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

McGill ID: ______________________________ **Section**: ______

Faculty of Science
COMP-202A - Introduction to Computing I (Fall 2010) - All Sections
Final Examination

Wednesday, December 8, 2010
14:00–17:00

Examiners: Maja Frydrychowicz [Section 1]
Mathieu Petitpas [Sections 2 and 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. However, a letter-sized (8.5” by 11”) **crib sheet** is permitted. This crib sheet can be single or double-sided, it can also be handwritten or typed, but the use of magnifying glasses is prohibited. Translation dictionaries (for human languages only) are permitted, but instructors and invigilators reserve the right to inspect them at any time during the examination.
- **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 24-26 or the booklets supplied upon request and clearly indicate where each question is continued. **In order to receive full marks for a question, you must show all work** unless otherwise stated.
- This final examination has **30** pages including this cover page, and is printed on both sides of the paper. Pages 27-30 contain information about useful classes and methods.

1	2	3	Subtotal
/6	/4	/5	/15

4	5	Subtotal
/10	/15	/25

6	7	8	9	10	Subtotal
/10	/15	/10	/15	/10	/60

Total
/100

Section 1 - Short Questions

[6] 1. Consider the following class:

```
public class CrazyCall {
  private int integer;

  public CrazyCall(int i) {
    integer = i;
  }

  public void first() {
    System.out.println("Boo!");
  }

  public void second() {
    CrazyCall call = new CrazyCall(1);

    first(); /* Call A */
    this.first();
    CrazyCall.first(); /* Call B */
    call.first();
  }

  public static void third() {
    CrazyCall call = new CrazyCall(2);

    second();
    this.second(); /* Call C */
    CrazyCall.second();
    call.second();
  }

  public static void fourth() {
    third();
    this.third();
    CrazyCall.third(); /* Call D */
  }
}
```

For each of the method calls marked by comments in the above program, state whether or not the call will cause a compilation error.

- If the call causes a compilation error, explain why it does.
- If the call does not cause a compilation error, state whether or not a `CrazyCall` object is bound to the call (such an object is also referred to as the calling object or the target of the method call).
  - If **no** `CrazyCall` object is bound to the method call, explain why.
  - If a `CrazyCall` object **is** bound to the method call, state which object is bound.

(a) `first(); /* Call A */`

Check off **one** option and complete your answer in the space below. **Call A**. . .

□ causes a compilation error **because**...

□ does **not** cause a compilation error; **no** `CrazyCall` object is bound to the call **because**...

□ does **not** cause a compilation error; the following `CrazyCall` object is bound to the call:

(b) `CrazyCall.first(); /* Call B */`

Check off **one** option and complete your answer in the space below. **Call B**. . .

□ causes a compilation error **because**...

□ does **not** cause a compilation error; **no** `CrazyCall` object is bound to the call **because**...

□ does **not** cause a compilation error; the following `CrazyCall` object is bound to the call:

(c) `this.second(); /* Call C */`

Check off **one** option and complete your answer in the space below. **Call C**. . .

□ causes a compilation error **because**...

□ does **not** cause a compilation error; **no** `CrazyCall` object is bound to the call **because**...

□ does **not** cause a compilation error; the following `CrazyCall` object is bound to the call:

(d) `CrazyCall.third(); /* Call D */`

Check off **one** option and complete your answer in the space below. **Call D**. . .

□ causes a compilation error **because**...

□ does **not** cause a compilation error; **no** `CrazyCall` object is bound to the call **because**...

□ does **not** cause a compilation error; the following `CrazyCall` object is bound to the call:

[4] 2. Consider the following `SerialNumber` class:

```
public class SerialNumber {
  private int number;
  private static int nextNumber = 1;

  public void SerialNumber() {
    this.number = nextNumber;
    nextNumber = nextNumber + 1;
  }

  public int getNumber() {
    return this.number;
  }
}
```

Each object that belongs to the `SerialNumber` class represents a serial number; serial numbers are unique numbers assigned to various entities for identification purposes, and each serial number varies from its predecessor or successor by a constant value. Serial numbers have various uses, notably in quality control and counterfeit currency detection.

The above class attempts to ensure that the value of the `number` attribute of each `SerialNumber` object is different. That is, the code is intended to assign the value i to the i^{th} `SerialNumber` object that gets instantiated; in other words, the class designer's intention is that the value of the number attribute of the first `SerialNumber` object to be created should be `1`, the value of the number attribute of the second `SerialNumber` object to be created should be `2`, and so on.

