First Name: ______________________________ **Last Name**: ______________________________

McGill ID: ______________________________ **Section**: ______

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

Wednesday, December 16, 2009
9:00–12:00

Examiners: Mathieu Petitpas [Section 1 and 3]
Kamal Zellag [Section 2]

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 22-23 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 **26** pages including this cover page, and is printed on both sides of the paper. Pages 24-26 contain information about useful classes and methods.

1	**2**	**3**	**Subtotal**
/5	**/5**	**/5**	**/15**

4	**5**	**Subtotal**
/10	**/15**	**/25**

6	**7**	**8**	**Subtotal**
/30	**/15**	**/15**	**/60**

Total
/100

Section 1 - Short Questions

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

(a) Loop

(b) Method

(c) Reference variable

(d) Class (in the context of object-oriented programming)

(e) Constructor

[5] 2. In one or two sentences, explain the meaning of each of the following Java reserved words. **BE BRIEF**; overly long answers will be grounds for mark deductions.

(a) `final`

(b) `public`

(c) `void`

(d) `return`

(e) `new`

[5] 3. Consider the following program.

```
public class MyInteger {
  private int x;

  public MyInteger(int myX) {
    x = myX;
  }

  public double add(int y, int z) {
    System.out.println("add(int, int): " + x + " + " + y + " + " + z);
    return x + y + z;
  }

  public double add(int y, double z) {
    System.out.println("add(int, double): " + x + " + " + y + " + " + z);
    return x + y + z;
  }

  public double add(double y, double z) {
    System.out.println("add(double, double): " + x + " + " + y +
      " + " + z);
    return x + y + z;
  }

  public static void main(String[] args) {
    MyInteger n1 = new MyInteger(1);
    MyInteger n2 = new MyInteger(2);
    double r1, r2, r3, r4;

    r1 = n1.add(3, 4);
    r2 = n2.add(3, 4);
    r3 = n1.add(5, 6.0);
    r4 = n2.add(7.0, 8.0);

    System.out.println("n1.add(3, 4) == " + r1);
    System.out.println("n2.add(3, 4) == " + r2);
    System.out.println("n3.add(5, 6.0) == " + r3);
    System.out.println("n4.add(7.0, 8.0) == " + r4);
  }
}
```

The program compiles without error and terminates normally when it is executed; what does it display?

WRITE THE PROGRAM'S OUTPUT IN THE SPACE BELOW:

Total marks for Section 1: **15**

Section 2 - Long Questions

[10] 4. Consider the following class:

```
public class Set {
  private int[] elements;

  public Set(int[] myElements) {
    /* Point 1 */
    elements = myElements;
  }

  public String toString() {
    String result;

    result = "[";
    for (int i = 0; i < elements.length; i++) {
      result = result + elements[i];
      /* Point 2 */
      if (i != (elements.length - 1)) {
        result = result + ", ";
      }
    }

    /* Point 3 */
    return result + "]";
  }

  public static void main(String[] args) {
    int[] a = {1, 2, 3};
    Set s;

    /* Point 4 */
    s = new Set(a);
    System.out.println(s);
  }
}
```

Which variables are in scope at each of the points marked by comments in the above class? That is, at each of these points, which variables can be used in a statement without the compiler reporting an unknown variable error?

List each variable that is in scope at each of the points marked by comments in the above class, and indicate whether it is an instance variable, a formal parameter, or a local variable. Note that you will be penalized for every variable that is out of scope at a given point but that you list as being in scope at that same point.

(a) Which variables are in scope at the point given by the comment `/* Point 1 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

(b) Which variables are in scope at the point given by the comment `/* Point 2 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

(c) Which variables are in scope at the point given by the comment `/* Point 3 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

(d) Which variables are in scope at the point given by the comment `/* Point 4 */`? For each variable you list, indicate whether it is an instance variable, a formal parameter, or a local variable.

[15] 5. What is displayed when the `main()` method of class `SwapHeadache` is executed? Clearly indicate the program output in the appropriate space on page 13.

