Midterm Exam

General description

There are 4 problems, some subdivided into smaller questions. Do as many as you can, or as much of a bigger problem as you can. There should be enough time to finish, or at least to start working on all the questions. Feel free to use extra paper, but be sure to put your name and which question is being answered at the top. Start with the ones that you consider to be easiest, then move to those that may take more time.

All code is and should be in ActionScript 3.0. If not explicitly stated, you should not be considering graphics (i.e., you should not be using MovieClips or frames). Be generous with comments. Part of the grade will be based on how readable and neat your code is.

1. Basic programming concepts (30 points)
   a. What is the following (partial) program doing? Explain in a few short sentences.

```javascript
var data : Array;
var n : int;
var flag : int;

... // the flag variable gets a value somewhere in here ...

n = 5;
data = new Array(n);
data[0] = 1;

if ( (flag*flag >= 0) )
{
    for ( var i : int = 1; i < n; i++ )
    {
        data[i] = i*data[i-1];
    }
}
else
{
    for ( var i :int = 1; i < n; i++ )
    {
        data[i] = flag*data[i-1];
    }
}
```

The data Array is created with length 5. Since the square of a number is always positive, the if part is always executed. This fills the data array with factorial values; i.e., data[i] = i!
b. A Magic Square is an nxn grid of integers, whose rows, columns and diagonals each add up to the same “magic sum.” For instance, the following is a Magic Square as each row, column and diagonal sum to 15:

\[
\begin{array}{ccc}
2 & 7 & 6 \\
9 & 5 & 1 \\
4 & 3 & 8 \\
\end{array}
\]

Implement a method (using the following skeleton) that takes in a two-dimensional Array full of int values and returns true if it is a Magic Square. You may assume that the Array passed to you has the same number of rows as columns. You are encouraged to define auxiliary methods that to help you.

```java
public function isMagicSquare( square : Array ) : Boolean {
    // store the potential magic sum
    var s : int = 0;

    // compute the potential magic sum using the first row
    for ( var j : int = 0; j < square[0].length; j++ )
        s = s + square[0][j];

    // check that all rows sum to s
    for ( var i : int = 1; i < square.length; i++ )
        if ( !checkRow( square, i, s ) )
            return false;

    // check that all columns sum to s
    for ( var j : int = 0; j < square.length; j++ )
        if ( !checkCol( square, j, s ) )
            return false;

    // check diagonals
    if ( !checkDiags( square, s ) )
        return false;

    // if we get here, everything added to s!
    return true;
}
```
private function checkRow( sq : Array, r : int, s : int ) :
  Boolean {
    // store the sum
    var sum : int = 0;
    // compute row sum
    for ( var j : int = 0; j < square[r].length; j++ )
      sum = sum + sq[r][j];
    // return true if row summed to passed s
    return s == sum;
  }

private function checkCol( sq : Array, c : int, s : int ) :
  Boolean {
    // store the sum
    var sum : int = 0;
    // compute column sum
    for ( var i : int = 0; i < square.length; i++ )
      sum = sum + sq[i][c];
    // return true if column summed to passed s
    return s == sum;
  }

private function checkDiags( sq : Array, s : int ) :
  Boolean {
    // store the sums
    var sum1 : int = 0;
    var sum2 : int = 0;
    // compute diagonals’ sums
    for ( var i : int = 0; i < sq.length; i++ ) {
      sum1 = sum1 + sq[i][i];
      sum2 = sum2 + sq[i][sq.length-i-1];
    }
    // return true if both diagonals summed to passed s
    return (s == sum1) && (s == sum2);
  }
c. Debug the following (or indicate that it has no bugs). If you find a bug, suggest a solution that would fix it.

```java
package {
    public class Mystery {
        private var array : Array;
        private var n : int;

        public function Mystery() {
            var m = f(2);
            n = 5;
            array = new Array(n);
        }

        public function f( x : int ) : int {
            for ( var i : int = 0; i < array.length; i++ ) {
                array[i] = x*array[i-1];
            }
        }
    }
}
```

