Programming Structures and Abstractions (CS 2334)
Project 1

September 12, 2009

Introduction

In this project, you will be practicing your skills in defining classes and class hierarchies. Specifically, you will be creating a set of classes that will allow us to represent and manipulate elemental actions for our Finches.

Milestones

1. Use program arguments to specify a file name (10 pts)
2. Create a superclass called **FinchAction** that represents general Finch actions (10 pts)
3. Create several subclasses for different action types: **FinchJog, FinchTone and FinchFlashBeak** (20 pts)
4. Create a class called **FinchActionList** that represents a list of FinchActions (10 pts)
5. Use simple File I/O to read a file and create a "populated" instance of FinchActionList (10 pts)
6. Implement a program that allows a user to search for and execute FinchActions in the list by name (15 pts)

Other components:

- Develop and use a proper design (UML and class stubs) (15 pts)
- Use proper documentation and formatting (javadoc and in-line documentation) (15 pts)

This lab is due in two phases:

1. Thursday, September 17th at 5:00pm: design.
2. Thursday, September 24th at 5:00pm: completed program and short demonstration.

More details for what to hand-in and when may be found below.

Resources

Main class web page:

- Java JDK 6 Classes
- Finch Introduction
- Finch API: how to talk to your Finch
- FinchSoftwarev3_1.zip: core Finch code and example programs (download and install on your hard disk)

Main web page / projects / general :

- Finch Driver Setup: setting up your computer to talk to the Finch
- Finch Manual: general information about the Finch hardware
- Documentation_Requirements
- Submission Instructions

Main web page / projects / project1 :

- project1.pdf: this project description
- cs2334_finch.pdf: a copy of the lab section slides
- dance.txt: an example input file
- ... other example files to come...

Notes

The key data elements are represented as follows (note some differences from our example in class):

- Action names are Strings
- All time units are in **milliseconds** (int)
- All distances are centimeters (double)
- All velocities are in centimeters/second (double)
- All frequencies represented as cycles/second (Hertz) using **integers**

Input Files

An input file will be composed of a set of lines. Each line will correspond to a single action and will be in one of the following forms:

- Jog the finch:

  ```
  <name> JOG <duration> <left> <right>
  ```

- Change the beak color and wait for duration:

  ```
  <name> RGB <duration> <red> <green> <blue> <darken>
  ```

  Color channels are integers in the range of [0 ... 255]

  darken is either 0 or 1 and indicates whether the beak is turned off after the action is complete.

- Generate a sound of a given duration:

  ```
  <name> SOUND <duration> <frequency>
  ```

Example File

```
forward_short JOG 2000 10.0 10.0
forward_long  JOG 5000 20.0 20.0
forward_left  JOG 2000 10.0 20.0
forward_right JOG 2000 20.0 10.0
dance         RGB 100 255 30 0 0
dance         JOG 2000 15.0 15.0
dance         RGB 100 0 30 255 1
dance         SOUND 1000 200
dance         JOG 4000 -20.0 20.0
dance         SOUND 500 400
dance         RGB 100 0 255 20 0
```

After loading of this file into your data structure, the user will be able to search for a particular name and either display or execute the sequence of FinchActions that match the name. If the user specifies the name as "all", all of the FinchActions are displayed/executed in order.

Milestones

A milestone is a "significant point in development." Milestones serve to guide you in the design and development of your project. Listed below are a set of milestones for this project along with a brief description of each.

Milestone 1: Use program arguments to specify a file name

Sometimes it is handy to be able to give a program some input when it first starts executing. Program arguments can fulfill this need. In Eclipse, these arguments are equivalent to MS-DOS or Unix command line arguments. Program arguments are handled in Java using a String array that is traditionally called **args** (the name is actually irrelevant, however).

The following program will print out the program arguments:

```
public static void main(String[] args) {
    System.out.println(args.length + " program arguments:");
    for (int i=0; i< args.length; i++) {
        System.out.println("args[" + i + "] = " + args[i]);
    };
};
```

For project 1, the name of the file that stores the list of Finch actions will be passed to your program using program arguments. The main method for your program should use the first argument specified by the user as the name of the file to load. Note that it is considered an error if the user specifies zero or more than one arguments to the program. Your program should detect this error and print out an appropriate error message before exiting.

