First Name: _____ **Last Name**: _____

McGill ID: _____ **Section**: _____

Faculty of Science
COMP-202B - Introduction to Computing I (Winter 2009) - All Sections
Final Examination

Wednesday, April 29, 2009
9:00–12:00

Examiners: Mathieu Petitpas [Section 1]
Prof. Xue Liu [Section 2]
Ekwa J. Duala-Ekoko [Section 3]

Instructions:

- **DO NOT TURN THIS PAGE UNTIL INSTRUCTED**
- This is a **closed book** final examination; notes, slides, textbooks, and other forms of documentation are **not** allowed.
- **Non-programmable calculators** are allowed (though you should not need one).
- **Computers, PDAs, cell phones, and other electronic devices** are **not** allowed.
- Answer **all** questions **on this examination paper** and return it. If you need additional space, use pages 21-22 or the booklets supplied and clearly indicate where each question is continued. **In order to receive full marks for a question, you must show all work.**
- This final examination has **24** pages including this cover page, and is printed on both sides of the paper. Pages 23-24 contain information about useful classes and methods.

1	2	3	Subtotal
/10	/5	/5	/20

4	5	6	Subtotal
/10	/10	/10	/30

7	8	9	Subtotal
/15	/20	/15	/50

Total
/100

Section 1 - Short Questions

[10] 1. In one or two sentences, explain the differences between the concepts in each of the following pairs. **BE BRIEF**; overly long answers will be grounds for mark deductions.

(a) An `if` statement and a `while` statement

(b) Classes and objects

(c) Instance variables and class (or static) variables

(d) An array and an `ArrayList`

(e) Checked exceptions and unchecked exceptions

[5] 2. The most common method students encounter in this course is the `main()` method; its header can have slight variations but is usually the following:

```
public static void main(String[] args)
```

In one or two sentences, describe what each of the following elements of the header of the `main()` method means. **BE BRIEF**; overly long answers will be grounds for mark deductions.

(a) `public`

(b) `static` (in the context of methods)

(c) `void`

(d) `String[] args`

[5] 3. On December 31st, 2008, first-generation portable music players of a well-known brand seemingly refused to start. The problem was linked to the following code fragment, which is part of the start-up program of the music player:

```
year = ORIGINYEAR; /* = 1980 in the actual code, but assume 2008 for
                      the purposes of this question; the bug occurs
                      regardless of the actual value */
while (days > 365) {
  if (isLeapYear(year)) {
    if (days > 366) {
      days = days - 366;
      year = year + 1;
    }
  } else {
    days = days - 365;
    year = year + 1;
  }
}
```

The above code attempts to calculate the year based on the the value of the `days` variable, which represents the total number of days since January 1st in the year given by the value of the `ORIGINYEAR` variable.

Describe what the problem is. Assume that the `isLeapYear()` method works properly, and that variables `days`, `year`, and `ORIGINYEAR` are of type `int`.

Hint: Since the music player's release, this bug only manifested itself on December 31st, 2008. On that date, how many days have passed since January 1st of `ORIGINYEAR`?

Section 2 - Long Questions

[10] 4. What is displayed when the `main()` method of class `Trickery` is executed? Clearly indicate the program output in the appropriate space.

You may use the other blank space as a scratch pad to track the state of memory as the program executes, but the contents of this other space will not be graded.

The definition of the `Trickery` class:

```
public class Trickery {
  public static int v = 0;

  public static void trick(int v, Tuple t1, Tuple t2, Tuple[] a1,
      Tuple[] a2) {

    v = 100;

    t1.setFirst(20);
    t2 = new Tuple(40, 50);

    a1[0] = t2;
    a2 = new Tuple[1];
    a2[0] = new Tuple(80, 90);
  }

  public static void main(String[] args) {
    int value;
    Tuple firstTuple, secondTuple;
    Tuple[] firstArray, secondArray;

    value = 1;
    firstTuple = new Tuple(2, 3);
    secondTuple = new Tuple(4, 5);
    firstArray = new Tuple[1];
    firstArray[0] = new Tuple(6, 7);
    secondArray = new Tuple[1];
    secondArray[0] = new Tuple(8, 9);

    trick(value, firstTuple, secondTuple, firstArray, secondArray);

    System.out.println("v == " + v);
    System.out.println("firstTuple == " + firstTuple);
    System.out.println("secondTuple == " + secondTuple);
    System.out.println("firstArray == [" + firstArray[0] + "]");
    System.out.println("secondArray == [" + secondArray[0] + "]");
  }
}
```