**Bug #1**: m is missing a type  
Fix by changing to: `var m : int = f(2);`

**Bug #2**: f tries to access elements of array, which is not yet allocated  
Fix by moving “var m : int = f(2);” to end of constructor

**Bug #3**: when i – 0 in for loop, array[i-1] gives an out of bounds error  
Fix by adding “array[0] = 1;” before for loop and change for loop to start at 1: for ( var i : int = 1; ... )
d. **Arrays and loops (20 points)**
The Bell triangle is defined by first placing a 1 in row 1. The first element of subsequent rows is defined by repeating the last entry of the previous row. Each entry following is the sum of the entry to the left and the entry above and to the left. The first 5 rows of the Bell triangle is given below:

```
1
1  2
2  3  5
5  7 10 15
15 20 27 37 52
```

For instance, row 3 is computed by repeating the 2 from the last entry of row 2, then adding 1+2 to get 3 and 2+3 to get 5.

Write a method that computes the first $n$ rows of the Bell triangle and returns a two-dimensional array holding the result.

```java
// function to compute the first n rows of the Bell triangle
// return a two-dimensional array holding the result
public function bellTriangle( n : int ) : Array {
    // array to hold triangle
    var a : Array = new Array( n );
    // initialize the rows of the array
    // to each be an array of length i+1
    for ( var i : int = 0; i < a.length; i++ )
        a[i] = new Array( i+1 );
    // initialize row 0
    a[0][0] = 1;
    // fill out remaining rows
    for ( var i : int = 1; i < a.length; i++ ) {
        // first entry is last of previous row
        a[i][0] = a[i-1][i-1];
        // subsequent entries are sum of left and above
        for ( var j : int = 1; j < a[i].length; j++ )
            a[i][j] = a[i-1][j] + a[i-1][j-1];
    }
    // return result
    return a;
}
```
2. **Objects (30 points)**

In this problem, you will create a **Donut** ActionScript 3.0 class. Every donut (instance) should contain a frosting (such as “maple walnut,” “chocolate,” “peanut butter”) and a flag as to whether or not the donut contains nuts.

a. Define the **Donut** class; your code should have fewer than 7 lines (not including comments).

```actionscript
package {
    /**
     * The Donut class represents a Donut with frosting.
     **/
    public class Donut {
        private var frosting : String; // type of frosting
        private var nuts : Boolean; // true if contains nuts
    }
}
```

b. The **Donut** class definition should be in a file named ______ **Donut.as** _______.

c. Write a **Donut** constructor that takes in two arguments, for the frosting and whether or not the donut contains nuts. Your code should have fewer than 5 lines (not including comments).

```actionscript
/**
 * Constructor creates a donut with specified frosting (with nuts?)
 **/
public function Donut(
    frosting : String, nuts : Boolean ) {
    this.frosting = frosting; // type of frosting
    this.nuts = nuts; // true if contains nuts
}
```
d. Write the code for a `printAllergyInformation` method, which prints allergy information to the command line (trace), such as “This peanut butter frosted donut contains nuts” or “This chocolate frosted donut contains no nuts.” Your code should have fewer than 10 lines (not including comments).

```java
public function printAllergyInformation() : void {
    // string to be printed
    var s : String = "This " + frosting + " frosted donut contains ";
    // if nuts
    if ( nuts )
        s = s + "nuts";
    // otherwise, no nuts
    else
        s = s + "no nuts";

    // print it out
    trace( s );
}
```

e. Write the code for a `toString` method, which returns a String representation of the donut, such as “Mmm.. maple walnut frosted donut.” Your code should have fewer than 5 lines (not including comments).

```java
public function toString() : String {
    return "Mmm... " + frosting + " frosted donut";
}
```

f. Write a statement that declares a variable of type `Donut`.

```java
var donut : Donut;
```