In Eclipse, you can enter command line arguments as follows:

1. Select your driver class in the **package explorer** window on the left hand side of the Eclipse window.
2. Click the **run button** (green button with the white arrow) and then click the **run** menu option. (This mutton may be called **Configuration Options** in some versions)
3. Select the **(x) = Arguments** tab.
4. In the **Program arguments** window, type in your arguments. In general, these arguments are space-separated. However, for the purposes of this project, you should have exactly one argument (the input file name).
5. Click **Apply**.
6. Then you can **Run**.

Milestone 2: Create the FinchAction Class

This superclass should:

- Provide class variables that are common to all children. Note that **int duration** represents durations in milliseconds. Also note that negative durations are not meaningful and therefore should not be allowed by the class constructor or mutator methods.
- Provide a reasonable set of constructors that allows for flexibility in how the class is used.
- Provide an appropriate set of accessor and mutator methods.
- Provide a toString() method.
- Force the child classes to provide the following method:

```
public void execute (Finch myFinch, String name)
```

Milestone 3: Create the FinchJog, FinchTone and FinchFlashBeak Classes

These subclasses should:

- Provide appropriate class variables.
- Provide a flexible set of constructor methods that make appropriate use of the superclass constructors.
- Provide an appropriate set of mutators and accessors.
- Provide a **toString()** method.
- Provide an **execute()** method that will send appropriate commands to your Finch.

FinchTone Notes:

- Represent frequency in Hertz using an **integer**. Because negative frequencies do not have any meaning, your class implementation should not allow the user class to specify negative frequencies. Your class may respond to such errors by setting the frequency to some default (but legal) value.

FinchFlashBeak Notes:

- Represent color as an array of three integers capturing red, green, and blue illumination levels. These integer values may only take on values between 0 and 255. Your class methods should enforce this range.

FinchJog Notes:

- Provide **setDistance()** methods for both the left and right wheels. These methods should assume that duration is already specified and deal appropriately with the case in which duration = 0.

Milestone 4: Create the FinchActionList Class

This class represents an ordered list of individual FinchActions. The class definition should:

- Provide appropriate class variables.
- Provide a no-parameter constructor that creates an initial array for the storage of the list of actions.

- Provide the following method that adds new actions to the end of the list:

```
public void add(FinchAction action)
```

  Note that this method should deal appropriately with the internal array becoming full by expanding the size of the array.

- Provide a method that will print (in order) the list of actions to the console that match a given name:

```
public void display(String name)
```

  Note that names are case insensitive (capitalization does not matter), and that a name of "all" will result in the display of all actions.

- Provide a method that will execute (in order) the list of actions that match a given name:

```
public void execute(Finch myFinch, String name)
```

- **Note: you are to create your own class using arrays and any necessary primitive types. Do not use other java classes that represent lists, sets, vectors, etc.**

Milestone 5: Use simple File I/O to read a file

We will discuss File I/O in depth later in the class; this project is just designed to give you a brief introduction to the technique. Reading files is accomplished in Java using a collection of classes in the java.io package. To use the classes you must import the following package:

```
import java.io.*;
```

The first action is to open the file. This associates a variable in the program with the name of the file sitting on the disk:

```
FileReader fr;

String fileName = "dance.txt";
try {
    fr = new FileReader(fileName);
}catch(Exception e){
    // An error has occurred
    System.out.println(``Error opening file '' + fileName);
    return(null);  // Or take some other action to quit the program
}
```

Don't worry about the details of the try-catch keywords and exceptions at this time (other than to detect and address the errors). We will talk about these later in the semester.

Next, the FileReader is wrapped with a BufferedReader. A BufferedReader is more efficient than a FileReader since a BufferedReader saves groups of characters during a single operation instead of working with characters individually. Another advantage of using a BufferedReader is that there is a command to read an entire line of the file, instead of a single character at a time. This feature comes in particularly handy for this project:

```
BufferedReader br = new BufferedReader(fr);
```

The BufferedReader can now read in Strings:

```
String nextline;
try {
    nextline = br.readLine();
}catch(Exception e){
    System.out.println(``Error reading file.'');
    return(null);  // Or take some other action to quit the program
}
// Now do something with nextline...
```

Now that you have read in a line from the file, you must extract the individual words (also called *symbols* or *tokens*). For the purposes of this project, you may use the split() method provided by the String class (do not use other mechanisms such as the StringTokenizer class). The following will split your line into a set of words that are separated by *white space* (one or more spaces or tabs):

```
String[] string_list = nextline.split(``\\s+'');
```

You may now work through string_list to extract the information that you need to create your action objects. Note that **Integer.parseInt()**, **Double.parseDouble()**, and **String.compareToIgnoreCase()** will be useful for these purposes. Should a line not contain enough words or the line is malformed in some way, simply ignore the line and continue with the next.

