CSE 142, Autumn 2011
Final Exam
(*key at end)*

1. Array Mystery

Consider the following method:

```
public static void mystery(int[] a) {
    for (int i = 1; i < a.length - 1; i++) {
        a[i] = (a[i - 1] + a[i + 1]) / 2;
    }
}
```

Indicate in the right-hand column what values would be stored in the array after the method `mystery` executes if the array in the left-hand column is passed as its parameter.

Original Contents of Array	Final Contents of Array
`int[] a1 = {1, 1, 3}` `mystery(a1);`	_____
`int[] a2 = {2, 1, 2, 4};` `mystery(a2);`	_____
`int[] a3 = {6, 13, 0, 3, 7};` `mystery(a3);`	_____
`int[] a4 = {-1, 6, 3, 5, -3};` `mystery(a4);`	_____
`int[] a5 = {7, 2, 3, 1, -3, 12};` `mystery(a5);`	_____

2. Reference Semantics Mystery

Write the output of the following program below, as it would appear on the console.

```
public class ReferenceMystery {
    public static void main(String[] args) {
        String name = "Janet";
        int money = 30;
        Account a = new Account(name, money);

        mystery(name, money, a);
        System.out.println(name + ", " + money + ", " + a);

        money = money + 10;
        a.name = "Billy";

        mystery(name, money, a);
        System.out.println(name + ", " + money + ", " + a);
    }

    public static void mystery(String name, int money, Account a) {
        a.money++;
        name = "Susan";
        System.out.println(name + ", " + money + ", " + a);
    }
}

public class Account {
    String name;
    int money;

    public Account(String name, int money) {
        this.name = name;
        this.money = money;
    }

    public String toString() {
        return name + ": $" + money;
    }
}
```

3. Inheritance Mystery

Assume that the following four classes have been defined:

```
public class Tulip extends Rose {
   public void method1() {
      System.out.print("Tulip 1 ");
   }
}

public class Violet {
   public void method1() {
      System.out.print("Violet 1 ");
   }

   public void method2() {
      System.out.print("Violet 2 ");
   }

   public String toString() {
      return "Violet";
   }
}
```

```
public class Rose extends Lily {
   public String toString() {
      return "Rose " + super.toString();
   }
}

public class Lily extends Violet {
   public void method1() {
      super.method1();
      System.out.print("Lily 1 ");
   }

   public void method2() {
      System.out.print("Lily 2 ");
      method1();
   }

   public String toString() {
      return "Lily";
   }
}
```

Given the classes above, what output is produced by the following code?

```
Violet[] pretty = { new Tulip(), new Lily(), new Violet(), new Rose() };

for (int i = 0; i < pretty.length; i++) {
   System.out.println(pretty[i]);
   pretty[i].method1();
   System.out.println();
   pretty[i].method2();
   System.out.println();
   System.out.println();
}
```

4. File Processing

Write a static method named **`groceries`** that accepts as its parameter a `Scanner` for an input file. The data in the `Scanner` represents grocery items purchased along with their price and their discount category. Your method should compute and return a double representing the total cost of the grocery items.

Each item is represented by three tokens starting with the name of the item (a single word) followed by its discount category ("red", "blue" or "none") followed by its full price. The discount category may include capitalization. The different discount options are:

- red: 10% off full price
- blue: 25% off full price
- none: full price

For example, given a `Scanner` named `input` referring to an input file that contains the following text:

```
avocado RED 1  blueberries none 5 milk blue
   2.00       cream                  red 1.00    cereal None 1.29
```

The call on `groceries(input)` should return `9.59`.

The avocado will cost $0.9 because a discount of 10% off of $1 is $0.1. Blueberries cost the full price of $5. Milk will cost $1.50 because it receives a discount of 25% off of $2.00. Cream will cost $0.9 and cereal will cost the full price of $1.29. The total is 0.9 + 5 + 1.5 + .9 + 1.29 = 9.59.

Notice that the input may span multiple lines and may have different spacing between tokens. The entire file represents a single grocery bill.

You may assume that the input file exists and has the format described above. The file will always contain at least one grocery item and will always contain a number of tokens that is a multiple of 3. The second token in every triple will always be one of "red", "blue" or "none".

5. File Processing

Write a static method named **`mostUnique`** that accepts as its parameter a Scanner for an input file. The data in the Scanner represents integer quiz scores separated by spaces. Each class period is on its own line and contains a different number of students. Your method should return the highest number of unique scores given in a single class period. The method should also print the number of unique scores given in each period.

On a given line, repeated scores are always next to each other.