g. Write a statement that creates (constructs) a `Donut` instance that represents a maple walnut frosted donut, (which contains nuts…), and assigns it to the variable declared in Part f.

```java
donut = new Donut( "maple walnut", true );
```

h. Write a statement that simultaneously declares a variable of type `Donut` and assigns to it a new `Donut` instance that represents a chocolate frosted donut, which does not contain nuts.

```java
var chocolate : Donut = new Donut( "chocolate", false );
```
i. Write a statement that simultaneously declares and allocates an array of length 3.

```javascript
var donuts : Array = new Array( 3 );
```

j. Write code to fill the array from step i with the maple walnut donut from steps f and g, the chocolate donut from step h, and another donut of your choosing. Your code should have fewer than 5 lines (not including comments).

```javascript
donuts[0] = donut;
donuts[1] = chocolate;
donuts[2] = new Donut( "cider", false );
```

k. Write code to print the allergy information for each item in the array filled by step j. Your code should have fewer than 5 lines (not including comments) and should use the method defined in step d.

```javascript
for ( var i : int = 0; i < donuts.length; i++ )
  donuts[i].printAllergyInformation();
```
3. **Inheritance (20 points)**

In this problem, you will create a class **FilledDonut** that represents a donut with a filling by subclassing the **Donut** class you defined in Problem 3. Every filled donut (instance) should contain (in addition to the basic donut properties) a type of filling (such as “almond,” “cream,” “chocolate”) and a flag as to whether or not the filling contains nuts.

a. Define the **FilledDonut** class; your code should have fewer than 7 lines (not including comments).

```java
package {
    /**
     * The FilledDonut class represents a frosted Donut with a filling.
     * with a filling.
     **/
    public class FilledDonut extends Donut {
        private var filling : String; // type of filling
        // true if filling contains nuts
        private var fillingNuts : Boolean;
    }
}
```

b. Write a **FilledDonut** constructor that takes in 4 arguments, for the types of frosting and filling and whether or not the frosting and filling contain nuts. Your code should have fewer than 6 lines (not including comments).

```java
/**
 * Constructor creates a filled donut with specified frosting (with nuts?) and filling (with nuts?)
 **/
public function FilledDonut(
    frosting : String, nuts : Boolean,
    filling : String, fillingNuts : Boolean ) {
    // call the superclass Donut constructor
    super( frosting, nuts );

    this.filling = filling; // type of frosting
    // true if contains nuts
    this.fillingNuts = fillingNuts;
}
```
c. Override the inherited method `printAllergyInformation` to print allergy information after taking the filling type into account. For instance, “This maple walnut frosted donut with cream filling contains nuts” or “This chocolate frosted donut with almond filling contains nuts” or “This chocolate frosted donut with strawberry filling does not contain nuts.” Your code should have fewer than 5 lines (not including comments).

Indicate how, if necessary, you would modify your implementation of the `Donut` class to accomplish your method.

```javascript
// override superclass method to take filling into account
public override function printAllergyInformation() : void {
    // if either the frosting or filling contains nuts
    if ( nuts || fillingNuts )
        trace( "This " + frosting + " frosted donut with " + filling + " filling contains nuts";)
    // otherwise, no nuts
    else
        trace( "This " + frosting + " frosted donut with " + filling + " filling does not contain nuts";)
}

The `Donut` class would need to be modified to make both the `frosting` and `nuts` fields protected instead of private.

d. Override the inherited method `toString` to display the type of filling as well as the original donut information, i.e., “Mmm… maple frosted donut with cream filling.” Your method definition should not explicitly use “Mmm…”

Indicate how, if necessary, you would modify your implementation of the `Donut` class to accomplish your method.

```javascript
// override superclass method to take filling into account
public override function toString() : String {
    // invoke superclass’ implementation
    return super.toString() + " with " + filling + " filling";
}
```

There is no modification necessary after part c.