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 `Pair` class:

```
public class Pair {
  private int left, right;

  public Pair(int l, int r) {
    left = l;
    right = r;
  }

  public void swap() {
    int temp = left;
    left = right;
    right = temp;
  }

  public String toString() {
    return "[L: " + left + "; R: " + right + "]";
  }
}
```

The definition of the `SwapHeadache` class:

```
public class SwapHeadache {
  public static void swapInts(int v1, int v2) {
    int temp = v1;
    v1 = v2;
    v2 = temp;
  }

  public static void swapIntArray(int[] a) {
    int temp = a[0];
    a[0] = a[1];
    a[1] = temp;
  }

  public static void swapIntMatrix(int[][] m) {
    int[] temp = m[0];
    m[0] = m[1];
    m[1] = temp;
  }

  public static void swapSinglePair(Pair p) {
    p.swap();
  }

  public static void swapMultiplePairs(Pair p1, Pair p2) {
    Pair temp = p1;
```

```
      p1 = p2;
      p2 = temp;
    }

    public static void swapPairArrays(Pair[] pA1, Pair[] pA2) {
      Pair temp = pA1[0];
      pA1[0] = pA2[0];
      pA2[0] = temp;
    }

    public static void main(String[] args) {
      int myV1 = 1;
      int myV2 = 2;
      int[] myA = {3, 4};
      int[][] myM = {{5, 6}, {7, 8}};
      Pair myP1 = new Pair(9, 10);
      Pair myP2 = new Pair(11, 12);
      Pair myP3 = new Pair(13, 14);
      Pair[] myPA1 = {new Pair(15, 16), new Pair(17, 18)};
      Pair[] myPA2 = {new Pair(19, 20), new Pair(21, 22)};

      swapInts(myV1, myV2);
      swapIntArray(myA);
      swapIntMatrix(myM);
      swapSinglePair(myP1);
      swapMultiplePairs(myP2, myP3);
      swapPairArrays(myPA1, myPA2);

      System.out.println(myV1 + ", " + myV2);
      System.out.println("{" + myA[0] + ", " + myA[1] + "}");

      System.out.println("{{" + myM[0][0] + ", " + myM[0][1] + "}, {" +
        myM[1][0] + ", " + myM[1][1] + "}}");

      System.out.println(myP1 + ", " + myP2 + ", " + myP3);

      System.out.println("{" + myPA1[0] + ", " + myPA1[1] + "}");
      System.out.println("{" + myPA2[0] + ", " + myPA2[1] + "}");
    }
  }
```

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):

Total marks for Section 2: $\overline{\textbf{25}}$

Section 3 - Programming Questions

6. In a well-known video game franchise, the main character is an anthropomorphic (human-shaped) robot, who must fight similar robots bent on conquering the world. Each robot is armed with a weapon, and may be vulnerable to the weapons of other robots. Note that not every weapon affects every robot in the same way; a robot might be immune to certain weapons while another robot might be immune to different weapons, and a weapon which causes minor damage to one robot might cause major damage to another robot.

   In this question, you will write a program that simulates a fight between two such robots.

[20] **Part 1**:

Write a class called `Robot` that represents a robot that can fight other robots. Each robot has the following attributes:

- A name (a `String`)
- A life force rating (an `int`), which represents the endurance of the robot; the robot stops to function if the value of this attribute decreaes to `0`
- An accuracy rating (a `double` between `0.0` and `1.0`, inclusive), which represents the probability that the robot will hit its target when firing its weapon
- A weapon (a `String`)
- A set of weak points, which specifies the weapons to which the robot is vulnerable. Each of these weak points is represented by a `Weakness` object, and each `Weakness` object has two attributes: a weapon attribute, which specifies a weapon to which the robot is vulnerable, and a damage attribute, which specifies the amount of damage the robot sustains when it takes a hit from this weapon.
  `Robot` objects **MUST** keep track of their weak points using a **PLAIN ARRAY** (of `Weakness` objects); you **MUST NOT** use `ArrayLists` (or any other class which is part of the Java Collection Framework such as `LinkedList`) within the `Robot` class under **ANY** circumstances.

Note that the `Weakness` class has already been implemented; **complete details for this class**, such as the methods it defines, what these methods do, what parameters they take, what values they return, and so on, **are specified on the last page of this examination.**

The `Robot` class defines the following **INSTANCE** methods:

- A constructor, which takes as parameters a `String`, an `int`, a `double`, another `String`, and an array of `Weaknesses`, in this order. These parameters represent the name, life force rating, accuracy rating, weapon, and weaknesses of the newly-created `Robot`, respectively. The contents of the array of `Weaknesses` are copied, so that any subsequent modification of this array does not affect the state of the newly-created `Robot`, and vice-versa. Your constructor **MAY** assume that the value of the accuracy rating parameter is in the proper range, and that the `String` parameters and array parameter are **NOT** `null`.
- A method called `getName()`, which takes no parameters and returns a `String` representing the name of this `Robot`.
- A method called `isFunctional()`, which takes no parameters and returns a `boolean`. This method returns `true` if this `Robot` is functional, `false` otherwise; a `Robot` is functional if and only if the value of its life force rating is greater than `0`.