Unfortunately, while the above `SerialNumber` class compiles without error, it contains a logical error: all calls to the `getNumber()` method always return `0`, regardless of the `SerialNumber` object used as a target for the call.

Describe what causes this logical error to occur, and suggest a simple way to fix the program. **BE SPECIFIC BUT BRIEF**; vague and overly long answers will be grounds for mark deductions.

DESCRIBE THE PROBLEM IN THE SPACE BELOW:

[5] 3. Consider the following program.

```
public class Vector2D {
  private double x;
  private double y;

  public Vector2D(double myX, double myY) {
    x = myX;
    y = myY;
  }

  public Vector2D add(Vector2D v) {
    System.out.println("add(Vector2D)");
    return new Vector2D(x + v.x, y + v.y);
  }

  public Vector2D add(int x, int y) {
    System.out.println("add(int, int)");
    return new Vector2D(x + x, y + y);
  }

  public Vector2D add(double x, double y) {
    System.out.println("add(double, double)");
    return new Vector2D(this.x + x, this.y + y);
  }

  public String toString() {
    return "(" + x + ", " + y + ")";
  }

  public static void main(String[] args) {
    Vector2D u, v;

    u = new Vector2D(1.0, 2.0);
    v = new Vector2D(3.0, 4.0);

    System.out.println("u + v == " + u.add(v));
    System.out.println("v + (5, 6.0) == " + v.add(5, 6.0));
    System.out.println("u + (7, 8) == " + u.add(7, 8));
  }
}
```

The above program compiles without error and runs without crashing; what does it display when it is executed?

WRITE THE PROGRAM'S OUTPUT IN THE SPACE BELOW:

Total marks for Section 1: $\overline{\textbf{15}}$

Section 2 - Long Questions

[10] 4. Consider the following class:

```
import java.util.Scanner;

public class MiniString {
  private char[] sequence;

  public MiniString(String s) {
    sequence = s.toCharArray();
  }

  public char charAt(int index) {
    return sequence[index];
  }

  public int length() {
    return sequence.length;
  }

  public String toString() {
    /* Point 1 */
    return new String(sequence);
  }

  public MiniString concatenate(MiniString other) {
    if (other == null) {
      /* Point 2 */
      return this;
    } else {
      char[] sequence = new char[this.length() + other.length()];
      for (int i = 0; i < sequence.length; i++) {
        /* Point 3 */
        if (i < this.length()) {
          sequence[i] = this.charAt(i);
        } else {
          sequence[i] = other.charAt(i - this.length());
        }
      }
      /* Point 4 */
      return new MiniString(new String(sequence));
    }
  }

  public static void main(String[] foo) {
    Scanner reader = new Scanner(System.in);
    /* Point 5 */
    MiniString first, second;

    System.out.print("Enter a String: ");
    first = new MiniString(reader.nextLine());
    System.out.print("Enter another String: ");
    second = new MiniString(reader.nextLine());
    System.out.println("The two Strings concatenated: " +
      first.concatenate(second));
  }
}
```

The above class compiles and runs without error.

Which variables are in scope at each of the points marked by comments in the above class? Do **NOT** list variables which can only be accessed using the `.` (dot) operator. 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 point.

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

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

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

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

(e) List all variables in scope at the point given by the comment `/* Point 5 */`. For each variable you list, indicate whether it is a **member variable**, a **formal parameter**, or a **local variable**.

[15] 5. The two classes below compile without error. What is displayed when the `main()` method of class `MadScramble` 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 `Gadget` class:

```
public class Gadget {
  private int left;
  private int right;
  private static int counter = 100;

  public Gadget(int myLeft) {
    left = myLeft;
    right = counter;
    counter = counter + 1;
  }

  public void scramble(Gadget g) {
    left = g.right;
    g.left = right;
  }

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

The definition of the `MadScramble` class:

```
public class MadScramble {
  public static void scrambleInts(int i1, int i2) {
    i1 = i2;
  }

  public static void scrambleArrayWithInt(int[] a, int i, int v) {
    a[i] = v;
  }

  public static void scrambleArraysOfInt(int[] a1, int[] a2) {
    a1 = a2;
  }

  public static void scrambleMatrixWithInt(int[][] m, int i1, int i2,
    int v) {
    m[i1][i2] = v;
  }

  public static void scrambleMatrixWithArray(int[][] m, int i, int[] a) {
    m[i] = a;
  }
```

