Exam 1 Solutions
Recursion, Induction, and Object-Oriented Programming

Problems

1. **Recursion**

```
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
    return "";
  }
  public boolean equals(Object other) { // -- Complete this
    return other instanceof Empty;
  }
  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
    return first + rest.toString();
  }
  public boolean equals(Object other) { // -- Complete this
    return other instanceof Letter
      && first == ((Letter)other).first
      && rest.equals(((Letter)other).rest);
  }
  protected Word reverse_helper(Word temp) {
    return rest.reverse_helper(new Letter(first, temp));
  }
}
```

2. **Induction**

   Proof of correctness of `reverse_helper`:

   P(`e`): `e.reverse_helper(e2)` for any `Word e2` returns the `Word` consisting of the letters of `e` in reverse order followed by the letters of `e2`.

   Proof by structural induction:

   Base case: If `e` is `Empty`, then `e2` is returned, so P(`e`) is true.

   Induction step: Assume that P(`e`) is true for some `Word e`. (This is our inductive hypothesis.) Construct an `e1 = new Letter(c,e)`. Then `e1.reverse_helper(e2)` returns the value `e.reverse_helper(new Letter(c,e2))`, which by the inductive hypothesis is the `Word` consisting of the letters of `e` in reverse order followed by `c` and the letters of `e2`. This is the same as the letters of `e1` in reverse order followed by the letters of `e2`, so P(`e1`) is true.

3. **Inheritance**

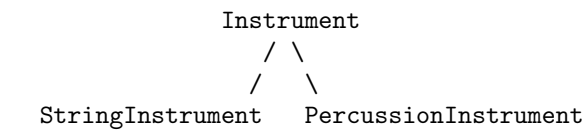
   (a) Suppose we would like an Instrument class. Write the Java code for it below. What are the fields and method(s)? Don't forget to write a constructor (one non-empty one will suffice). You don't have to fill in the body of your method (though you do need to write the constructor body).

```
class Instrument {

    // Fields
    protected int   minPitch;
    protected int   maxPitch;

    // Constructor
    public Instrument( int minPitch, int maxPitch ) {
        this.minPitch = minPitch;
        this.maxPitch = maxPitch;
    }

    // Method
    public void playRange() { }
}
```

   (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.

See the listing on the next page.

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



   (d) 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?

> The problem is that I would like to create a new class that extends both `StringInstrument` and `PercussionInstrument`. Java only allows you to extend one class, though. The solution to this problem is interfaces. I would create one interface for `Instrument`, one for `StringInstrument`, and one for `PercussionInstrument`. My hybrid class would implement the combination of these three interfaces.

4. **Short Answer**

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

   - It places related code together. (encapsulation)
   - It divides code into objects that model real-life entities. (encapsulation)

```
class StringInstrument extends Instrument {
    // Fields
    protected int     numberOfStrings;
    protected boolean isBowed;

    // Constructor
    public StringInstrument( int minPitch, int maxPitch,
                             int numberOfStrings, boolean isBowed ) {
        super( minPitch, maxPitch );
        this.numberOfStrings = numberOfStrings;
        this.isBowed         = isBowed;
    }

    // Method
    public void tune() { }
}

class PercussionInstrument extends Instrument {
    // Fields
    protected boolean isSinglePitch;
    protected boolean isMembranophone;

    // Constructor
    public PercussionInstrument( int minPitch, int maxPitch,
                                 boolean isSinglePitch,
                                 boolean isMembranophone ) {
        super( minPitch, maxPitch );
        this.isSinglePitch   = isSinglePitch;
        this.isMembranophone = isMembranophone;
    }

    // Method
    public void strike() { }
}
```

- It prevents abusive users from accessing/modifying the inner workings of an object. (information hiding)
- It lets the programmer make code changes without affecting the user's code. (information hiding)

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

A public field can be accessed from outside the class and is inherited. A protected field cannot be accessed from outside the class, but is also inherited.

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

```
ComedyMovie c = new Movie();        // illegal
Movie       m = new SciFiMovie();   // legal
SciFiMovie  s = new ComedyMovie();  // illegal
```

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

If there were a method, `m` say, that I would like all subclasses to have, but has no meaning in the superclass itself then I would make `m` abstract in the superclass. This would declare `m()`'s signature in the abstract superclass, and all subclasses would either have to implement `m` or be abstract themselves.

(e) 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
            return myList[cursor++];
        }
    }
}
```