put code in it that should only be executed once in a program, no matter how many objects of the class are created.

      (5) An initialization block would be useful if you want to confuse someone who was trying to cheat by looking at your program code, because they would never be able to understand it.

   c. Which of the following statements about generics in Java are accurate?

      (1) One reason that generics were added to the language is that before that, there was no way in Java to write code that operated upon more than one type of data.

      (2) One reason that generics were added to the language is that with generics some mistakes would be able to be detected as syntax errors, rather than runtime exceptions.

      (3) A class' generic type parameter cannot be used inside the class.

      (4) A class' generic type parameter can be used inside the class only for declaring arrays.

      (5) One reason that generics were added to the language is that Java did not have any collection classes (like `ArrayList`) before that, because it was impossible for collection classes to be implemented without using generics.

   d. What happens if a class does not implement all of the abstract methods of its superclass?

      (1) A runtime error (exception) will occur when unimplemented methods of the class are called.

      (2) The abstract class becomes an interface instead.

      (3) A class doesn't have to implement all of the methods in an interface, but it's a syntax error for a class to not implement all of the methods in its superclass if the superclass is abstract.

      (4) A class that doesn't implement all of the abstract methods of its superclass would have to be an abstract class also.

      (5) The programmer receives a small electric shock every time the program is compiled, gently leading them over the long term to realize the error of their ways.

e. A list could be stored in an array, or in a linked list (made up of nodes joined together by references). How would these approaches compare?

(1) A linked list would be more efficient than an array at inserting elements.

(2) A linked list would be more efficient than an array at indexing. (What indexing is was discussed in class.)

(3) A linked list storing n elements would use less memory than an array storing n elements.

(4) A linked list would have a maximum capacity and would have to be resized if it filled up, while an array would not have a maximum capacity other than the amount of memory available.

(5) An array would have a maximum capacity and would have to be resized if it filled up, while a linked list would not have a maximum capacity other than the amount of memory available.

f. Suppose a class has a static field (also called a class variable) named `sharedField`, and a nonstatic field (or instance variable) named `nonSharedField`. It also has a static method named `staticMethod()`, and a nonstatic method (or instance method) named `nonStaticMethod()`. Which of these are true?

(1) `staticMethod()` can access `sharedField` and also `nonSharedField`.

(2) `nonStaticMethod()` can access `sharedField` and also `nonSharedField`.

(3) `staticMethod()` can access `sharedField` but not `nonSharedField`.

(4) `nonStaticMethod()` can access `nonSharedField` but not `sharedField`.

(5) `staticMethod()` can only access `nonSharedField` if it is a private field.

g. Which of these are true about the foreach loop (also called an enhanced for loop) in Java:

(1) It can be used with any object of any class that implements the `Iterable` interface.

(2) It can be used with any object of any class that implements the `Iterator` interface.

(3) Any loop that is written by explicitly calling iterator methods can always be rewritten using a foreach loop instead, so that it has exactly the same effects.

(4) It can be used with any of the Java library collection classes (i.e., the library data structures).

(5) It kind of looks like a regular for loop if you squint hard enough when you look at it.

h. How do binary files compare to text files?

(1) Binary files are intended to be human–readable, while text files are not.

(2) Text files are intended to be human–readable, while binary files are not.

(3) A program can read text files faster than binary files.

(4) Data is stored in a text file in exactly the same way that it's stored internally in memory.

(5) If a list of large numbers have to be stored in either a text file or a binary file, using a binary file would save space (a binary file would be smaller than a text file containing the same numbers).

i. Why were generics added to Java?

(1) Because now that Java has generics, an `ArrayList` of a subclass type (like an `ArrayList<Car>`) is considered to be a subclass of an `ArrayList` of a superclass type (like an `ArrayList<Vehicle>`), so if a method has an `ArrayList<Object>` as a parameter, any `ArrayList` storing any type of values can be passed into it.

(2) Because before generics were added, Java did not have **any** mechanism that would allow writing code that could operate upon different kinds of data.

(3) Because before generics were added, Java did not have **any** library collection classes (data structures).

(4) Because generics allow the compiler to give syntax errors for certain mistakes, rather than runtime exceptions occurring.

(5) Because the designers of Java were embarrassed that C++ already had generics for a long time already.

j. In class we discussed the formula that the JVM uses to refer to elements of arrays:

$$\begin{array}{c}\text{the memory address}\\ \text{of an array element}\end{array} = \begin{array}{c}\text{the memory address of}\\ \text{the beginning of the array}\end{array} + \left(\begin{array}{c}\text{the subscript of the}\\ \text{desired element}\end{array} * \begin{array}{c}\text{the size of}\\ \text{each element}\end{array}\right)$$

Which of these are fact(s) about arrays that make this formula work? (**Note**: correct answers are **not** just true but irrelevant facts about arrays, but facts about arrays that cause this formula to work.)