- A method called `takeDamage()`, which takes as its only parameter a `String` representing a weapon, and returns a `boolean`. This method looks in this `Robot`'s set of weak points for a `Weakness` object whose weapon attribute matches the parameter `String`; the comparison is to be done in a case-**IN**sensitive manner. If no such `Weakness` object is found, the method returns `false`, and the `Robot`'s state does not change.
  On the other hand, if a `Weakness` object whose weapon attribute matches the parameter `String` is found in this `Robot`'s set of weak points, the `Robot`'s life force rating is diminished by that `Weakness` object's damage attribute, and the method returns `true`.
  Note that a `Robot`'s life force rating can never be less than `0`; if the `Robot` is hit by a blast from a weapon against which it is weak, and this blast causes more damage than the `Robot`'s current life force rating, then the latter should be reduced to `0`.
- A method called `attack()`, which takes as its only parameter a `Robot` object and returns a `boolean`. When the `attack()` method is invoked on a `Robot` object, the robot represented by this object attempts to use its weapon to attack the robot represented by the `Robot` object the `attack()` method receives as parameter. The `attack()` method will return `true` if the attack succeeds, `false` otherwise.
  To determine if the attack succeeds, call the `random()` method of the `Math` class. This method takes no parameters, and returns a value of type `double` between `0.0` (inclusive) and `1.0` (exclusive). If the value returned by the `random()` method is less than the accuracy rating of the attacking robot, then the attack hits. In this case, the robot being attacked takes damage from the attacking robot's weapon as described in the specification of the `takeDamage()` method. The attack is successful if it hits **AND** the robot being attacked is damaged as a result; otherwise, the attack fails.
- A method called `toString()`, which takes no parameters and returns a `String` object. This `String` object is a textual representation of the `Robot` object, and consists of the following: the `Robot`'s name, followed by the `String` `" ("`, followed by the `Robot`'s life force rating, followed by the `String` `" HP)"`.

Note that you must respect proper encapsulation practices; that is, the attributes and methods of your class **MUST** be declared using the proper visibility modifiers.

WRITE YOUR `Robot` CLASS IN THE SPACE BELOW:

YOUR `Robot` CLASS CONTINUED:

[10] **Part 2**:

Write a `public` and `static` method called `fight()`, which takes as parameters two `Robot` objects and returns a `Robot` object. This method does the following:

- Repeat the following steps as long as both `Robot`s are still functional:
  - Have the first `Robot` attempt to attack the second `Robot`
  - Have the second `Robot` attempt to attack the first `Robot`

  One execution of the above set of steps is called a *round*.
- Once at least one of the two `Robot`s is no longer functional, return the `Robot` which is still functional. If both `Robot`s caused each other to stop functioning during the same round, the method **MUST** return `null`.

You **MUST** call the methods you were asked to write in the previous part to determine whether a `Robot` is still functional, and to have a `Robot` attack its opponent. You **MAY** assume that the `Robot` class has been implemented correctly, even if you did not successfully complete the previous part.

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

[15] 7. The `diff` program is a Unix command-line utility which compares text files line by line. It takes as arguments the names of two files to be compared. If the two files have the same content, then `diff` produces no output. On the other hand, if the two files are different, it will display to the screen the lines where the files differ.

In this question, you will write a method which has the same basic functionality as the `diff` program. The header of this method is as follows:

```
public static void diff(String left, String right) throws
  IOException
```

This method does the following:

- Open the two files whose names are given by the `String` parameters called `left` (the left-hand file) and `right` (the right-hand file) for reading.
- Repeat the following steps as long as there are lines to be read from at least one of the files:
  - If there are lines to be read from both files, the method reads one line from each file and compares these lines. If the lines are **NOT** equal, then the method displays the lines to the screen in the following format:
    ```
    < leftLine
    > rightLine
    ```
    where `leftLine` is the actual line read from the left-hand file and `rightLine` is the actual line read from the right-hand file. If the lines read from the files are equal, the method does not display them. The lines are compared in a case-**SENSITIVE** manner.
  - If there are lines to be read from the left-hand file, but not from the right-hand file, then the method reads one line from the left-hand file, and displays it to the screen in the following format:
    ```
    < leftLine
    ```
    where `leftLine` is the actual line the method read from the left-hand file.
  - Likewise, if there are lines to be read from the right-hand file, but not from the left-hand file, then the method reads one line from the right-hand file, and displays it to the screen in the following format:
    ```
    > rightLine
    ```
    where `rightLine` is the actual line the method read from the right-hand file.
  - Note that whenever a line is displayed, only one space separates the `">"` or `"<"` character from the line being displayed.
- Close both files once there are no more lines to be read from either of them.

