Eric Roberts
CS 106A

Handout #34
February 5, 2016

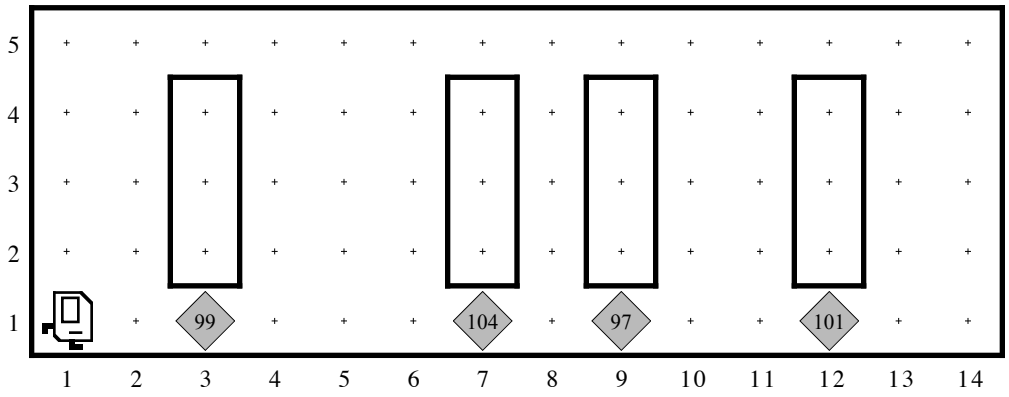
Practice Midterm Examination #2

Review session: **Monday, February 8, 6:30–8:00 P.M., NVIDIA Auditorium**
Midterm exams: **Tuesday, February 9, 9:00–11:00 A.M., CEMEX Auditorium**
Tuesday, February 9, 3:00–5:00 P.M., CEMEX Auditorium

Problem 1: Karel the Robot (10 points)

> *[We will] wield technology's wonders to raise health care's quality and lower its cost.*
>
> —President Barack Obama, January 20, 2009

Given that nothing could possibly be more representative of "technology's wonders" than Karel the Robot, it is clear that a central component of the President's health care plan must involve putting our tireless robot to work in support of patient care. As part of meeting that challenge, the Obama administration is seeking to use Karel to monitor the temperatures of patients on a hospital ward that looks something like this:

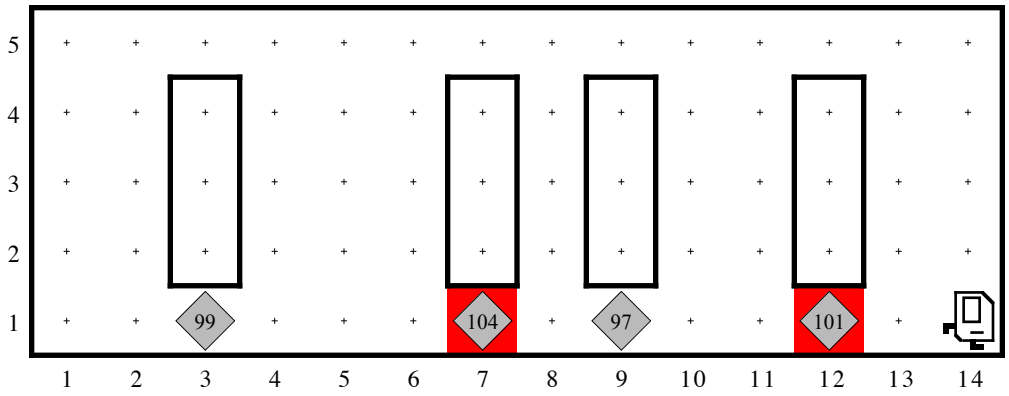


The stacks of beepers under each bed record the current temperature of the patient. Thus, the patient occupying the bed on 3rd Avenue has a temperature of 99°, which is indicated by a stack containing 99 beepers. Although 99° is slightly higher than the normal body temperature of 98.6°F, it is nothing to worry about. The patient in the 7th Avenue bed, on the other hand, has a temperature of 104°, which is dangerously high. The 97° for the patient in the 9th Avenue bed is below normal, but the 101° temperature of the patient in the bed on 12th Avenue is again high enough to cause concern.