The definition of the `Tuple` class:

```
public class Tuple {
  private int first;
  private int second;
```

```
  public Tuple(int first, int second) {
    this.first = first;
    this.second = second;
  }

  public void setFirst(int first) {
    this.first = first;
  }

  public void setSecond(int second) {
    this.second = second;
  }

  public String toString() {
    return "{" + this.first + ", " + this.second + "}";
  }
}
```

USE THE SPACE BELOW AS A SCRATCH PAD TO TRACK THE STATE OF THE VARIABLES IN MEMORY (THE CONTENTS OF THIS SPACE **WILL NOT** BE GRADED):

CLEARLY INDICATE THE PROGRAM'S OUTPUT IN THE SPACE BELOW (THE CONTENTS OF THIS SPACE **WILL** BE GRADED):

[10] 5. Consider the following class:

```
public class ExceptionStory {
  public static void tellStory(int[] a, int first, int second) {
    try {
      System.out.println("There was a method that did not crash when " +
        "trying to divide.");
      System.out.println(a[first] + " / " + a[second] + " == " +
        (a[first] / a[second]));
    } catch (ArithmeticException exception) {
      System.out.println("A problem? No problem!");
    } finally {
      System.out.println("This is not over!");
    }
    System.out.println("Told ya!");
  }

  public static void main(String[] args) {
    int[] array = {4, 2, 0};

    System.out.println("Once upon a time...");
    try {
      // Call to tellStory() method
    } catch (ArrayIndexOutOfBoundsException exception) {
      System.out.println("But there was a buggy method call.");
    }
    System.out.println("The end.");
  }
}
```

Suppose the comment

```
// Call to tellStory() method
```

in the `main()` method is replaced an actual statement calling the `tellStory()` method.

(a) What will be displayed if the comment is replaced by the following call:

```
tellStory(array, 0, 1);
```

and the `main()` method is executed?

(b) What will be displayed if the comment is replaced by the following call:

```
tellStory(array, 0, 2);
```

and the `main()` method is executed?

(c) What will be displayed if the comment is replaced by the following call:

```
tellStory(array, 0, 3);
```

and the `main()` method is executed?

(d) What will be displayed if the comment is replaced by the following call:

```
tellStory(null, 0, 1);
```

and the `main()` method is executed?

[10] 6. A *hailstone sequence* is a particular sequence of integers. The initial value of a hailstone sequence can be any strictly positive (that is, greater than 0) integer n, and the i^{th} term of the hailstone whose initial value is n is denoted as $x_{n,i}$.

Each term in the sequence can be calculated in the following manner:

- $x_{n,0} = n$ (that is, the initial term of the sequence is equal to n)
- If the current term $x_{n,i}$ is odd, then the next term $x_{n,i+1} = 3x_{n,i} + 1$ (three times the current term, plus 1)
- If the current term $x_{n,i}$ is even, then the next term $x_{n,i+1} = x_{n,i}/2$ (half the current term)

For example, here are the first few terms of the hailstone sequence whose initial value is 24:

- $x_{24,0} = 24$ (initial)
- $x_{24,1} = 12$ (24 is even, so $24/2 = 12$)
- $x_{24,2} = 6$ (12 is even, so $12/2 = 6$)
- $x_{24,3} = 3$ (6 is even, so $6/2 = 3$)
- $x_{24,4} = 10$ (3 is odd, so $3 \times 3 + 1 = 10$)
- $x_{24,5} = 5$ (10 is even, so $10/2 = 5$)
- $x_{24,6} = 16$ (5 is odd, so $3 \times 5 + 1 = 16$)

Write a method with the following header:

```
public static int hailstone(int n, int i)
```

This method computes the `i`$^{\text{th}}$ term of the hailstone sequence whose initial value is `n`. That is, if the value of `n` is `24`, and the value of `i` is `4`, then this method should return $x_{24,4}$, or `10`. You may assume that the value of `n` is a strictly postive integer, and that the value of `i` is a non-negative integer (that is, greater than or equal to `0`).

