ASSIGNMENT 3
Classes, Objects and the Robot World

COMP-202B, Winter 2009, All Sections

Due: Tuesday, March 3, 2009 (23:55)

You **MUST** do this assignment individually and, unless otherwise specified, you **MUST** follow all the general instructions and regulations for assignments. Graders have the discretion to deduct up to 10% of the value of this assignment for deviations from the general instructions and regulations.

Part 1, Question 1:	0 points
Part 2, Question 1:	45 points
Part 2, Question 2:	55 points
	100 points total

Introduction

In this assignment, we provide you with an object-oriented way to model the world using cities, robots and other things. This system was developed by Byron Becker at the University of Waterloo.

Part of this assignment involves reviewing the application programming interface (API) for the classes provided in the `becker.robots` package and using this information to write programs in which a robot performs specific tasks. To complete this assignment, you will need the following files:

- `a3-becker.jar`: An archive file containing all the Java classes not already included in the Java Standard Class Library that you may need to use in order to complete the classes required by this assignment.
- `a3-becker-docs.zip`: An archive file containing documentation about the classes included in the provided `a3-becker.jar` file that you are likely to use to complete this assignment. Note that the only class whose use is absolutely required is `Robot` (in package `becker.robots`), although you may find of other classes useful, such as the `Direction` enum (also in package `becker.robots`).

Part 1 (0 points): Warm-up

Do **NOT** *submit this part, as it will not be graded. However, doing these exercises might help you to do the second part of the assignment, which will be graded. If you have difficulties with the questions of Part 1, then we suggest that consult the TAs during their office hours; they can help you and work with you through the warm-up questions.*

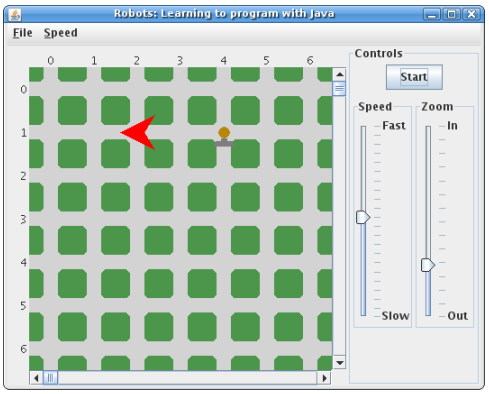
Warm-up Question 1 (0 points)

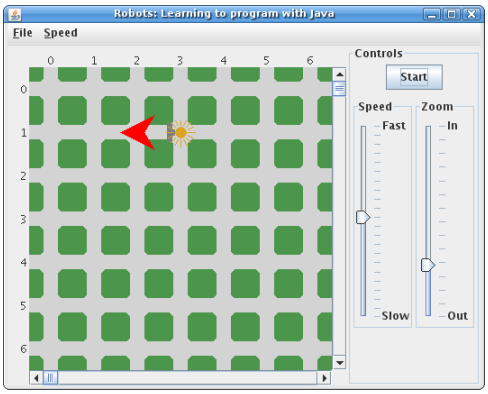
A construction crew needs to move a flashing light one intersection to the west of where it is currently located. They decide to use a robot to perform this task. However, the robot is currently located 2 intersections west of the flashing light, and faces west.

Your task involves programming the robot so that it performs the following actions:

- Move to the intersection where the flashing light is located, and pick up the light
- Move to the intersection where the flashing light should be moved, and put the light there
- Go back to its starting position

The following images show the robot and the flashing light both before and after the robot moves the light:





A file named `MoveLight.java` is provided to you as a starting point. A class called `MoveLight` is defined in this file, and this class defines an incomplete `main()` method. Some of statements already present in this method deserve additional explanations:

- The statement

  ```
  City montreal = new City();
  ```

  declares a variable of type `City` called `montreal`, creates a new object which belongs to the `City` class, and stores the address of this new object in variable `montreal`.

  The object created in this statement represents the city where the robot and the flashing light are located. In such a city, streets run from west to east (with their number increasing as one moves east), and avenues run from north to south (with their number increasing as one moves south).

- Likewise, the statement

  ```
  Robot asimo = new Robot(montreal, ROBOT_STREET, ROBOT_AVENUE,
    Direction.WEST);
  ```

  declares a variable of type `Robot` called `asimo`, creates a new object which belongs to the `Robot` class, and stores the address of this new object in variable `asimo`.

  The object created in this statement represents the robot whose task will be to move the flashing light. The robot starts at the intersection of `ROBOT_STREET` Street and `ROBOT_AVENUE` Avenue of the `City` whose address is stored in variable `montreal`; `ROBOT_STREET` and `ROBOT_AVENUE` are two `int` constants declared at the beginning of the `main()` method of the `MoveLight` class. The robot initially faces west.