Your job in this problem is to implement a program called **`KarelCare`** in which Karel checks each of the patients in turn and flags any temperature that is greater than 100. One way to flag an out-of-range temperature is to paint the corner red, which **`SuperKarel`** can do by calling

```
paintCorner(RED);
```

At the end of the program's operation, Karel's world should look like this:



where the 104 and 101 values are marked by a red corner.

In writing this program, you should keep the following points in mind:

1. The only part of Karel's world you need to consider is 1st Street, which is the row at the bottom of the window (the walls marking the beds are merely decorative). Karel can find where the beds are simply by looking for the stacks of beepers.
2. Karel always begins at the corner of 1st Street and 1st Avenue, facing east, with an empty beeper bag. Karel should finish on the easternmost corner of 1st Street.
3. The world can be of any length, and there can be any number of beds. The beds, moreover, can be separated by any number of open spaces, and can even be adjacent.
4. Once Karel figures out that a particular temperature needs to be flagged, it must put all the beepers back to ensure that the beeper pile continues to show the correct temperature. Given that Karel has no way of telling how many beepers are on a corner without taking them away, the number of beepers in each pile will change as the program runs. (It may help to recall that Karel starts out with an empty beeper bag.)
5. In order to meet the hospital's threshold of concern, a temperature must be strictly greater than 100. Thus, Karel should not paint the square if the patient's temperature is exactly 100, but should do so if the temperature is 101.
6. Remember that Karel has no variables, no arithmetic operations, and no **`break`** statement. Even so, Karel can, for example, repeat an operation 100 times by using a **`for`** loop.

Problem 2: Simple Java expressions, statements, and methods (10 points)

(2a) Compute the value of each of the following Java expressions. If an error occurs during any of these evaluations, write "Error" on that line and explain briefly why the error occurs.

```
6 / 5 + 5 + 8 % 3 == 7
```
_____

```
('6' - '2') + 'A'
```
_____

```
"E" - "A"
```
_____

(2b) Assume that the method **mystery** has been defined as given below:

```
private int mystery(int n) {
    while (n >= 10) {
        int k = 0;
        while (n > 0) {
            k += n % 10;
            n /= 10;
        }
        n = k;
    }
    return n;
}
```

What is the value of **mystery(1729)**?

(2c) What output is printed by the following program:

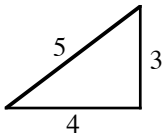
```
/*
 * File: Problem2c.java
 * --------------------
 * This program doesn't do anything useful and exists only to
 * test your understanding of parameters and string methods.
 */

import acm.program.*;

public class Problem2c extends ConsoleProgram {

   public void run() {
      String s1 = "Heart";
      String s2 = valentine("candy", s1);
      println("s1 = " + s1);
      println("s2 = " + s2);
   }

   private String valentine(String s1, String s2) {
      int num = (s1.substring(1, 2)).length();
      s1 = s2.substring(num);
      s2 = cupid(s1, s2.charAt(0));
      return (s2);
   }

   private String cupid(String s1, char ch) {
      return (s1 + Character.toLowerCase(ch));
   }

}
```

Problem 3: Simple Java programs (15 points)

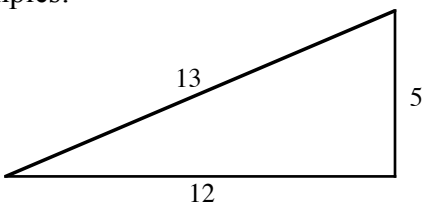
As you undoubtedly learned in school, the Pythagorean Theorem holds that the length of the hypotenuse (z) of a right triangle with sides x and y is given by the following formula:

$$x^2 + y^2 = z^2$$

As it turns out, there are an infinite number of triangles in which all three of these edge lengths are integers, including the following examples:



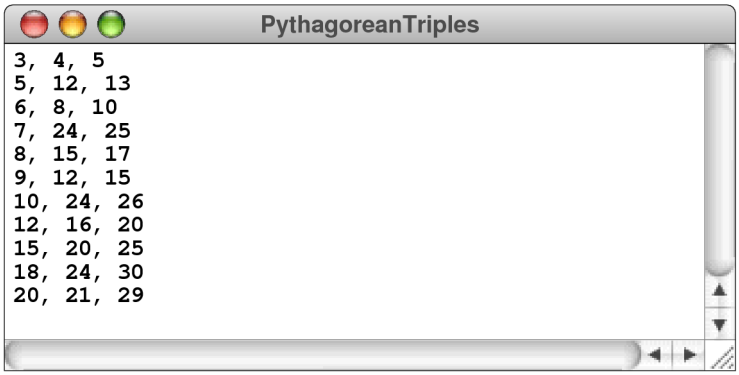
$3^2 + 4^2 = 5^2$



$5^2 + 12^2 = 13^2$

Because of this connection to the Pythagorean Theorem, any set of integers x, y, and z that meets this condition is called a ***Pythagorean triple.***

Write a Java program that prints out all Pythagorean triples in which both x and y are less than or equal to a named constant **`MAX`** and x is less than y For example, if **`MAX`** is 25, your program should generate the following sample run:



In writing this problem, you should keep the following points in mind:

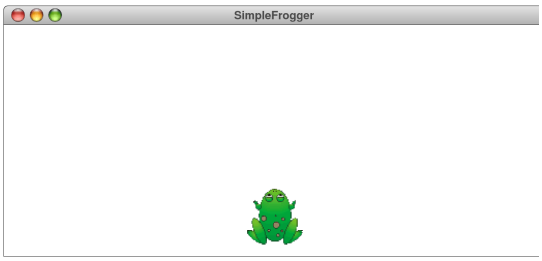
- You should not worry at all about efficiency. Trying every possible pairing of x and y and seeing whether it works is perfectly acceptable.
- The **`Math.sqrt`** method returns a **`double`**, which is only an approximation. It is possible, for example, that **`Math.sqrt(25)`** returns a **`double`** that is ever so slightly different from 5, which means that it might be come out as 4.9999999999999999 or 5.0000000000000001. The important point is that it might not be equal to an integer. Thus, to check whether an integer is a perfect square, you have to make the final test in the **`int`** domain where computation is exact. In case it comes in handy, there is a method **`GMath.round(x)`** that rounds a **`double`** to the nearest **`int`**.

Problem 4: Using the graphics and random number libraries (15 points)

> *You never see a frog so modest and straightfor'ard as he was, for all he was so gifted. And when it come to fair and square jumping on a dead level, he could get over more ground at one straddle than any animal of his breed you ever see.*
>
> —Mark Twain, "The Notorious Jumping Frog of Calaveras County," 1865

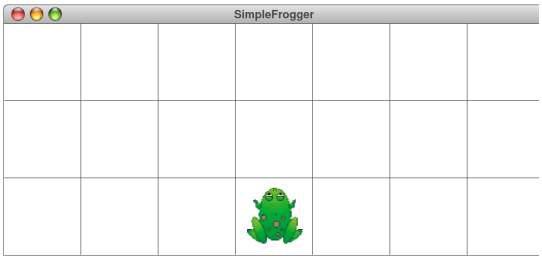
Although I didn't show it off in class, the winner in the aesthetic division of the 2010 Karel Contest played the game of Frogger, in which the object is to guide a frog across a screen filled with moving cars, floating logs, and alligators. Such a game is beyond the scope of an exam problem, but it is relatively straightforward to write the code that (1) displays the image of a frog and (2) gets the frog to jump when the user clicks the mouse.