In order to get full marks for this question, your method **MUST** be **RECURSIVE**. However, a correct iterative solution will still be worth 6 marks.

WRITE YOUR `hailstone()` METHOD IN THE SPACE BELOW:

YOUR `hailstone()` METHOD CONTINUED:

Section 3 - Programming Questions

[15] 7. The `String` class is one of the most useful classes among those provided by the Java Standard Class Library, but how does it actually work?

Write a class called `MiniString`; like `String` objects, objects that belong to the `MiniString` class represent ordered sequences of characters. Also like `String` objects, valid character indices in a `MiniString` range from 0 (inclusive) to the number of characters in the `MiniString` (exclusive); that is, the i^{th} character of a `MiniString` has index $i - 1$.

`MiniString` objects **MUST** keep track of the character sequences they represent using a **PLAIN ARRAY** (of `char`); you **MUST NOT** use regular `String`s within the `MiniString` class under **ANY** circumstances. You **MUST NOT** use `ArrayList`s (or any other class which is part of the Java Collection Framework such as `LinkedList`) within the `MiniString` class either.

Your `MiniString` should provide the following `public` methods:

- A constructor that takes no parameters, and initializes the newly-created `MiniString` so that it represents an empty character sequence, that is, one that contains no characters.
- A second constructor, which takes as a parameter an array of `char`, and initializes the newly-created `MiniString` so that it represents the sequence of characters currently contained in the array passed as parameter. The contents of the character array are copied, so that any subsequent modification of the character array does not affect the newly-created `MiniString`
- A method called `length()`, which takes no parameters and returns the a value of type `int` representing the length of this `MiniString`, that is, the number of characters it contains.
- A method called `charAt()`, which takes as a parameter a value of type `int`, and returns a value of type `char` representing the character in the `MiniString` at the position given by the parameter. Remember that `MiniString` indexing works exactly like regular `String` indexing.
  If the value of the parameter does not represent a valid index in the `MiniString` (it is either negative, or greater than or equal to the length of `MiniString`), then the `charAt()` method **MUST** throw a `StringIndexOutOfBoundsException`.
- A method called `concat()`, which takes as a parameter a `MiniString`, and returns a new `MiniString` repesenting the concatenation of the `MiniString` this method is called on and the `MiniString` parameter, in that order. For example, if the `MiniString` this method is called on represents the character sequence `"COMP"`, and the parameter `MiniString` represents the character sequence `"-202"`, then the method should return a new `MiniString` representing the character sequence `"COMP-202"`.
  If the `MiniString` passed as parameter is `null`, then the method **MUST** return a reference to the `MiniString` on which this method is called.
- A method called `equals()`, which takes as a parameter a `MiniString`, and returns a value of type `boolean`. This method returns `true` if the `MiniString` this method is called on represents the same character sequence as the `MiniString` parameter (that is, both contain the same number of characters in the same order), `false` otherwise. If the `MiniString` passed as parameter is `null`, then the method **MUST** return `false`.

Note that `MiniString` objects are immutable, meaning that the state of a `MiniString` object cannot change once the object is created. This implies that, other than the constructors, the methods of the `MiniString` class **MUST NOT** change the states of the `MiniString` objects they are called on, nor the states of any `MiniString` objects they take as parameters.

WRITE YOUR `MiniString` CLASS IN THE SPACE BELOW:

[20] 8. **Part 1**:

Write a class called `Movie`; each object that belongs to this class contains information about a movie, such as its title, the year during which it was released, and the people who are featured in it.

