Object-Oriented Programming in Java

Problem Set 2 — Due: Jan 17, 2001

Exercises

Exercise 1: Goodbye World

Write a program that opens a window, prints "Goodbye World" in it, and exits properly when the close window ('x') icon on the window title bar is clicked.

First try doing it using the `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);` method as described in Chapter 7. If you are ambitious, try catching the WindowEvents following the code developed in Chapter 8 (page 364+). Try using anonymous inner classes in your implementation.

Exercise 2: Threads

Extend the `Thread` class to write two classes, `Walk` and `Chew`. Have `run` method of `Walk` print out "left" then "right" forever. Have `run` method of `Chew` print out "chomp" forever. Allocate and start both `Walk` and `Chew` and examine the output. You will have to use control-c to quit.

[To give credit where it is due: This exercise was strongly inspired by *Mr. Bunny's Bug Cup O' Java*]

Problems

Problem 1: File I/O and Exception Handling

The Caesar Cipher is a (very insecure) method for encrypting text dating back to the Romans. It is a simple alphabetic shift cypher that replaces each letter in the text to be encoded with the letter 3 later in the alphabet. For example, 'A' gets coded as 'D' and 'L' gets coded to 'O'. Letters at the end of the alphabet wrap around to the beginning, so 'Z' gets coded as 'C'. To decode the message, one simply reverses the shift.

Java will let you treat 'char' variables as number under some circumstances, you can shift by `n` characters by adding `n` to a character. For example:

```
int shift = 3; // for example
char input - 'c';
char output = input + 3; // output <- 'f'
```

One can generalize this technique by using a shift other than 3, the value of the shift then becomes the cipher 'key'. One can also generalize to handle characters other than capital letters. Most of the interesting printable characters lie between ' ' (ASCII 32) and ' ' (ASCII 126). If we shift within this space, we can encode most messages. The shift algorithm now becomes:

```
int temp = input + shift;  // compute new char
// handle wrap around (127 == '~', 32 == ' ')
if(temp >= 127) temp = 32 + (temp - 127);
char output = (char) temp; // convert back to 'char'
```

Write a Java program Encode with command line usage:

```
java Encode key infile [outfile]
```

This program opens the file *infile*, reads and encrypts each line using the shift *key* (a number), then writes each line to *outfile*. If *outfile* is not given the program should print to the console.

The program should be robust to error conditions including:

- Wrong number or type of arguments given (for example, no infile or a key that is not a number or too big),
- Infile not found or readable,
- Outfile not found or writable,
- Infile is binary or contains non-printable characters.

By robust, we mean the program should exit gracefully and print out a useful error message.

Note: This problem is about opening, reading, writing, and closing text files, and handling error conditions and exceptions gracefully. If you find the encrypting a distraction, skip it and just copy the files verbatim. If, on the other hand, you enjoy it, try extending your program to take a word as a key, using each letter of the keyword in turn as the start of the shift. For example, if the keyword is "cat" the first letter of the message is encoded with shift ('c' - ' '), the second with shift ('a' - ' '), and so on. When the end of keyword is reached, start over at the beginning. (This is a variant on the Vigenere Cipher, somewhat more secure than the simple shift, but nowhere near good enough to befuddle the NSA.)

Problem 2: SameJava

Below is a screen-shot of the Gnome game "Same Gnome", available as part of the gnome-games package. The goal of this problem is to implement this game in Java, though with less ambitious rendering of the balls.

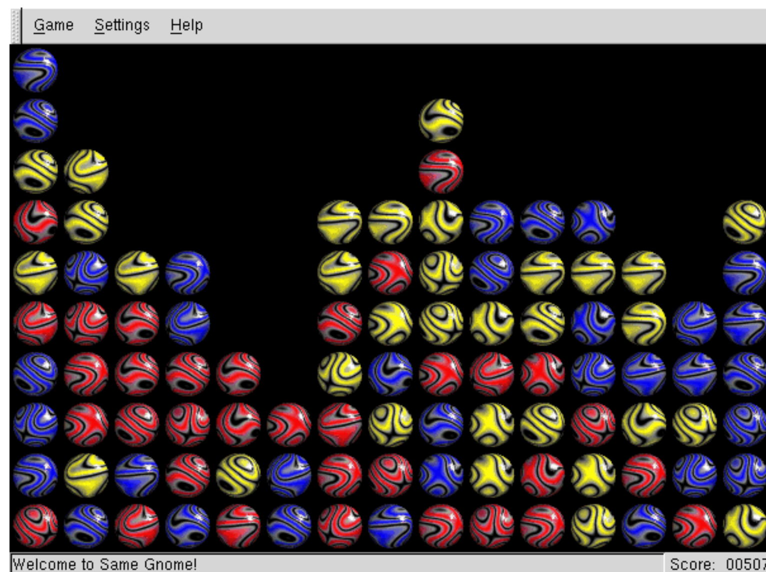

Figure 1: Screenshot of Same Gnome

The game board consists of a 15 by 10 array of squares. Each square can contain a ball of one of three colors. The board is initialized by placing a ball in each square, choosing its color at random. When the player moves the mouse cursor over a ball, that ball and all connected balls of the same color are highlighted. (Two balls are connected if they share a side, or share a side with a another ball connected to either. In other words, if two balls are connected, there is a path from one to the other that remains on the same color). Isolated balls of any color are not activated by the mouse nor deleted.

If the player clicks the mouse, the connected set of balls in removed from the board and the board is compacted. The board is compacted by first dropping balls within columns to fill in empty spaces (as if under the influence of gravity). The board is then compacted by sliding complete columns to the left to remove and empty columns. Each time the player clicks the mouse, their score it incremented by the *square* of the number of balls deleted by that click.

The game ends when there are no more groups of balls to click on.

Design and Implementation strategy

Since this is a substantial project, one should approach the design systematically rather than plunging into implementation. We recommend following closely the steps below.

In addition, we will try to supply, for those who want to use it, a framework that has mostly the standard code necessary start up this type of graphics application, as well as some (ugly) display methods to get you started.

Step 1: Write out, in outline form, in English, what is happening during the play up a game.

Step 2: Ignoring the display operations, decide on what classes are required to implement the game.

Step 3: Decide on what methods these classes need to implement. Write out the method declarations (the name of the method, its arguments, and its return type). Leave the implementations empty, but include JavaDoc comments.

Step 4: Decide on what data belongs on each object and how it will be represented. Add appropriate instance variables.

Step 5: Talk to a TA about your design.

Step 6: Implement the methods in a reasonable order to allow for thorough testing. A good strategy is:

- Implement constructors and methods to initialize the board.
- Main() and window initialization (use our version if you like).
- Methods to display the board. Start with simple methods that draw the balls as circles and highlight them by turning them white.
- Code to handle mouse event and highlight a ball when mouse passes over.
- Methods to compute and highlight the connected set of selected balls. [Note: This is probably the hairiest method to implement. My advice is to write a version that selects only the Ball that the mouse is over. The rest of the game can be then be finished. Once everything else is working, improve this method to find the connected subset.]
- Methods to delete balls and compress board.

Step 7: Once everything is working (and if you have the time and inclination), there are several ways to spiff up the game.

- Better rendering of balls. Fancier graphics or images.

- Better highlighting. For example, some animation (a good chance to experiment with threads or timers).
- Display current score and keep top ten scores.
- Whatever else you think is cool..