- Finally, the statement

  ```
  new Flasher(montreal, FLASHER_STREET, FLASHER_AVENUE, true);
  ```

  creates a new object which belongs to the `Flasher` class. Even though the address of this new object is not assigned to a variable, the object is not garbage-collected; this is due to the way the constructor for the class is implemented, and knowledge of the details is not required or even desirable to complete this question.

The object created in this statement represents the flashing light that the robot must pick up. When it is created, it is located at the intersection of `FLASHER_STREET` Street and `FLASHER_AVENUE` Avenue of the `City` whose address is stored in variable `montreal`; again, `FLASHER_STREET` and `FLASHER_AVENUE` are two `int` constants declared at the beginning of the `main()` method of the `MoveLight` class. The last parameter, `true`, indicates that the light is flashing.

Your task therefore consists of adding the appropriate statements to the `main()` method of the provided `MoveLight` class so that the `Robot` whose address is stored in variable `asimo` moves the `Flasher` to the appropriate location. Note that the statements which initialize the `City`, `Robot`, and `Flasher` objects also ensure that they are displayed to the screen; therefore, you do not have to worry about updating the display.

Information on the methods defined in the `Robot` class is available from the documentation included in `a3-becker-docs.zip`. In particular, the following methods **MAY** be useful: `getDirection()`, `move()`, `turnLeft()`, `pickThing()`, and `putThing()`. You **MAY** also find it useful to consult the documentation for the `Direction` enum; an enum is basically syntactic sugar for a class which defines only class constants, and the constants it defines can be used like any other normal class constant. Documentation for the `Direction` enum is also included in `a3-becker-docs.zip`.

Hint: Start small, compile often, and test often. For example, start by adding a single statement which makes the robot turn in one direction. Compile the program, run it (see the instructions below on compiling and runnin programs which use the classes included in `a3-becker.jar`), and observe the result. Then, replace the statement that makes the robot turn by one that makes it move one block. Compile the program, run it, and observe the result. Keep experimenting with the library (by writing various small combinations of statements) until you gain a solid enough understanding of what the various methods do. Then, attempt to write the program that performs the task.

Warm-up Question 2 (0 points)

Modify the program you completed in the previous question so that the robot will correctly move the light regardless of the value of the `FLASHER_AVENUE` constant, as long as it is greater than or equal to 4.

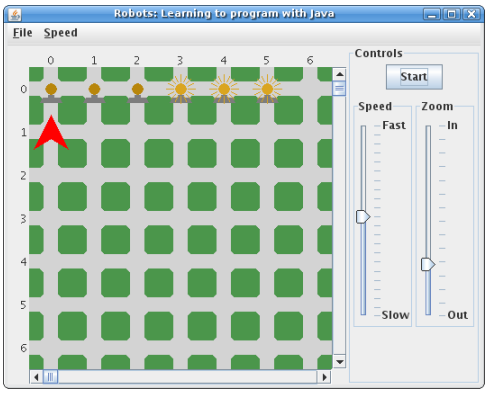
Part 2 (45 + 55 = 100 points)

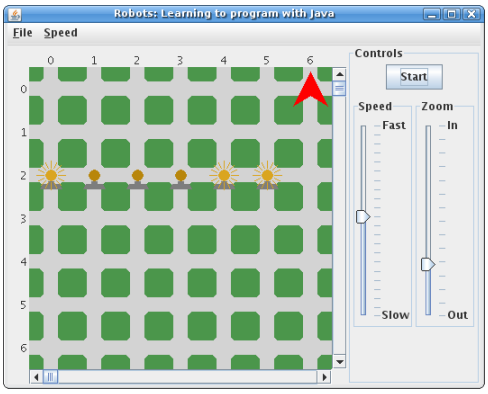
The questions in this part of the assignment will be graded.

Question 1: Moving a Series of Flashing Lights (45 points)

Construction crews have placed flashing lights along a certain street in the city, one at each intersection between two avenues. However, these flashing lights need to be moved a number of blocks south of where they currently are. Again, the crew members decide to use a robot to perform this task; initially, the robot is located one block south of the westmost flashing light, and faces north.