Because the `diff()` method is specified to throw an exception object of type `IOException`, you **MAY** assume that no I/O-related problems will occur while reading from the files. In other words, your method **DOES NOT** have to check that no such problems occur, and **DOES NOT** have to manage `IOExceptions` using `try-catch` blocks. Your method **MAY** also assume that the `String` parameters it receives are **NOT** equal to `null`.

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

[15] 8. Write a method with the following header:

```
public static void eliminateComments(ArrayList<String>
  lines, String path) throws IOException
```

The `eliminateComments()` method does the following:

- Open the file whose name is given by the `String` parameter called `path` (the output file) for writing.
- Process each `String` stored in the `ArrayList` parameter called `lines` as follows:
  - If the `String` does not contain the substring `"//"`, the method writes it to the output file without modification.
    For example, the `String`
    `"a = b;";`
    does not contain the substring `"//"`, and therefore would be written to the output file without modification.
  - If the `String` contains the substring `"//"`, the first occurrence of this substring and all characters occurring to its right within the `String` are discarded; only the part of the `String` to the left of the first occurrence of the substring is written to the output file.
    For example, the `String`
    `"i = i + 1; // Increment i"`
    contains the substring `"//"`; therefore, only
    `"i = i + 1; "`
    (that is, the characters to the left of the first occurrence of `"//"` within this `String`) would be written to the output file.

  The `Strings` stored in the `ArrayList` are processed in the same order as the one in which they occur in the `ArrayList`; that is, the `String` stored at position `0` in the `ArrayList` is processed first, the `String` at position `1` is processed second, and so on.
- Close the output file once all the `Strings` stored in the `ArrayList` have been processed.

Because the `eliminateComments()` method is specified to throw an exception object of type `IOException`, you **MAY** assume that no I/O-related problems will occur while writing to the output files. In other words, your method **DOES NOT** have to check that no such problems occur, and **DOES NOT** have to manage `IOExceptions` using `try-catch` blocks. Your method **MAY** also assume that the `String` parameters it receives are **NOT** equal to `null`.

Hint: Use the methods defined in the `String` class to analyze and extract the characters that a `String` contains. In particular, the `indexOf()` and `substring()` methods should prove to be immensely useful.

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

Total marks for Section 3: **60**

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 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 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.
- `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.

Note that the `PrintWriter` class defines the same methods and constructors (except for the fact that the constructors are called `PrintWriter` instead of `PrintStream`).

- `Math` (package `java.lang`) Methods:
  - `public static double pow(double a, double b)`: Returns the value of `a` raised to the power of `b`.
  - `public static double sqrt(double a)`: Returns the correctly rounded positive square root of `double` value `a`.
  - `public static double random()`: Returns a `double` value with a positive sign, greater than or equal to `0.0` and less than `1.0`.
  - `public static double sin(double a)`: Returns the trigonometric sine of angle `a`, where `a` is in radians.
  - `public static double cos(double a)`: Returns the trigonometric cosine of angle `a`, where `a` is in radians.
  - `public static double tan(double a)`: Returns the trigonometric tangent of angle `a`, where `a` is in radians.
  - `public static double toDegrees(double angrad)`: Converts angle `angrad` measured in radians to an approximately equivalent angle measured in degrees.
  - `public static double toRadians(double angdeg)`: Converts angle `angdeg` measured in degrees to an approximately equivalent angle measured in radians.
  - `public static double exp(double a)`: Returns Euler's number *e* raised to the power of `double` value `a`.
  - `public static double log(double a)`: Returns the natural logarithm (base *e*) of `double` value `a`.
  - `public static double log10(double a)`: Returns the base 10 logarithm of `double` value `a`.
- `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 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.
- `File` (package `java.io`) Methods:
  - `public File(String pathname)`: Creates a `File` representing the file at the given `pathname`.

DESCRIPTIONS OF CLASSES PROVIDED FOR QUESTION 6

- `Weakness`
  - `public Weakness(String weapon, int damage)`: Creates a new `Weakness` object whose weapon attribute is `weapon` and whose damage attribute is `damage`.
  - `public String getWeapon()`: Returns the weapon attribute of this `Weakness` object.
  - `public int getDamage()`: Returns the damage attribute of this `Weakness` object.