Look at the Java API listing for BufferedReader and find out what readLine() returns when it encounters the end of the file (also called a *stream*).

When you are finished with the BufferedReader, the file should be closed. This informs the operating system that you are finished using the file:

```
br.close();
```

Closing the BufferedReader also closes the FileReader.

Milestone 6: Implement a high-level driver program

The main method performs the following functions:

- Creates an instance of the Finch class.
- Reads the file that is specified at the command line and creates a FinchActionList from this file.
- Enters a loop that accepts keyboard input from the user.
  - The input can be "ALL" or a name.
  - The list of matching FinchActions is displayed and then executed.
  - "QUIT" closes the connection to the Finch object and exits your program.

Accepting input from the keyboard is very similar to accepting input from a file:

```
BufferedReader input =
    new BufferedReader(new InputStreamReader(System.in));

String strg;

// Loop once for each input line
while(true){
    // Get the next line
    try {
        strg = input.readLine();
    }catch (Exception e) {
        // Deal with the error in some way
    }
    // Do something with the string

}
```

In Eclipse, you can see the strings that are "printed" by your program and enter keyboard input in the **Console** tab at the bottom of the Eclipse window.

The connection to the Finch is closed using:

```
myFinch.stopWheels();
myFinch.quit();
```

Should you forget to properly close the Finch, it may not respond to future connection requests. To reset the Finch, simply unplug it from your computer and plug it back in.

Hand-In Procedures

Part 1: Design

Deadline: Thursday, September 17th at 5:00pm

Each group must hand in one copy of each of the following:

1. A printed cover page that lists the group members, work contributed by each, and any outside citations. Turn in the hardcopy to the TA or the lecturer.

2. UML diagram on engineering paper: turn in the hardcopy to the TA or the lecturer.

3. project1_design.zip to D2L. This is the zip file produced by Eclipse that contains:

- Each of the classes with class variables, method header documentation and method stubs (prototypes). **Other than dummy return calls, do not include any other code (points will be subtracted if other code is included).** The dummy return calls will enable you to compile your java code.
- html description of your project produced by javadoc. Make sure to check that the resulting html files contain all of the correct information.

Part 2: Complete program and short demonstration.

Deadline: Thursday, September 17th at 5:00pm

Each group must do the following:

1. A printed cover page that lists the group members, work contributed by each, and any outside citations. Turn in the hardcopy to the TA or the lecturer.
2. Hand in the modified UML diagram on engineering paper: turn in the hardcopy to the TA or the lecturer.
3. Turn in project1.zip to D2L. This is the zip file produced by Eclipse that contains:
   - Each of the class implementations with documentation. The author(s) of each class should be documented at the top of the java file.
   - html description of your project produced by javadoc.
4. A short demonstration. We will reserve time during your laboratory section for you to demonstrate your working program. You may also attend office hours or make appointments to perform the demonstrations.

Hints

- Start your design with your UML diagram. Given this diagram, produce the required components for part 1. It is likely that you will make changes to these as you start your implementation for part 2. This is OK!
- Implement and test your classes incrementally. Convince yourself that the small pieces are working properly before you move on to more complicated pieces. Note that you may implement multiple "driver" classes to test these various components.

- As you implement your classes, make sure to keep your UML diagram and your documentation up to date.
- Implement your project by following the sequence of milestones that we have supplied.
- Work closely with your partner. Ideally, work will be done only when both of you are present. Also: as one person is typing, the other should be looking over the shoulder of the "driver" to make explicit statements about what should be typed next and to check what is being typed.
- Your program must be your own work. Do not discuss or look at the solutions of other groups in the class. However, you may discuss general issues (i.e., not directly related to the project requirements) with your classmates, as well as use the book and the resources available on the net.
- Start your work early. This is not a trivial programming assignment.
- Ask for help early. If you are stuck on something, talk to the TA or the instructor sooner than later (this is what we are here for).
- See the Finch API documentation for a list of methods that will allow you to access the sensors and to produce behavior.
- Use the **Finch.buzz()** method to produce tones.
- Wheel motions may be inprecise on the Finches, especially at low velocities. In addition, there will be some variation in behavior between the individual Finches.