Section II

1. (a)

```
public Critter(double probBreeding)
{
    super();
    myProbBreeding = probBreeding;
    myBreed = false;
}
```

(b)

```
private void breed()
{
    ArrayList<Location> breedLocs =
          getGrid().getEmptyAdjacentLocations(getLocation());
    for (Location loc : breedLocs)
    {
        Critter c = new Critter(myProbBreeding);
        c.putSelfInGrid(getGrid(), loc);
    }
}
```

(c)

```
public void act()
{
    if (getGrid() == null)
        return;
    ArrayList<Actor> actors = getActors();
    processActors(actors);
    myBreed = Math.random() < myProbBreeding;
    if(myBreed)
    {
        breed();
    }
    else
    {
        ArrayList<Location> moveLocs = getMoveLocations();
        Location loc = selectMoveLocation(moveLocs);
        makeMove(loc);
    }
}
```

NOTE

- In part (a), the line `super()` is optional. The compiler automatically calls the constructor of the superclass to initialize the inherited private instance variables (like color, direction, and so on.) Since the `Actor` class has a default constructor, omitting the line will not cause an error. If you *do* include the line, it must be the first line of code in the implementation of the constructor.
- In part (c), the first three statements are executed whether the `Critter` breeds or not. In other words, first the `Critter` eats, then it either breeds or moves.

2. (a)
```
public void printNotes()
{
    int count = 1;
    for(String note : noteList)
    {
        System.out.println(count + ". " + note);
        count++:
    }
}
```

Alternative solution for part (a):

```
public void printNotes()
{
    for (int index = 0; index < noteList.size(); index++)
    {
        System.out.println(index + 1 + ". "
                    + noteList.get(index));
    }
}
```

(b)
```
public void removeNotes(String word)
{
    int index = 0;
    while (index < noteList.size())
    {
        String note = noteList.get(index);
        if (note.indexOf(word) == -1)
            index++;
        else
            noteList.remove(index);
    }
}
```

Alternative solution for part (b):

```
public void removeNotes(String word)
{
    Iterator<String> itr = noteList.iterator();
    while (itr.hasNext())
    {
        String note = itr.next();
        if (note.indexOf(word) != -1)
            itr.remove();
    }
}
```

NOTE

- In part (b), you should increment the index only if you don't remove a note. This is because removing an element causes all notes following the removed item to shift one slot to the left. If, at the same time, the index moves to the right, you may miss elements that need to be removed.
- The alternative solution for part (b) uses an iterator, which is not part of the Level A Java subset. This solution, however, is included for those of you who have learned iterators because it is less complicated than the given solution. If you use an iterator to remove items from a list, you don't need to worry about the index: The iterator keeps track.

3. (a)
```
public boolean equals(Coin c)
{
    return myValue == c.myValue && myName.equals(c.myName);
}
```

Alternatively,

```
public boolean equals(Coin c)
{
    return getValue() == c.getValue() &&
            getName().equals(c.getName());
}
```

(b)
```
public int getTotal()
{
    int sum = 0;
    for (Coin c : myCoins)
    {
        sum += c.getValue();
    }
    return sum;
}
```

(c)
```
public int roundTotal()
{
    int cents = getTotal();
    return (cents + 50) / 100;
}
```

Alternatively,

```
public int roundTotal()
{
    double dollars = getTotal() / 100.0;
    return (int) (dollars + 0.5);
}
```

(d)
```
public int howMany(Coin c)
{
    int count = 0;
    for (Coin aCoin : myCoins)
    {
        if (c.equals(aCoin))
            count++;
    }
    return count;
}
```

NOTE

- In part (a), it is OK for the parameter `c` to access the private instance variables of `Coin`, since it too is a `Coin`.
- The solution as given in part (a) returns the truth value of the compound boolean expression. This single line of code is a compact form of

```
if (getValue() == c.getValue() && getName().equals(c.getName())
    return true;
else
    return false;
```

- In part (c), use `getTotal`. Do not re-implement the code for finding the total. You will not earn full credit on the AP exam if you do that.
- The alternative solution for part (c) uses floating-point (real number) division, not integer division. Another way to do the real number division is to cast to `double`:

```
double dollars = (double) getTotal() / 100;
```

- In part (d), use the `equals` method that you wrote in part (a). Do not re-implement the code.

4. (a)
```
public class Employee implements PersonalInfo
{
    private String myName;
    private String myCity;
    private boolean isUSCitizen;
    private double mySalary;

    public Employee(String name, String city,
        boolean isCitizen, double salary)
    {
        myName = name;
        myCity = city;
        isUSCitizen = isCitizen;
        mySalary = salary;
    }

    public String getName()
    { return myName; }

    public String getCity()
    { return myCity; }

    public boolean getCitizenStatus()
    { return isUSCitizen; }

    public double getSalary()
    { return mySalary; }
}
```

(b)
```
public class PartTimeEmployee extends Employee
{
    private double whatFraction;
    private boolean isUnionMember;

    public PartTimeEmployee(String name, String city, boolean
        isCitizen, double salary, double fraction, boolean isMember)
    {
        super(name, city, isCitizen, salary);
        whatFraction = fraction;
        isUnionMember = isMember;
    }

    public double getFraction()
    { return whatFraction; }

    public boolean getUnionStatus()
    { return isUnionMember; }

    public void changeUnionStatus()
    {
        if (getSalary() > 15000)
            isUnionMember = !isUnionMember;
    }
}
```

NOTE

- All methods in the `PersonalInfo` interface must be implemented in the `Employee` class, since the class implements `PersonalInfo`.
- Since `PartTimeEmployee` is a subclass of `Employee`, it too implements `PersonalInfo`, and inherits the implemented methods from the `Employee` superclass. These methods should not be rewritten.
- In part (b), `changeUnionStatus` must access the salary field of the superclass. It *must* use the accessor method, `getSalary`, since it cannot access the private instance variable, `mySalary`.
- In part (b), you must write a constructor for `PartTimeEmployee`. Recall that constructors are not inherited. Also, you must use `super` to initialize the inherited data fields `myName`, `myCity`, `isUSCitizen`, and `mySalary`.