Your `Movie` class should provide the following `public` methods:

- A constructor, which takes as a parameter a `String` object and a value of type `int`, in this order; the `String` object represents the title of the newly-created `Movie` object, and the value of type `int` represents the year during which the movie represented by the newly-created `Movie` object was released.
- An accessor method called `getTitle()`, which takes no parameter and returns a `String` object representing the title of the movie this `Movie` object represents.
- An accessor method called `getYear()`, which takes no parameter and returns a value of type `int` representing the year during which the movie represented by this `Movie` object was released.
- A method called `addToCast()`, which takes as parameter a `String` object, and returns nothing. This method should add the person whose name is given by the `String` parameter to the list of people featured in the movie represented by the `Movie` object. If the person whose name is given by the `String` parameter is already in the list of people featured in the movie represented by the `Movie` object, then this method should not modify the list; name comparisons are done in a case-**SENSITIVE** manner.
- A method called `features()`, which takes as a parameter a `String` object, and returns a value of type `boolean`. This method returns `true` if the movie represented by the `Movie` object features the person whose name is given by the `String` parameter, `false` otherwise; again, name comparisons are done in a case-**SENSITIVE** manner. If the `String` object passed as parameter is `null`, then this method **MUST** return `false`.

Hint: Use an `ArrayList` (of `String`) to keep track of the names of the people featured in the movie represented by each `Movie` object. Is there a method defined in the `ArrayList` class which determines whether an element is already stored in an `ArrayList`?

WRITE YOUR `Movie` CLASS IN THE SPACE BELOW:

YOUR `Movie` CLASS CONTINUED:

Part 2:

Write a method with the following header:

```
public static void writeMovieList(ArrayList<Movie> allMovies,
  String person, String fileName)
```

This method first attempts to open the file whose name is given by the `String` parameter called `fileName` for writing. If the attempt to open the file is successful, the method then processes the `Movie` objects stored in the `ArrayList` parameter called `allMovies`. For each of these `Movie` objects, if the object represents a movie which features the person whose name is given by the `String` parameter `person`, then information about this movie is written to the file; otherwise, no information about this movie is written to the file. Once it has processed all the `Movie` objects in `allMovies`, the method closes the file and terminates.

The format of the files generated by this method is as follows:

- The first line contains `"List of movies featuring: "`, followed by the person's name.
- For each `Movie` object in the `ArrayList` parameter, the method checks whether the movie it represents features the person whose name is given by `person`. If it does, then the method prints information about the `Movie` object in the following format: the `String "- "` is written first, followed by the title of the movie, the `String " ("`, the year during which the movie was released, and finally, `")"`. Information about each movie is written on a separate line.
- After all `Movie` objects in the `ArrayList` parameter have been processed as described above, the `String "Number of movies: "` is written to the file, followed by the actual number of `Movie` objects in the list representing movies which feature the person whose name is given by `person`. This information is written printed on the last line in the file.

Note that the order in which the movies are written to the file does not matter.

For example, suppose `allMovies` contains 5 `Movie` objects, that parameter `person` is a reference to the `String "Gerard Depardieu"`, and that 2 of the 5 `Movie` objects stored in `allMovies` represent movies featuring Gerard Depardieu: *Cyrano de Bergerac*, released in 1990, and *Asterix & Obelix vs Caesar*, released in 1999. In this case, the file generated by the `writeMovieList()` method would look like this:

```
List of movies featuring: Gerard Depardieu
- Cyrano de Bergerac (1990)
- Asterix & Obelix vs Caesar (1999)
Number of movies: 2
```

Remember that when opening and writing to the file, `IOException`s could be thrown. In such circumstances, your `writeMovieList()` method must propagate these exceptions to the method that called `writeMovieList()` (this might involve making a slight change to the method header).

WRITE YOUR `writeMovieList()` METHOD IN THE SPACE BELOW:

[15] 9. The `grep` program is "*a command-line text search utility originally written for Unix*"[1]. It takes the following arguments from the command-line: the name of a file, as well as a pattern. It then searches the specified file for all the lines which contain a substring that matches the specified pattern, and displays these lines to the screen.

Write a class called `MiniGrep`, which defines a single method, `main()`. This method does the following:

- The method first checks the number of command-line arguments. If there are not *exactly* 2 command-line arguments, the method should display the following error message to the standard **error** stream and terminate:

  ```
  Usage:  java MiniGrep <file> <pattern>
  ```

- The method then attempts to open the file whose name is given by the first command-line argument for reading. If an error occurs while attempting to open the file (for example, it does not exist), the method should display the following error message to the standard **error** stream and terminate:

  ```
  Could not read from file <file>
  ```

  In the above message, `<file>` should be replaced by the actual name of the file that could not be read.