(1) Arrays are stored in the runtime stack in Java.

(2) Arrays in Java must be created using `new`.

(3) All of the elements of an array are the same size in memory.

(4) An array's size cannot change after it has been created.

(5) Elements of arrays are stored in contiguous memory locations, in order by increasing subscript.

5. Consider the partial `MyList` class on the right. Obviously you don't know anything about the class other than what is shown, so you don't know whether a `MyList` object's elements are stored in an array, a linked list, or some other horrible type of complex data structure we never even talked about. Assume there is a default compiler–generated constructor. Note: this problem doesn't rely on knowing anything about what a `MyList` does, other than that the method `add()` adds an object to a `MyList`.

```
public class MyList<T>
             implements Iterable<T> {

  void add(T obj) {
    // details omitted
  }

  ⋮

}
```

Assume that the `MyList` class has a `size` field (an `int`) that keeps track of how many elements are stored in a `MyList`, and a public method `getSize()` that returns its value, which are just not shown above. Write a complete comparator class named `MyListSizeComparator` that will allow two `MyList`s to be compared based on their sizes. You can't assume the presence of anything in the `MyList` class other than what is shown or implied by what's above. Just write the `MyListSizeComparator` class, assuming any necessary libraries are already imported, but you don't have to write an `equals()` method in the comparator class.

6. Now consider the partial `MyArrayList` class (storing integers) on the right. It also stores a list of numbers using an array. The field `count` keeps track of how many values are currently being stored in the array of a `MyArrayList` object; there may be zero or more at any time but there will be no more than 100. You may assume the class has a constructor and methods to add elements to lists, which are not shown.

```
public class MyArrayList {

  final int SIZE= 100;

  private int arr[]= new int[SIZE];
  private int count= 0;

}
```

   a. Write the method `insertAtPosition(int position, int value)`, which should add the value `value` at position `position` of its current object, shifting the values at that position and beyond to make room for the new element.

   If `count` is 100 the method should just have no effect, and the same applies if `position` is greater than `count`.

   Your method **may not call any other methods** of the `MyArrayList` class, since you know there must be some but you're not told exactly what they are or exactly how they work. You may not write any helper methods, and do not assume that the `List` class has any fields other than those shown.

   The method **may not** create a new array. The method should work correctly, without errors, regardless of how many elements the list has, which may be zero one, or more than one. Make sure your method works right for all possible values of `position`!

   b. Write an `equals()` method for the `MyArrayList` class.

7. Consider the partial `MyArrayList` class on the right. It stores a list of `Integer`s, using an array. The field `count` keeps track of how many elements are currently being stored in the array of a `MyArrayList` object; there may be zero or more at any time, but the `add()` method shown ensures that there will be no more than 100. Note that the class has no constructor; the fields are just initialized as shown.

   On the next page, implement an iterator for `MyArrayList`, **using an inner class**. All the code you write on the next page will be inserted in the `MyArrayList` class where the comment is. Also, **give any additions or changes to the class that would be needed to allow the class to have an iterator and to allow the iterator to be written**. (Assume that any necessary imports have already been included, so you do not need to mention them.)

```
public class MyArrayList {

  final int SIZE= 100;

  private int arr[]= new int[SIZE];
  private int count= 0;

  boolean add(int newElt) {
    if (count == SIZE)
      return false;  // no room
    else {
      arr[count++]= newElt;
      return true;  // was able to add
    }
  }

  // your code magically appears here

}
```

You should implement the `remove()` method for your iterator. `remove()` must remove the element that was returned by the most recent call to `next()`.

Be sure to trace your `remove()` method carefully, ensuring that it will correctly remove just one element if it's called just once during an iteration over a `MyArrayList` object, and also that it will correctly remove all elements if it's called after every call to `next()` during an iteration.

Your iterator methods **may not** create any new arrays. An iterator should work correctly, without errors, regardless of how many elements a `MyArrayList` has, which may be zero, one, or more than one (up to 100). It should be possible for code to create **multiple** iterators that are iterating over the same `MyArrayList` object simultaneously, and each iterator should work independently and correctly.

Comments are optional, but if you have time to come back and write short ones at the end of the exam it might help you get more partial credit in case of mistakes. Write your method ***neatly and legibly*** on the next page, with good style and formatting.

8. Consider the partial singly–linked list definition below, which stores `Integer`s.

```
public class LinkedList {

  private static class Node {

    private Integer data;
    private Node next;

    Node(Integer data) {
      this.data= data;
      next= null;
    }

  }

  private Node head= null;

  public int length() {
    Node travel= head;
    int count= 0;

    while (travel != null) {
      count++;
      travel= travel.next;
    }

    return count;
  }

  public void createList() {
    Node temp= null;

    // each fragment would be here
  }

}
```

Suppose each code fragment below appears in the `createList()` method of the `LinkedList` class, where the comment appears. Give the output of each code fragment, which in each case will be the result of calling the method `length()` above on the current object list, at the end of the fragment. Explanation or justification is not necessary. However, if executing any part would result in an error, **briefly** explain the problem instead.