```
public static void scrambleGadgetStates(Gadget g1, Gadget g2) {
  g1.scramble(g2);
}

public static void scrambleGadgetAddresses(Gadget g1, Gadget g2) {
  g1 = g2;
}

public static void scrambleArrayWithGadgetState(Gadget[] a, int i,
  Gadget g) {
  a[i].scramble(g);
}

public static void scrambleArrayWithGadgetAddress(Gadget[] a, int i,
  Gadget g) {
  a[i] = g;
}

public static void scrambleArraysOfGadgets(Gadget[] a1, Gadget[] a2) {
  a1 = a2;
}

public static void main(String[] args) {
  final int CONSTANT = 2;
  int[] myA1 = {3, 5};
  int[] myA2 = {7, 11};
  int[][] myM = {{13, 17}, {19, 23}};
  Gadget myG1 = new Gadget(29);
  Gadget myG2 = new Gadget(31);
  Gadget[] myGA1 = {new Gadget(37), new Gadget(41)};
  Gadget[] myGA2 = {new Gadget(43), new Gadget(47)};

  scrambleInts(myA1[0], myA1[1]);
  scrambleArrayWithInt(myA1, 1, CONSTANT);
  scrambleArraysOfInt(myA2, myA1);

  scrambleMatrixWithInt(myM, 0, 0, CONSTANT);
  scrambleArrayWithInt(myM[0], 1, CONSTANT);
  scrambleArraysOfInt(myM[0], myM[1]);
  scrambleMatrixWithArray(myM, 1, myA2);
  scrambleArrayWithInt(myA2, 0, CONSTANT);

  scrambleGadgetStates(myG1, myG2);
  scrambleGadgetAddresses(myGA1[0], myGA1[1]);
  scrambleArrayWithGadgetState(myGA2, 0, myG2);
  scrambleArrayWithGadgetAddress(myGA2, 1, myG2);
  scrambleArraysOfGadgets(myGA1, myGA2);

  System.out.println("1: {" + myA1[0] + ", " + myA1[1] + "}");
  System.out.println("2: {" + myA2[0] + ", " + myA2[1] + "}");
  System.out.println("3: {{" + myM[0][0] + ", " + myM[0][1] + "}, {" +
    myM[1][0] + ", " + myM[1][1] + "}}");
  System.out.println("4: " + myG1 + ", " + myG2);
  System.out.println("5: {" + myGA1[0] + ", " + myGA1[1] + "}");
```