- If the method is successful in its attempt to open the file, then it should read each line one by one, and check whether the line contains the second command-line argument as a substring. In this case, the method should display the number of this line within the file, followed by the line itself, to the screen. This process is repeated for each line in the file. Substring matching is to be done in a case-**SENSITIVE** manner.

  For example, if the second command-line argument is the `String` `"lo"`, and line 1 in the specified file is `"Hello world"`, then because `"lo"` is a substring found in `"Hello world"`, the program should display line 1 to the screen in the following format:

  ```
  1:  Hello world
  ```

  However, if line 2 in the specified file is `"How are you?"`, then because `"lo"` is not a substring found in `"How are you?"`, the program should not display line 2 to the screen. If none of the lines in the file contain the second command-line argument as a substring, then the method should not display anything.

- Once the method has read all the lines in the file and displayed all those which contained the second command-line argument as a substring, it should close the file and terminate.

If an error occurs at any point while reading from the input file, your program should display an appropriate error message to the standard **error** stream, attempt to close the files, and terminate.

Sample session: Suppose the file called `test.txt` contains the following information:

```
Then there
were
others
```

In this case, if the user types

```
java MiniGrep test.txt the
```

[1] `http://en.wikipedia.org/wiki/Grep`

on the command prompt, the program will display the following:

```
1: Then there
3: others
```

Hint: Is there a method defined in the `String` class which determines whether a `String` contains another `String` as a substring?

WRITE YOUR `MiniGrep` CLASS IN THE SPACE BELOW:

YOUR `MiniGrep` CLASS CONTINUED:

Total marks for Section 3: **50**

Total marks: **100**

USE THIS PAGE IF YOU NEED ADDITIONAL SPACE. CLEARLY INDICATE WHICH QUESTION(S) YOU ARE ANSWERING HERE.

USE THIS PAGE IF YOU NEED EVEN MORE ADDITIONAL SPACE. CLEARLY INDICATE WHICH QUESTION(S) YOU ARE ANSWERING HERE.

SUMMARY OF JAVA STANDARD LIBRARY METHODS FOR SELECTED CLASSES

- `String` (package `java.lang`) Methods:
  - `public int length()`: Returns the length of this `String`.
  - `public char charAt(int index)`: Returns the `char` value at the specified `index`.
  - `public boolean equals(Object anObject)`: Compares this `String` to `anObject`
  - `public boolean equalsIgnoreCase(String anotherString)`: Compares, ignoring case considerations, this `String` to `anotherString`.
  - `public int compareTo(String anotherString)`: Compares this `String` to `anotherString` lexicographically.
  - `public boolean startsWith(String prefix)`: Tests if this `String` starts with the specified `prefix`.
  - `public boolean endsWith(String suffix)`: Tests if this `String` ends with the specified `suffix`.
  - `public int indexOf(int ch)`: Returns the index within this `String` of the first occurrence of character `ch`, `-1` if it does not occur.
  - `public int indexOf(int ch, int fromIndex)`: Returns the index within this `String` of the first occurrence of character `ch`, starting the search at position `fromIndex`; returns `-1` if `ch` does not occur in this `String`.
  - `public int indexOf(String str)`: Returns the index within this `String` of the first occurrence of substring `str`, `-1` if it does not occur.
  - `public int indexOf(String str, int fromIndex)`: Returns the index within this `String` of the first occurrence of substring `str`, starting at position `fromIndex`; returns `-1` if `str` does not occur in this `String`.
  - `public String substring(int beginIndex)`: Returns a new `String` which is a substring of this `String`, composed of the characters starting at position `beginIndex` (inclusive).
  - `public String substring(int beginIndex, int endIndex)`: Returns a new `String` that is a substring of this `String`, composed of the characters starting at position `beginIndex` (inclusive), and ending at position `endIndex` (exclusive).
  - `public String replace(char oldChar, char newChar)`: Returns a new `String` resulting from replacing all occurrences of `oldChar` in this `String` with `newChar`.
  - `public String toLowerCase()`: Returns a new `String` consisting of all the characters in this `String` converted to lower case.
  - `public String toUpperCase()`: Returns a new `String` consisting of all the characters in this `String` converted to upper case.
  - `public String trim()`: Returns a copy of this `String`, with leading and trailing whitespace omitted.
- `Scanner` (package `java.util`) Methods:
  - `public Scanner(File source) throws java.io.FileNotFoundException`: Constructs a new `Scanner` that produces values scanned from the specified file.
  - `public Scanner(InputStream source)`: Constructs a new `Scanner` that produces values scanned from the specified input stream.
  - `public Scanner(String source)`: Constructs a new `Scanner` that produces values scanned from the specified string.
  - `public void close()`: Closes this `Scanner`
  - `public boolean hasNext()`: Returns `true` if this `Scanner` has another token in its input.
  - `public boolean hasNextDouble()`: Returns `true` if the next token in this `Scanner`'s input can be interpreted as a `double` value using the `nextDouble()` method.
  - `public boolean hasNextInt()`: Returns `true` if the next token in this `Scanner`'s input can be interpreted as an `int` value using the `nextInt()` method.
  - `public boolean hasNextLine()`: Returns `true` if there is another line in the input of this `Scanner`
  - `public boolean hasNextLong()`: Returns `true` if the next token in this `Scanner`'s input can be interpreted as a `long` value using the `nextLong()` method.
  - `public String next()`: Finds and returns the next complete token from this `Scanner`.
  - `public double nextDouble()`: Scans the next token of the input as a `double`.
  - `public int nextInt()`: Scans the next token of the input as an `int`.
  - `public String nextLine()`: Advances this `Scanner` past the current line and returns the input read.
  - `public long nextLong()`: Scans the next token of the input as a `long`.