The following images show the robot and the flashing lights both before and after the robot moves 6 lights located between 0th Avenue and 5th Avenue on 0th Street 2 blocks south:





A file named `MoveAllLights.java` is provided to you as a starting point. A class called `MoveAllLights` is defined in this file, and this class defines an incomplete `main()` method which contains statement that initialize the city, the robot, and the flashing lights. The constants declared at the beginning of this method deserve additional explanations:

- `FLASHER_STREET`: The number of the street where the westmost flashing light is initially located.
- `FLASHER_AVENUE`: The number of the avenue where the westmost flashing light is initially located.
- `NUMBER_FLASHERS`: The number of flashing lights to be moved. Initially, the flashing lights are located at every intersection on `FLASHER_STREET` Street between `FLASHER_AVENUE` Avenue and the avenue whose number is given by (`FLASHER_AVENUE` + `NUMBER_FLASHERS` - 1), inclusively.
- `DISTANCE`: The number of blocks by which to move the flashing lights. That is, if the flashing light are initially on `FLASHER_STREET` Street, after having been moved, they will be on (`FLASHER_STREET` + `DISTANCE`) Street.

Your task therefore consists of adding the appropriate statements to the `main()` method of the provided `MoveAllLights` class so that the `Robot` whose address is stored in variable `asimo` moves the `Flashers` to the appropriate locations.

Notes:

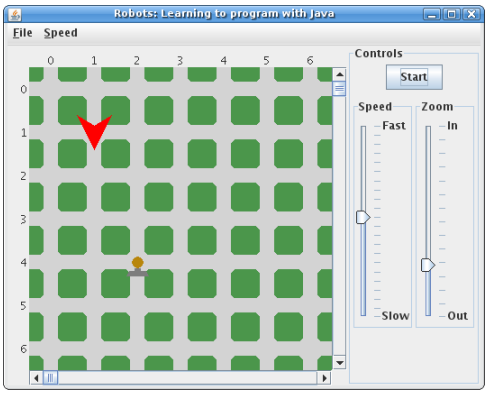
- Your completed program **MUST** work regardless of the initial locations of the flashing lights or of the number flashing lights that need to be moved; however, you **MAY** assume that all the flashing lights are initially located on adjacent intersections along a street, and that the robot always starts one block south of the westmost flashing light, facing north.
- Your program **MUST** work regardless of the distance by which the flashing lights are to be moved, as long as this distance is greater than 0.
- Finally, the position and direction of the robot after it finishes moving the flashing lights can be anything, as long as it is **NOT** on the same intersection as a flashing light.

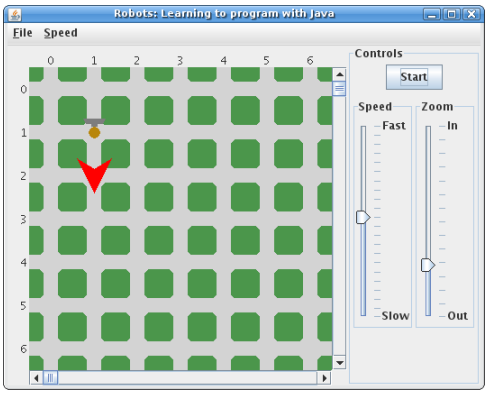
Question 2: Searching for a Lost Flashing Light (55 points)

After finishing their work, construction crews picked up their flashing lights and drove off. However, on their way back, they realize that one of the flashing lights is missing. They figure that the missing flasher is probably located in a certain area of the city (which we will refer to as the *search area* from now on), and decide to use a robot to search this area for the missing flashing light.

Initially, the robot is located in the northwest corner of the search area, facing south. The robot's task is to search the entire area (including its edges) for the flashing light. If the robot finds the missing flashing light, it brings it back to its initial position (that is, the northwest corner of the search area). Regardless of whether or not it finds the missing flashing light or searches the entire area without finding it, once the robot is done, it must park itself one block south of its initial position, facing south.

The following images show the robot and the flashing light both before and after the robot brings the light back. The light was lost at the corner of 4th Street and 2nd Avenue, and the search area is formed by a rectangle whose sides consist of 1st Street, 5th Street, 1st Avenue, and 5th avenue (inclusively):





A file named `SearchLight.java` is provided to you as a starting point. A class called `SearchLight` is defined in this file, and this class defines an incomplete `main()` method which contains statement that initialize the city, the robot, and the flashing light. The constants declared at the beginning of this method deserve additional explanations:

- `NORTHWEST_STREET`, `NORTHWEST_AVENUE`: The numbers of the street and avenue whose intersection forms the northwest corner of the search area.
- `SOUTHEAST_STREET`, `SOUTHEAST_AVENUE`: The numbers of the street and avenue whose intersection forms the southeast corner of the search area.
- `FLASHER_STREET`, `FLASHER_AVENUE`: The numbers of the street and avenue whose intersection is where the flashing light was lost. It may or may not be inside the search area.