For example, given a `Scanner` named `input` referring to an input file that contains the following text:

```
10 10 10 9 9 8 3
3 3 8 10 9 7 7 6 6
4 1 9 9 10 7 7
10 10 10 10
```

The call on `mostUnique(input)` should return `6` and generate the following output:

```
Period 1: 4 unique scores
Period 2: 6 unique scores
Period 3: 5 unique scores
Period 4: 1 unique scores
```

On the first line, there are 4 unique scores: 10, 9, 8 and 3. The second line contains 6 unique scores: 3, 8, 10, 9, 7 and 6. The third line contains 5 unique scores: 4, 1, 9, 10 and 7. The fourth line only has one unique score: 10. The value returned is 6 because it is the highest number of unique scores given in a class period.

Assume that the file exists, that it contains at least one line of data and that each line contains at least one score.

6. Array Programming

Write a static method named **`repeatedSequence`** that accepts two arrays of integers *a1* and *a2* as parameters and returns `true` if *a2* is composed entirely of repetitions of *a1* and `false` otherwise. For example, if *a1* stores the elements `{2, 1, 3}` and *a2* stores the elements `{2, 1, 3, 2, 1, 3, 2, 1, 3}`, the method would return `true`.

If the length of *a2* is not a multiple of the length of *a1*, your method should return `false`. You may assume that both arrays passed to your method will have a length of at least 1.

The following table shows some calls to your method and their expected results:

Arrays	Returned Value
`int[] a1 = {2, 1};` `int[] a2 = {2, 1, 2, 1, 2, 1};`	`repeatedSequence(a1, a2)` returns `true`
`int[] a3 = {2, 1, 3};` `int[] a4 = {2, 1, 3, 2, 1, 3, 2};`	`repeatedSequence(a3, a4)` returns `false`
`int[] a5 = {23};` `int[] a6 = {23, 23, 23, 23};`	`repeatedSequence(a5, a6)` returns `true`
`int[] a7 = {5, 6, 7, 8};` `int[] a8 = {5, 6, 7, 8};`	`repeatedSequence(a7, a8)` returns `true`
`int[] a9 = {5, 6};` `int[] a10 = {5, 6, 7, 5, 6, 5};`	`repeatedSequence(a9, a10)` returns `false`

Do not modify the contents of the arrays passed to your method as parameters.

7. Array Programming

Write a static method named **`evenOdd`** that accepts an array of integers as a parameter and rearranges the array's elements so that all of its odd elements are in positions with odd-numbered indexes and all of its even elements are in positions with even-numbered indexes. The array passed in will always contain exactly as many even values as odd values. The exact order of the elements in the array after your method is run on it is unimportant as long as its content alternates between even and odd values, starting with even ([*even value, odd value, even value, odd value…*]).

For example, if your method were passed the following array:

```
int[] a1 = {5, 6, 3, 3, 2, 5, 2, 6};
evenOdd(a1);
```

One acceptable ordering of the elements after the call would be:

```
[6, 5, 2, 3, 2, 5, 6, 3]
```

The even-numbered indexes of this array are 0, 2, 4, 6 and each of these positions contains an even integer. The odd-numbered indexes of this array are 1, 3, 5, 7 and each of these positions contains an odd integer.

You may not create an additional array or use a String to solve this problem.

8. Critters

Write a Critter class **`Grasshopper`** along with its movement and fighting behavior. All unspecified aspects of `Grasshopper` use the default behavior. Write the complete class with any fields, constructors, etc. necessary.

A `Grasshopper` sits still until getting into a fight. Once it fights, it celebrates its victory by doing a "hop". A "hop" consists of moving north a certain number of times, then south the same number of times, then west one time. The hops start with a height of 1 (one move north, then one move south) but each subsequent fight causes the next hop to be larger by one. The second hop is 2 moves north, 2 moves south, then 1 move west. After finishing the hop, the `Grasshopper` sits idle again until it gets into another fight.

If a `Grasshopper` is sitting still, it always fights with `Attack.ROAR`. If a `Grasshopper` gets into a fight in the middle of a hop (while it is not sitting still), it always returns `Attack.FORFEIT`, causing it to lose the fight.

Here is an example sequence of moves for one `Grasshopper`:

CCCCC *(fights)* NSWCCC *(fights)* NNSSWCCCCCCC *(fights)* NNNSSSWCC *(fights)* NNNNSS *(fights and dies)*

9. Classes and Objects

Suppose you have the class `Date` at right. (This is the same `Date` class from your homework. Only the headings are shown.)

Write an instance method **`rewind`** to be placed inside the `Date` class. The method accepts an integer representing a number of days as a parameter and modifies the `Date`'s state by moving it backward in time by that many days. You may assume that the value passed is non-negative, but it could be very large, causing the `Date` to wrap into previous month(s) or year(s).