- `ArrayList<E>` (package `java.util`) Methods:
  - `public int size()`: Returns the number of elements in this list.
  - `public boolean isEmpty()`: Returns `true` if this list contains no elements.
  - `public boolean contains(Object o)`: Returns `true` if this list contains element `o`.
  - `public int indexOf(Object o)`: Returns the index of the first occurrence of element `o` in this list, or -1 if this list does not contain this element.
  - `public Object[] toArray()`: Returns an array containing all of the elements in this list in proper sequence (from first to last element).
  - `public E get(int index)`: Returns the element at position `index` in this list.
  - `public E set(int index, E element)`: Replaces the element at the position `index` in this list with the specified `element`.
  - `public boolean add(E e)`: Appends the specified `element` to the end of this list.
  - `public void add(int index, E element)`: Inserts the specified element at the position `index` in this list.
  - `public E remove(int index)`: Removes the element at position `index` in this list.
  - `public boolean remove(Object o)`: Removes the first occurrence of the specified element `o` from this list, if it is present.
  - `public void clear()`: Removes all of the elements from this list.
- `PrintStream` (package `java.io`) Methods:
  - `public PrintStream(File file) throws java.io.FileNotFoundException`: Creates a new `PrintStream` with the specified `file`.
  - `public PrintStream(String fileName) throws java.io.FileNotFoundException`: Initializes a new `PrintStream`, with the specified `fileName`.
  - `public print(boolean b)`: Prints `boolean` value `b`.
  - `public print(char c)`: Prints `char` value `c`.
  - `public print(double d)`: Prints `double` value `d`.
  - `public print(int i)`: Prints `int` value `i`.
  - `public print(Object o)`: Prints `Object` `o`.
  - `public print(String s)`: Prints `String` `s`.
  - `public println()`: Terminates the current line by writing the line separator string.
  - `public println(boolean b)`: Prints `boolean` value `b` and then terminates the line.
  - `public println(char c)`: Prints `char` value `c` and then terminates the line.
  - `public println(double d)`: Prints `double` value `d` and then terminates the line.
  - `public println(int i)`: Prints `int` value `i` and then terminates the line.
  - `public println(Object o)`: Prints `Object` `o` and then terminates the line.
  - `public println(String s)`: Prints `String` `s` and then terminates the line.
- `File` (package `java.io`) Methods:
  - `public File(String pathname)`: Creates a `File` representing the file at the given `pathname`.