```
    System.out.println("6: {" + myGA2[0] + ", " + myGA2[1] + "}");
  }
}
```

YOU MAY 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

This section involves a program which could be used in a library to manage subscribers and loans. The complete program involves seven classes, and your task will be to write the following five of these seven classes:

- `Loan`
- `Subscriber`
- `SubscriberFileReader`
- `ReportWriter`
- `ReportGenerator`

Complete details for these classes will be provided in the relevant questions.

The last two classes are `Date` and `LoanFilter`. **These two classes have already been implemented**, and you can use them in the classes that you write. Complete details for these two classes are specified on pages 29-30 of this examination, and no `import` statements are required for these two classes.

General notes which apply to all questions in this section:

- `null` **references**: Unless otherwise stated, you **MAY** assume that for every method or constructor you write, none of the parameters it accepts can be a `null` reference; in other words, your methods or constructors do not have to handle cases where the arrays or objects they take as parameters are `null` references. In addition, unless otherwise stated, you **MAY** also assume that none of the arrays your methods or constructors take as parameters **contain** `null` references.
- **Using the five classes you are asked to define**: Whenever a class that you are asked to write needs to create or handle objects which belong to a different class that you are asked to write, it **MUST** do so using the methods defined in this other class. Therefore, when writing a class, you **MAY** assume that all the classes in Section 3 (except the one you are currently writing) have been successfully implemented, even if you did not successfully complete the relevant question or have not even attempted it.
- **Encapsulation**: You **MUST** respect proper encapsulation practices; that is, the attributes and methods of your classes **MUST** be declared using the proper visibility modifiers.
- **Exceptions and** `try`**-**`catch` **blocks**. You **MAY** assume that no I/O-related problems will occur in the methods you write; in other words, these methods **DO NOT** have to explicitly check whether or not such problems occur, and **DO NOT** have to manage `java.io.IOExceptions` (or any other exceptions) using `try-catch` blocks.

[10] 6. Write a class called `Loan`; as its name implies, each instance of this class keeps track of the details of a document loan made by a library subscriber.

Each `Loan` object has the following attributes:

- The borrower of the document involved in the loan (a `Subscriber`)
- The call number of the document being borrowed (a `String`)
- The return date; that is, the date by which the document is to be returned (a `Date`)
- The return status of the loan; that is, whether or not the document has been returned by the subscriber (a `boolean`)

The Loan class **MUST** provide the following methods; note that in the description below, the phrase “this Loan” means “the Loan on which the method in question is called”.

- A constructor, which takes as parameter a Subscriber, a String, a Date, and a boolean, in this order; these parameters respectively represent the subscriber, call number, return date, and return status attributes of the newly-created Loan. The constructor initializes the attributes of the newly-created Loan so that their values are equal to those passed to the constructor as parameters.
- A method called getBorrower(), which takes no parameters and returns a reference to a Subscriber object representing the borrower attribute of this Loan.
- A method called getCallNumber(), which takes no parameters and returns a String representing the call number attribute of this Loan.
- A method called getReturnDate(), which takes no parameters and returns a reference to a Date object representing the return date attribute of this Loan.
- A method called isReturned(), which takes no parameters and returns a value of type boolean which is true if the document involved in this Loan has been returned, false otherwise.
- A method called setReturned(), which takes as parameter a value of type boolean, and sets the return status attribute of this Loan so that it is equal to the boolean parameter.

WRITE YOUR Loan CLASS IN THE SPACE BELOW:

[15] 7. Write a class called Subscriber; as its name implies, each instance of this class represents a subscriber authorized to borrow documents from a library.

Each Subscriber object has the following attributes:

- The name of the subscriber (a String)
- A list of all loans in which the subscriber is involved (by borrowing a document); this list can contain an arbitrary number of Loans, limited only by the memory available to the Java Virtual Machine.

The Subscriber class **MUST** provide the following methods; note that in the description below, the phrase "this Subscriber" means "the Subscriber on which the method in question is called".

- A constructor, which takes as its only parameter a String, and initializes the newly-created Subscriber so that the value of its name attribute is equal to the parameter String, and its list of loans is empty.
- A method called getName(), which takes no parameters and returns a String representing the value of the name attribute of this Subscriber.
- A method called findLoan(), which takes as its only parameter a String representing the call number of a loan, and returns a Loan. The method searches this Subscriber's list of loans for a Loan whose call number attribute is equal (in a case-**INSENSITIVE** manner) to the String parameter, and whose return status indicates that the document was not returned. If such a Loan exists, the findLoan() method returns it; otherwise, it returns null.
- A method called addLoan(), which takes as parameters a String, a Date, and a value of type boolean, and returns a value of type boolean. If this Subscriber's list of loans does not already contain a Loan whose call number attribute is equal (in a case-**INSENSITIVE** manner) to the String parameter and whose return status indicates that the document was not returned, then the addLoan() method creates a new Loan and sets the values of this new Loan's attributes appropriately using the parameters it accepts and this Subscriber as needed. Once the new Loan object is created, the addLoan() method adds it to this Subscriber's list of loans, and returns true.

  On the other hand, if this Subscriber's list of loans already contains a Loan whose call number attribute is equal to the String parameter and whose return status indicates that the document was not returned, then the addLoan() method returns false **WITHOUT** changing the state of this Subscriber object.

  HINT: Calling the findLoan() method may be useful when writing the addLoan() method.
- A method called getLoans(), which takes no parameters and returns an ArrayList of Loans that contains aliases to all the Loans currently stored in this Subscriber's list of loans.

  Changing the state of the ArrayList returned by this method **MUST NOT** change the list of loans of this Subscriber list of loans. Likewise, changing the state of this Subscriber's list of loans **MUST NOT** change the state of the ArrayList returned during any previous call to this method.

  HINT: This method can be written without using loops, by calling a method or constructor of the ArrayList class listed on pages 27-29.
- A method called getLoans(), which takes as its only parameter a LoanFilter and returns an ArrayList of Loans. The returned ArrayList of Loans contains aliases to all the Loans that belong to this Subscriber's list of loans **and** that are accepted by the LoanFilter parameter. In other words, for all Loans L stored in this Subscriber's list of

loans, L will be included in the ArrayList returned by this method if and only if calling the accept() method with the parameter LoanFilter as a target and L as a parameter returns true.

Changing the state of the ArrayList returned by this method **MUST NOT** change the list of loans of this Subscriber. Likewise, changing the state of this Subscriber's list of loans **MUST NOT** change the state of the ArrayList returned during any previous call to this method.

- A method called toString(), which takes no parameters, and returns a String which is a textual representation of this Subscriber. For example, if the value of this Subscriber's name attribute is Alan Turing, and the number of loans in this Subscriber's list of loans is 2, then the textual representation of this Subscriber will look as follows:

  ```
  Alan Turing: 2 loans
  ```

  On the other hand, if the number of loans in this Subscriber's list of loans had been 1, then the textual representation of this Subscriber would have looked as follows:

  ```
  Alan Turing: 1 loan
  ```

WRITE YOUR Subscriber CLASS IN THE SPACE BELOW:

[10] 8. Write a class called SubscriberFileReader. This class declares a single method with the following header:

```
public static Subscriber readSubscriberFile(String fileName)
  throws java.io.IOException
```

The readSubscriberFile() method processes a file whose name is given by the String parameter fileName. This file contains information from which **ONE** Subscriber object can be created, along with all the Loan objects **in** this Subscriber's list of loans. The method reads this information from the file, and creates a Subscriber object from this information. It then returns the resulting Subscriber object.

Inside the input file, the value of the name attribute of the Subscriber appears on the first line. Information used to create each of the Loans in the new Subscriber's loan list appears on each of the following lines. Also, every line in the file after the first line contains information used to create exactly one Loan object; in other words, the file does not contain any empty lines. Within a line, the information is listed in the following order:

- The value of the Loan's call number attribute (a call number **can never** contain whitespace)
- An integer representing the year component of the value of the Loan's return date attribute
- An integer representing the month component of the value of the Loan's return date attribute
- An integer representing the day component of the value of the Loan's return date attribute
- A boolean representing the Loan's return status

Within a single line, these values are separated by one or more whitespace characters (space or tab). Note that there also might be any number of whitespace characters (including none) on a line before the value of the Loan's call number attribute and after the value of return status attribute.

For example, suppose that Alan Turing is an authorized borrower at Schulich Library, and has borrowed two documents since subscribing:

- one with call number QA9.2-T87-2001, to be returned on November 1, 2009 and already returned
- one with call number QA7-T772-2004, to be returned on December 5, 2010 but not yet returned

The contents of the input file containing information about Alan Turing's file at Schulich Library would therefore look like this:

```
Alan Turing
QA9.2-T87-2001 2009 11 1 true
QA7-T772-2004 2010 12 5 false
```

You **MAY** assume that the input file follows the format description exactly; in other words, your method does not have to detect or handle any formatting errors.

If the value of the call number attribute is the same for two or more Loans created from information stored in the file, and the return status of these Loans is false, then **ONLY** the **first** such Loan in the file **MUST** be added to the new Subscriber; all subsequent Loans with the same call number attribute and a false return status **MUST** be discarded.

WRITE YOUR `SubscriberFileReader` IN THE SPACE BELOW:

[15] 9. Write a class called ReportWriter. This class declares a single class method with the following header:

```
public static void generateReport(Subscriber[] subscribers,
  String fileName) throws java.io.IOException
```

This method processes the objects stored in the the array subscribers and writes information to the file whose name is given by the String parameter called fileName. For each object in subscribers, the method retrieves the late loans. If a subscriber does have late loans, then the name of this subscriber is written to the file, along with information about all of this subscriber's late loans. If a subscriber **does not** have any late loans, then no information about this subscriber is written to the file.

Once it has processed all the Subscriber objects in subscribers, the method writes the total number of late loans to the file and terminates.

The format of the files generated by this method is best illustrated by an example. Suppose the array subscribers has length 3, and the Subscriber objects it contains represent subscribers Richard Stallman, Donald Knuth, and Alan Turing, in this order.

- Richard Stallman has three loans, two of which are late (call number QA76.76-C73-W55-2002, due November 1, 2010, and call number QA76.76-S46-O643-2006, due December 1, 2010).
- Donald Knuth has four loans, none of which are late
- Alan Turing has two loans, one of which is late (call number QA7-T772-2004, due on December 5, 2010)

In this case, the file generated by the generateReport() method would look like this:

```
LIST OF LATE DOCUMENTS

Date: 2010-12-08

- Subscriber: Richard Stallman
        - QA76.76-C73-W55-2002 (Due: 2010-11-01)
        - QA76.76-S46-O643-2006 (Due: 2010-12-01)
- Subscriber: Alan Turing
        - QA7-T772-2004 (Due: 2010-12-05)

Total number of late documents: 3
```

The format illustrated by the above example has the following general properties; the files generated by your method **MUST** match this format **EXACTLY**:

- The String "LIST OF LATE DOCUMENTS" appears on the first line.
- The String "Date: " appears on the third line, followed by the current date.
- Information about subscribers who have late loans, as well as these late loans, will appear on the following lines. Each of these lines contains information either about a subscriber who has late loans or about a late loan.
  - Lines which contain information about a subscriber who has late loans will consist of the String "- Subscriber: " followed by the subscriber's name.

  - Lines which contain information about a late loan will consist of a tab, followed by the call number of the document, the `String` `" (Due: "`, a text representation of the return date of this loan, and the String `")"`, in this order.
  - Information about the subscribers who have late loans appears in the order in which the corresponding `Subscriber` objects are stored in `subscribers`. The information about **all** of one specific subscriber's late loans appears on the lines which immediately follow the information about that subscriber.
- The `String` `"Total number of late documents: "` appears on the last line, followed by the total number of late loans.
- The file contains no empty lines, except for the second, fourth, and second-to-last lines.

HINTS:

- There is a **class field** in the `LoanFilter` class that can be used to easily determine whether or not a `Loan` is late.
- The no-parameter constructor of the `Date` class creates a `Date` object representing the current date.
- Use the `toString()` method of the `Date` class to produce the `String` representation of `Date` objects.

WRITE YOUR `ReportWriter` CLASS IN THE SPACE BELOW:

[10] 10. Write a class called `ReportGenerator`. This class declares a single method with the following header:

```
public static void main(String[] args) throws
  java.io.IOException
```

This `main()` method first checks if the number of command-line parameters it receives is greater than or equal to `1`. If there are no command-line parameters, the `main()` method should display the following error message to the standard error stream and terminate:

```
Usage: java ReportGenerator <outfile> [<infile>]*
```

If the number of command-line parameters passed to the `main()` method is greater than or equal to `1`, then the first parameter represents the name of an output file, and each of the following parameters represents the name of an input file which contains the information necessary to create a `Subscriber`. The method creates a `Subscriber` object from the information stored in each input file, and stores each new `Subscriber` object in a one-dimensional array. Note that the `Subscriber` object created from the information stored in the first input file **MUST** be stored in position `0` of the array, the `Subscriber` object created from the information stored in the second input file **MUST** be stored in position `1` of the array, and so on.

Once the `main()` method has read all input files, it generates a report of the late loans of all `Subscribers` in the array, and saves this report in the output file. It then terminates. Note that if no input files are provided, the `main()` method should just generate an output file which states that no loans are late.

WRITE YOUR `ReportGenerator` CLASS IN THE SPACE BELOW:

YOUR `ReportGenerator` CLASS CONTINUED:

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 MORE 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 String(char[] value)`: Allocates a new `String` so that it represents the sequence of characters currently contained in the character array `value`.
  - `public int length()`: Returns the length of this `String`.
  - `public char charAt(int index)`: Returns the `char` value at the specified `index`.
  - `public char[] toCharArray()`: Converts this `String` to a new character array.
  - `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; returns a negative value if this `String` occurs before `anotherString`, a positive value if this `String` occurs after `anotherString`, and `0` if both `String`s are equal.
  - `public int compareToIgnoreCase(String anotherString)`: Compares, ignoring case considerations, this `String` to `anotherString` lexicographically; returns a negative value if this `String` occurs before `anotherString`, a positive value if this `String` occurs after `anotherString`, and `0` if both `String`s are equal.
  - `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 int nextLong()`: Scans the next token of the input as an `long`.

- `PrintStream` (package `java.io`) Methods:
  - `public PrintStream(File file) throws java.io.FileNotFoundException`: Creates a new `PrintStream` which writes to the specified `File`.
  - `public PrintStream(String fileName) throws java.io.FileNotFoundException`: Initializes a new `PrintStream` which writes to the file with the specified `fileName`.
  - `public void close()`: Closes the stream.
  - `public void print(boolean b)`: Prints `boolean` value `b`.
  - `public void print(char c)`: Prints `char` value `c`.
  - `public void print(char[] s)`: Prints the array of `char` `s`.
  - `public void print(double d)`: Prints `double` value `d`.
  - `public void print(int i)`: Prints `int` value `i`.
  - `public void print(Object o)`: Prints `Object` `o`.
  - `public void print(String s)`: Prints `String` `s`.
  - `public void println()`: Terminates the current line by writing the line separator string.
  - `public void println(boolean b)`: Prints `boolean` value `b` and then terminates the line.
  - `public void println(char c)`: Prints `char` value `c` and then terminates the line.
  - `public void println(char[] s)`: Prints array of `char` `s` and then terminates the line.
  - `public void println(double d)`: Prints `double` value `d` and then terminates the line.
  - `public void println(int i)`: Prints `int` value `i` and then terminates the line.
  - `public void println(Object o)`: Prints `Object` `o` and then terminates the line.
  - `public void 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`.

- `Character` (package `java.lang`) Methods:

  - `public static boolean isDigit(char ch)`: Determines if character `ch` is a digit.
  - `public static int digit(char ch, int radix)`: Returns the numeric value of character `ch` in the radix `radix`, `-1` if `ch` does not represent a digit.
  - `public static char forDigit(int digit, int radix)`: Returns the character representation of `digit` in the radix `radix`.
  - `public static boolean isLetter(char ch)`: Determines if character `ch` is a letter.
  - `public static boolean isLowerCase(char ch)`: Determines if character `ch` is a lowercase character.
  - `public static boolean isUpperCase(char ch)`: Determines if character `ch` is an uppercase character.
  - `public static boolean isWhitespace(char ch)`: Determines if character `ch` is white space according to Java.
  - `public static char toLowerCase(char ch)`: Converts character `ch` to lowercase.
  - `public static char toUpperCase(char ch)`: Converts character `ch` to uppercase.

- `ArrayList<E>` (package `java.util`) Methods:
  - `public ArrayList<E>()`: Creates a new empty `ArrayList` whose elements are of type `E`.
  - `public ArrayList<E>(ArrayList<E> aL)`: Creates a new list whose elements are of type `E` and that contains aliases of all the elements of `aL`.
  - `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`; comparisons are performed using the `equals()` method on `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; comparisons are performed using the `equals()` method on `o`.
  - `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; comparisons are performed using the `equals()` method on `o`.
  - `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 QUESTIONS 6-10

- Class `Date`: Objects of this class represent dates in the Gregorian (usual) calendar.
  - `public Date()`: Creates a new `Date` representing the day on which this `Date` was created.
  - `public Date(int year, int month, int date)`: Creates a new `Date` representing the moment in time specified by the parameters.
  - `public int getYear()`: Returns the value of the year attribute of this `Date`.
  - `public int getMonth()`: Returns the value of the month attribute of this `Date`.
  - `public int getDay()`: Returns the value of the day attribute of this `Date`.
  - `public boolean equals(Object otherObject)`: Determines whether or not `anObject` is a `Date` object representing the same day as the one represented by this `Date`.
  - `public int compareTo(Date otherDate)`: Compares this `Date` to `otherDate`; returns a negative value if this `Date` occurs before `otherDate`, a positive value if this `Date` occurs after `otherDate`, and `0` if both `Date`s are equal.

  - `public String toString()`: Returns a textual representation of this `Date`.

- `LoanFilter`: Objects of this class represent conditions which `Loan`s can satisfy.
  - `public boolean accept(Loan loan)`: Returns `true` if `loan` satisfies the condition represented by this `LoanFilter`.
  - `public static final LoanFilter LATE_LOANS`: A `LoanFilter` whose `accept()` method returns `true` if the parameter `Loan` represents a late loan, `false` otherwise.