The parts are all independent– each part assumes a newly–declared and initialized current object list, and does not depend on the results of any parts above it.

a.
```
for (int i= 1; i <= 10; i++)
  head= new Node(i);
System.out.println(length());
```

b.
```
for (int i= 1; i <= 10; i++)
  head.next= new Node(i);
System.out.println(length());
```

c.
```
for (int i= 1; i <= 10; i++) {
  temp= new Node(i);
  temp.next= head;
  head= temp;
}
System.out.println(length());
```

d.
```
for (int i= 1; i <= 10; i++) {
  temp= new Node(i);
  temp.next= head;
  head= temp;
}
head= head.next.next.next;
System.out.println(length());
```

e.
```
for (int i= 1; i <= 10; i++) {
  if (head == null)
    head= new Node(i);
  else head.next= new Node(i);
}
System.out.println(length());
```

f.
```
for (int i= 1; i <= 10; i++) {
  if (head == null) {
    head= new Node(i);
    temp= head;
  } else {
    temp.next= new Node(i);
    temp= temp.next;
  }
}
System.out.println(length());
```

9. Consider the partial `List` class (storing integers) on the right. You may assume the class has a constructor and methods to add elements to lists, which we didn't show because their details are not necessary for this problem, but using them lists can be properly built. The values stored in lists of this class are not necessarily going to be in any particular order.

```
public class List {

  private static class Node {
    int data;
    Node next;
  }

  private Node head= null;

  void removeNth(int n);

}
```

Write the method `removeNth(int n)`, which should modify its current object list by removing the element at position `n` from it (where the element in the first node is considered to be at position 0, the one in the next node at position 1, etc.). For example, if a list `l` was storing the values 8, 1, 3, 6, 2, and 7 (shown in order from the beginning of the list) after the call `l.removeNth(2)`, the values in `l` would be (in order) 8, 1, 6, 2, and 7

The list the method is called upon may be empty (an empty list is represented by `head` being `null`), or have any number of zero or more elements, in any order, and your method must work properly, without any errors, regardless of its length. An empty list is indicated by `head` being `null`. `n` may have any integer value, but if for any reason there is no `n`'th element or position in the list the method should not modify the list, and should instead throw a `NoSuchElementException`.

Your method **may not call any methods** of the `List` or `Node` classes, since you know there must be some but you're not told exactly what they are or what they do. Similarly you can't assume there are any fields other than those shown. Do not write helper methods. The method **may not** copy data values between nodes or create new nodes. It must simply **remove** the `n`'th node from the list.

Comments are optional, but if you have time to come back and write short ones at the end of the exam it might help you get more partial credit in case of mistakes. Write your method ***neatly and legibly*** below, with good style and formatting.

10. The singly–linked `List` class on the right stores integers. It has various methods that aren't shown, because they're not needed for this problem. Elements are not stored in any particular order (it's **not** an ordered list.) Note that an empty list is represented by `head` being `null` (so a dummy head node is **not** being used). The `List` and `Node` classes **only** have the fields that you see.

```
public class List {

  private static class Node {
    int data;
    Node next;
  }

  private Node head= null;

  ⋮

}
```

Write the method `void removeThroughElt(int element)`. It should modify its current object list by removing all of its elements from the beginning of the list up through and **including** the **first** occurrence of its parameter `element`. However, if `element` is not present in the current object list, the list should not be modified.

For example, if a list named `list` contains the elements 9, 2, 4, 6, 8, 4, and 3, in this order, then after calling `list.removeThroughElt(6)`, the list would contain 8, 4, and 3 (everything up through and including the 6 was removed). But if `list.removeThroughElt(4)` had been called, the list would contain 6, 8, 4, and 3. And `list.removeThroughElt(1)` would have no effect.

Your method should work properly, without errors, regardless of the length of the list it's called upon– its current object list may be empty, or have any number of zero or more elements, and the parameter `element` may be present any number of times in the list, including not at all. For full credit your method:

- Should be **efficient**, meaning it should not do unnecessary work. (For example, it should not traverse parts of the list that it doesn't have to, or make unneeded traversals.)
- May **not** call any other methods of the `List` or `Node` classes, since there must be some other methods in them, but you're not told exactly what they are or what they do.
- May **not** copy values between nodes, or create new nodes (it **can** create **references** to nodes though).

Comments are optional, but if you have time to come back and write short ones at the end of the exam it might help you get more partial credit in case of mistakes. Write your method ***neatly and legibly*** below, with good style and formatting.