For example, if these `Date` objects are declared in client code:

```
Date jan10 = new Date(1, 10);
Date sep19 = new Date(9, 19);
Date nov30 = new Date(11, 30);
```

The following calls should change their state as indicated:

```
sep19.rewind(0);          // 9/19
sep19.rewind(5);          // 9/14
sep19.rewind(15);         // 8/30
jan10.rewind(11);         // 12/30 (of last year)
nov30.rewind(365 + 364); // 12/1 (two years ago)
```

```
// this class ignores leap years
public class Date {
  private int month;
  private int day;

  // constructs the given month/day
  public Date(int m, int d)

  // returns the day/month fields
  public int getDay()
  public int getMonth()

  // # of days in date's month
  public int daysInMonth()

  // compares dates (true if same)
  public boolean equals(Date d)

  // modifies this date's state
  // forward in time by 1 day,
  // wrapping month/year if needed
  public void nextDay()

  // set month/date to given values
  public void setDate(int m, int d)

  // your method would go here
}
```

Solutions

1. Array mystery.

```
[1, 2, 3]

[2, 2, 3, 4]

[6, 3, 3, 5, 7]

[-1, 1, 3, 0, -3]

[7, 5, 3, 0, 6, 12]
```

2. Reference mystery.

```
Susan, 30, Janet: $31
Janet, 30, Janet: $31
Susan, 40, Billy: $32
Janet, 40, Billy: $32
```

3. Inheritance mystery.

```
Rose Lily
Tulip 1
Lily 2 Tulip 1

Lily
Violet 1 Lily 1
Lily 2 Violet 1 Lily 1

Violet
Violet 1
Violet 2

Rose Lily
Violet 1 Lily 1
Lily 2 Violet 1 Lily 1
```

4. Groceries, 10 points. Two of many possible solutions follow:

```
// reads price in each branch, 3 counters, calculates discount at end
public static double groceries(Scanner s) {
   double red = 0;
   double blue = 0;
   double none = 0;
   while (s.hasNext()) {
      s.next();
      String sale = s.next();

      if (sale.equalsIgnoreCase("red")) {
         red += s.nextDouble();
      } else if (sale.equalsIgnoreCase("blue")) {
         blue += s.nextDouble();
      } else {
         none += s.nextDouble();
      }
   }
   return red * .90 + blue * .75 + none;
}
```

```
// reads price once before conditional, adds in each branch
public static double groceries(Scanner s) {
   double total = 0;
   while (s.hasNext()) {
      s.next();
      String sale = s.next();
      double value = s.nextDouble();
      if (sale.equalsIgnoreCase("red")) {
         total += value * .90;
      } else if (sale.equalsIgnoreCase("blue")) {
         total += value * .75;
      } else {
         total += value;
      }
   }
   return total;
}
```

5. Most Unique, 10 points. Two of many possible solutions follow:

```
// fencepost solution
public static int mostUnique(Scanner fileScan) {
    int most = 1;
    int line = 0;
    while (fileScan.hasNextLine()) {
        line++;
        String line = fileScan.nextLine();
        Scanner lineScan = new Scanner(line);
        int previous = tokens.nextInt();
        int count = 1;
        while (lineScan.hasNextInt()) {
            int current = lineScan.nextInt();
            if (previous != current) {
                count++;
            }
            previous = current;
        }
        System.out.println("Period " + line + ": " + count + " unique scores");
        if (count > most) {
            most = count;
        }
    }
    return most;
}

// Math.max and -1 bootstrap solution
public static int mostUnique(Scanner input) {
    int max = 1;
    int line = 1;
    while (input.hasNextLine()) {
        Scanner tokens = new Scanner(input.nextLine());
        int prev = -1;
        int count = 0;
        while (tokens.hasNextInt()) {
            int curr = tokens.nextInt();
            if (prev != curr) {
                count++; }
            previous = curr; }
        System.out.println("Period " + line + ": " + count + " unique scores");
        max = Math.max(max, count);
        line++; }
    return max;
}
```