Your task therefore consists of adding the appropriate statements to the `main()` method of the provided `SearchLight` class so that the `Robot` whose address is stored in variable `asimo` looks for the flashing light inside the search area, and brings it back to the northwest corner of the search area if it finds it. Once the robot brings the flashing light to the northwest corner of the search area, or searches the entire area without finding the flashing light, it **MUST** position itself one block south of its initial location, facing south, and stop moving.

Notes:

- Your completed program **MUST** work regardless of the location or size of the search area; however, you **MAY** assume that the souteast corner of the search area is indeed southeast of the northwest corner. You **MAY** also assume that the robot always starts facing south in the northwest corner of the search area, regardless of the latter's location.
- You **MUST NOT** use the constants `FLASHER_STREET` and `FLASHER_AVENUE` in **ANY** of the code that you add to `SearchLight.java`; in other words, the robot **MUST** be programmed as if it does not know the location of the missing flashing light.
- If the flashing light is located inside the search area, the robot **MUST NOT** continue looking for the light once it finds it, but instead proceed to bring the light back to the northwest corner of the search area as soon as it finds it.

Additional Information

Other tips and information:

- To combine the Java libraries in `a3-becker.jar` with the classes you write so that everything compiles and runs using the JDK, you need to specify when compiling or running your program that `a3-becker.jar` should be on the *Classpath*. To do so, follow the steps below:

  1. Place the file `a3-becker.jar` in the same directory / folder as the one in which the files containing the classes that you wrote.
  2. Open the command-line interface (Command Prompt under Windows, Terminal under Mac OS X, Linux, or other Unix-like operating systems) and navigate normally to the directory / folder containing your files.
  3. To compile a file called `MyFile.java` under Windows, issue the following command:

     ```
     javac -cp a3-becker.jar;.  MyFile.java
     ```

     To compile a file called `MyFile.java` under Mac OS X, Linux, or another Unix-like operating system), issue the following command:

     ```
     javac -cp a3-becker.jar:.  MyFile.java
     ```

     The only difference between the Windows and Unix versions of the above command is that in the Windows version, a semi-colon (`;`) separates `a3-becker.jar` from the dot (`.`) that follows; in the Unix version, a colon (`:`) separates `a3-becker.jar` from the dot (`.`) that follows. Make sure to replace `MyFile.java` with the actual name of the file you wish to compile.
  4. To run the `main()` method defined in a class called `MyFile` under Windows, issue the following command:

     ```
     java -cp a3-becker.jar;.  MyFile
     ```

     To compile a file called `MyFile.java` under Mac OS X, Linux, or another Unix-like operating system), issue the following command:

     ```
     java -cp a3-becker.jar:.  MyFile
     ```

     Again, the only difference between the Windows and Unix versions of the above command is that in the Windows version, a semi-colon (`;`) separates `a3-becker.jar` from the dot (`.`) that follows; in the Unix version, a colon (`:`) separates `a3-becker.jar` from the dot (`.`) that follows. Make sure to replace `MyFile` with the actual name of the class containing the `main()` method you wish to run.

  You should now be able to compile and run the `main()` methods of the classes that you are required to write by the above specification without the compiler or virtual machine reporting errors related to missing classes. Note that the `-cp` option is **ONLY** necessary when the code you want to compile / run requires libraries located in another directory.

- To combine the Java libraries in `a3-becker.jar` with the classes you write so that everything compiles and runs using Eclipse, you need to add `a3-becker.jar` to the Classpath of your project. To do so, follow the steps below:

  1. In the `Package Explorer` or `Navigator` view, right-click on your project. In the context menu that appears, select the `Properties` option.
  2. The window that appears will be split in two main portions. The smaller, left-hand portion contains a number of option categories, and you can select a category from those listed; the larger middle portion contains the options for the selected category. In the left-hand portion of the window, choose `Java Build Path`.
  3. The middle portion of the window will now contain four tabs. Click on the `Libraries` tab. The libraries included in your project will now appear, along with a number of buttons; click on the button labelled `Add External JARs...`.

4. Using the file selection window that appears, navigate to the directory / folder where you saved `a3-becker.jar`, select the latter, and click on the confirmation button (it might be labelled `OK`, `Open`, or something else, depending on the operating system that you use).

5. Selecting `a3-becker.jar` as described in the previous step you should bring you back to the `Properties` window, and the list of libraries included in your project will now contain one additional element: `a3-becker.jar`. At the bottom of the `Properties` window are two buttons; click on the one labelled `OK`.

You should now be able to compile all the classes that you are required to write by the above specification without the compiler reporting errors related to missing classes. Note that it is necessary to perform the above steps **ONLY** when the code you want to compile / run requires libraries not included in the Java Standard Library.

What To Submit

`MoveAllLights.java`
`SearchLight.java`