Exam 1
Recursion, Induction, and Object-Oriented Programming

Name:	NetID:

You have until 11:15 to complete this exam, though we don't expect you will need the whole time. As we noted in class on Friday, you may need to use the `instanceof` operator on some problems. As an example from the second assignment, `john instanceof Male` evaluates to `true` if the object `john` is an instance of class `Male` or any of its subclasses. (So if `john` is a `Male`, then `john instanceof Person` also evaluates to `true`.)

Problems

1. **Recursion**

   The `Word` class, partially implemented below, represents words as linked lists of letters. Fill in the incomplete methods in `Empty` and `Letter` so that `toString` and `equals` work. Notice that the type of the parameter to `equals` is `Object`, not `Word`!

```
public abstract class Word {
  public static Word fromString(String in) {
    return fromString_startingAt(in, 0);
  }
  private static Word fromString_startingAt(String in, int pos) {
    if (in.length() > pos)
      return new Letter(in.charAt(pos),
                        fromString_startingAt(in, pos+1));
    else
      return new Empty();
  }
  public abstract String toString();
  public abstract boolean equals(Object other);
  public Word reverse() {
    return reverse_helper(new Empty());
  }
  protected abstract Word reverse_helper(Word temp);
  public boolean isPalendrome() {
    return equals(reverse());
  }
}
```

```
class Empty extends Word {
  public String toString() { // -- Complete this




  }
  public boolean equals(Object other) { // -- Complete this




  }
  protected Word reverse_helper(Word temp) {
    return temp;
  }
}
```

```
class Letter extends Word {
  char first;
  Word rest; // not null

  public Letter(char first, Word rest) {
    this.first = first;
    this.rest = rest;
  }
  public String toString() { // -- Complete this




  }
  public boolean equals(Object other) { // -- Complete this




  }
  protected Word reverse_helper(Word temp) {
    return rest.reverse_helper(new Letter(first, temp));
  }
}
```

2. **Induction**

   Prove by structural induction that `reverse_helper` is correct. Be sure to clearly state your inductive hypothesis and point out where you make use of it.

3. **Inheritance**

   Hopefully everyone is familiar with musical instruments. For our purposes, an instrument is a mechanism that generates sound. All musical instruments have common properties such as minimum and maximum pitch. Furthermore, musical instruments can perform common actions, such as playing through the aforementioned range.

   In general, we can categorize musical instruments into four different groups: strings, woodwinds, brass, and percussion. For now we will assume that any instrument can be classified as one of these four types. Each of the four categories have their own properties in addition to those of just an instrument.

   A string instrument is an instrument that creates sound by the bowing or plucking one of the string of the instrument. Examples of string instruments are the violin, the double bass, the harp, and the guitar. String instruments have properties such as the number of strings and whether they are bowed or plucked. A common string instrument action is tuning.

   A percussion instrument is an instrument that creates sound by striking or shaking it. Examples of percussion instruments are the xylophone, the marimba, the snare drum, the tambourine, and castanets. Percussion instruments have properties such whether or not it plays a single pitch and whether or not it is a membranophone (instrument sounded from tightly stretched membranes) vs an idiophone (instrument sounded by hitting the actual instrument). A common percussion instrument action is to strike it.

   I won't describe the other two categories of woodwinds and brass here, but I can always tell you about them some other time.

Use the above information in the following questions:

(a) Suppose we would like an Instrument class. Write the Java code for it below. What are the fields and method(s)? Don't forget to write a constructor (one non-empty one will suffice). You don't have to fill in the body of your method (though you do need to write the constructor body).

(b) Suppose I'd like to make a class each for StringInstrument and PercussionInstrument. Write the Java code for it below. What would the fields and methods be for each type of instrument? Don't forget a non-empty constructor for each! Again, you can ignore the body of the method.

(c) Draw an inheritance diagram representing the hierarchy described.

(d) A few months ago I was walking through Collegetown at 1:30am on a Friday night. There was a trio out on the street across from CTB playing old-style blues music. There was an acoustic guitar player, a harmonica player, and this dude playing a piece of string attached by a pole to an upside-down metal washing tub. True story, I swear. He was using this home-made instrument as (1) a bass (a string instrument), and (2) a drum (a percussion instrument). So this nifty contraption was both a string instrument *and* a percussion instrument!

Suppose I'd like to make a new class in Java to encompass this string-percussion instrument. Given the above classes, what particular problem will I encounter? What is the solution to this problem?

4. **Short Answer**

(a) What are two advantages of encapsulation/information hiding?

(b) What is the difference between a public field and a protected field?

(c) Suppose class `SciFiMovie` and class `ComedyMovie` are subclasses of class `Movie`. Which of the following are legal?

```
ComedyMovie c = new Movie();
```

```
Movie       m = new SciFiMovie();
```

```
SciFiMovie  s = new ComedyMovie();
```

(d) Why would you make a method (and hence class) abstract?

(e) As a reminder, the Iterator interface is as follows:

```
interface Iterator {
    boolean hasNext();
    Object  next();
    void    remove();
}
```

I have the following incomplete implementation of a `IntArrayIterator` class, that iterates over an array of integers. Fill in the remaining portion of the `next()` method.

```
class ListIterator implements Iterator {
    int[] myArray;  // an array over which I want to iterate
    int   cursor;   // my current position in my array

    // constructor
    public ListIterator( int[] myArray ) { this.myArray = myArray; }

    // Returns true if there is another element to return
    boolean hasNext() { return cursor < myArray.length; }

    // Returns the next element to return
    Object next() throws NoSuchElementException {

        // If there are no elements left, throw an exception
        if( !hasNext() )
            throw new NoSuchElementException();

        // Otherwise, return the next element, etc.
        else {
            // FILL IN HERE









        }
    }
}
```