6. Repeated Sequence, 15 points. Seven of many possible solutions follow:

```
// "nested loops, increment outer loop by a1.length" solution
public static boolean repeatedSequence(int[] a1, int[] a2) {
    if (a2.length % a1.length != 0) {
        return false;
    }

    for (int i = 0; i < a2.length; i += a1.length) {
        for (int j = 0; j < a1.length; j++) {
            if (a1[j] != a2[i + j]) {
                return false;
            }
        }
    }

    return true;
}

// "nested loops, increment outer loop by 1, but with inner if test" solution
public static boolean repeatedSequence(int[] a1, int[] a2) {
    if (a2.length % a1.length != 0) {
        return false;
    }

    for (int i = 0; i < a2.length; i++) {
        if (i % a1.length == 0) {
            for (int j = 0; j < a1.length; j++) {
                if (a1[j] != a2[i + j]) {
                    return false;
                }
            }
        }
    }

    return true;
}

// "outer loop divides, inner index re-multiplies" solution
public static boolean repeatedSequence(int[] a1, int[] a2) {
    if (a2.length / a1.length * a1.length != a2.length) {
        return false;
    }

    for (int i = 0; i < a2.length / a1.length; i++) {
        for (int j = 0; j < a1.length; j++) {
            if (a1[j] != a2[j + i * a1.length]) {
                return false;
            }
        }
    }
    return true;
}


// "based on smaller array" solution
public static boolean repeatedSequence(int[] a1, int[] a2) {
    if (a2.length % a1.length != 0) {
        return false;
    }

    for (int i = 0; i < a1.length; i++) {
```

```
        for (int j = i; j < a2.length; j += a1.length) {
            if (a1[i] != a2[j]) {
                return false;
            }
        }
    }
    return true;
}


// "based on smaller array #2" solution
public static boolean repeatedSequence(int[] a1, int[] a2) {
    if (a2.length % a1.length != 0) {
        return false;
    }

    for (int i = 0; i < a1.length; i++) {
        for (int j = 0; j < a2.length; j++) {
            if (j % a1.length == i) {
                if (a1[i] != a2[j]) {
                    return false;
                }
            }
        }
    }
    return true;
}

// "single loop over a2" solution
public static boolean repeatedSequence(int[] a1, int[] a2) {
    if (a2.length % a1.length != 0) {
        return false;
    }

    for (int i = 0; i < a2.length; i++) {
        if (a2[i] != a1[i % a1.length]) {
            return false;
        }
    }
    return true;
}

// "single loop, external variable for index" solution
public static boolean repeatedSequence(int[] a1, int[] a2) {
    if (a2.length % a1.length != 0) {
        return false;
    }

    int i = 0;
    for (int j = 0; j < a2.length; j++) {
        if (a2[j] != a1[i]) {
            return false;
        }
        i++;
        if (i == a1.length) {
            i = 0;
        }
    }
    return true;
}
```

7. Even Odd, 10 points. Six solutions of many possible follow:

```
// "nested for loops with mod test and swap" solution
public static void evenOdd(int[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            if (a[j] % 2 == i % 2) {
                int temp = a[i];   // swap
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}


// "nested for loops with LESS ELEGANT mod test and swap" solution
public static void evenOdd(int[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = i; j < a.length; j++) {
            if (a[j] % 2 == 0 && i % 2 == 0 ||
                a[j] % 2 == 1 && i % 2 == 1) {
                int temp = a[i];    // swap
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}

// "outer even-only loop, inner odd-only loop" solution
public static void evenOdd(int[] a) {
    for (int i = 0; i < a.length; i += 2) {
        if (a[i] % 2 != 0) {
            for (int j = 1; j < a.length; j += 2) {
                if (a[j] % 2 == 0) {
                    int temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
    }
}

// "nested loops, convoluted inner tests" solution
public static void evenOdd(int[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = i + 1; j < a.length; j++) {
            if (i % 2 == 0 && a[i] % 2 == 1 && a[j] % 2 == 0) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            } else if (i % 2 == 1 && a[i] % 2 == 0 && a[j] % 2 == 1) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
```

```
// "inner while loop, looking for proper index" solution
public static void evenOdd(int[] a) {
    for (int i = 0; i < a.length; i++) {
        if (a[i] % 2 != i % 2) {
            int j = i;
            while (j < a.length - 1 && a[j] % 2 != i % 2) {
               j++;
            }
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
}

// "nested while loops; look for a bad even/odd value then swap them" solution
public static void evenOdd(int[] a) {
    int even = 0;
    int odd = 1;
    while (even < a.length || odd < a.length) {
        while (even < a.length && a[even] % 2 == 0) {
            even += 2;
        }
        while (odd < a.length && a[odd] % 2 != 0) {
            odd += 2;
        }

        if (even < a.length || odd < a.length) {
            int temp = a[even];
            a[even] = a[odd];
            a[odd] = temp;
        }
    }
}
```

8. Grasshopper, 15 points. Four of many possible solutions follow:

```
// "two ints and a boolean" solution
public class Grasshopper extends Critter {
    private boolean attacked = false;
    private int height = 0;
    private int moves = 0;

    public Direction getMove() {
        if (!attacked) {
            return Direction.CENTER;
        }

        moves++;
        if (moves <= height) {
            return Direction.NORTH;
        } else if (moves <= 2 * height) {
            return Direction.SOUTH;
        } else {
            moves = 0;
            attacked = false;
            return Direction.WEST;
        }
    }

    public Attack fight(String opponent) {
        if (moves == 0) {
```

```
            attacked = true;         // sitting idle; wants to fight
            height++;
            return Attack.ROAR;
        } else {
            return Attack.FORFEIT;   // in motion; forfeit
        }
    }
}

// "just two ints" solution
public class Grasshopper extends Critter {
    private int max = 0;
    private int move = 0;

    public Direction getMove() {
        if (move == 0) {
            return Direction.CENTER;
        } else if (move <= max) {
            move++;
            return Direction.NORTH;
        } else if (move <= 2 * max) {
            move++;
            return Direction.SOUTH;
        } else {
            move = 0;
            return Direction.WEST;
        }
    }

    public Attack fight(String opponent) {
        if (move == 0) {
            move++;              // idle; wants to fight
            max++;
            return Attack.ROAR;
        }
        return Attack.FORFEIT;   // in motion; forfeit
    }
}

// "separate north/south counters" solution
public class Grasshopper extends Critter {
    private int hopHeight = 1;
    private int north = 0;
    private int south = 0;
    private boolean moving = false;

    public Direction getMove() {
        if (!moving) {
            return Direction.CENTER;
        } else if (north < hopHeight) {
            north++;
            return Direction.NORTH;
        } else if (north == hopHeight && south < hopHeight) {
            south++;
            return Direction.SOUTH;
        } else {
            north = 0;
            south = 0;
            hopHeight++;
            moving = false;
            return Direction.WEST;
        }
```

```
    }

    public Attack fight(String opponent) {
        if (!moving) {
            moving = true;
            return Attack.ROAR;
        } else {
            return Attack.FORFEIT;
        }
    }
}

public class Grasshopper extends Critter {
    private int height = 0;
    private Direction lastMove = Direction.CENTER;
    private int movesInRow = 0;

    public Direction getMove() {
        if (lastMove == Direction.CENTER) {
            return Direction.CENTER;
        } else if (lastMove == Direction.WEST) {
            lastMove = Direction.CENTER;
        } else if (movesInRow < height) {
            movesInRow++;
        } else if (movesInRow == height) {
            movesInRow = 1;
            if (lastMove == Direction.NORTH) {
                lastMove = Direction.SOUTH;
            } else if (lastMove == Direction.SOUTH) {
                lastMove = Direction.WEST;
            }
        }
        return lastMove;
    }

    public Attack fight(String opponent) {
        if (lastMove == Direction.CENTER) {
            movesInRow = 0;
            height++;
            lastMove = Direction.NORTH;
            return Attack.ROAR;
        } else {
            return Attack.FORFEIT;
        }
    }
}
```

9. Rewind, 10 points. Six of many possible solutions follow:

```
// "enumerate 3 cases separately, set month/date exactly once" solution
public void rewind(int days) {
    for (int i = 0; i < days; i++) {
        if (month == 1 && day == 1) {
            month = 12;
            day = 31;
        } else if (day == 1) {
            month--;
            day = daysInMonth();
        } else {
            day--;
        }
    }
}
```

```
// "subtract then check for wrap" solution
public void rewind(int days) {
    for (int i = 0; i < days; i++) {
        day--;
        if (day == 0) {
            month--;
            if (month == 0) {
                month = 12;
            }
            day = daysInMonth();
        }
    }
}

// "calls setDate" solution
public void rewind(int days) {
    for (int i = 0; i < days; i++) {
        if (month == 1 && day == 1) {
            setDate(12, 31);
        } else if (day == 1) {
            setDate(month - 1, day);
            setDate(month, daysInMonth());
        } else {
            setDate(month, day - 1);
        }
    }
}

// "ninja, yet slow, rewinding by 1 means adding 364 days" solution
public void rewind(int days) {
    for (int i = 0; i < days * 364; i++) {
        nextDay();
    }
}

// "ninja, rewinding by 1 means adding 364 days" solution
public void rewind(int days) {
    for (int i = 0; i < 365 - days % 365; i++) {
        nextDay();
    }
}

// "while loop to count down the days" solution
public void rewind(int days) {
    while(days > 0) {
        day--;
        days--;
        if(day == 0) {
            month--;
            if (month == 0) {
                month = 12;
            }
            day = this.daysInMonth();
        }
    }
}
```