Your first task in this problem is to place the frog at the bottom of the graphics window, as shown on the right. The frog itself is the easy part because all you need to do is create a **GImage** object with the appropriate picture, as follows:




```
GImage frog = new GImage("frog.gif");
```

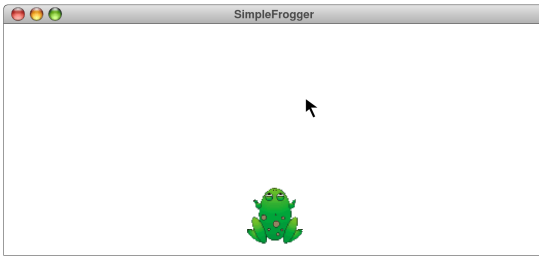
The harder part is getting the image in the appropriate place in the bottom of the window. In Frogger, the frog image cannot be just anywhere on the screen but must instead occupy a position in an imaginary grid such as the one shown on the right. The size of the grid is controlled by three named constants, which have the following values for this grid:




```
public static final int SQSIZE = 75;
public static final int NCOLS = 7;
public static final int NROWS = 3;
```

The **SQSIZE** constant indicates that each of the squares in the grid is 75 pixels in each dimension and the other two parameters give the width and height of the grid in terms of the number of squares. Remember that the squares shown in the most recent diagram do not actually exist but simply define the legal positions for the frog. In the initial position, the frog must be in the center square along the bottom row. You may assume **NCOLS** is odd so that there is a center square, and you may also assume that **APPLICATION_WIDTH** and **APPLICATION_HEIGHT** have been set so the **NCOLS** x **NROWS** squares fill the window.

The second part of the problem is getting the frog to jump when the user clicks the mouse. The goal is to get the frog to jump one square in the direction that moves it closest to the mouse. For example, if you click the mouse at the location shown in the diagram at the right, the frog should move **SQSIZE** pixels upward so that it occupies the center square in the grid. If the user then clicked the mouse at the left edge of the screen, the frog should jump **SQSIZE** pixels to the left. The frog, however, should never jump outside the window.



Problem 5: Strings and characters (10 points)

How do I love thee? Let me count the ways.
—Elizabeth Barrett Browning, *Sonnet 43,* 1850

Computers are, of course, very good at counting things. Write a method

```
private int countLove(String str)
```

that takes a string as its argument and returns the number of times the word *love* appears in that string, ignoring differences in case, but making sure that *love* is not just part of a longer word like *clover, glove, pullover,* or *slovenly.* For example, if you were to call

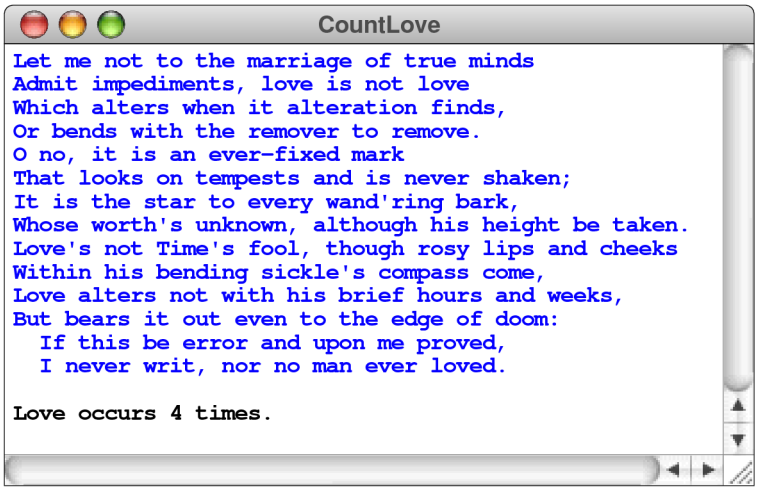
```
countLove("Love in the clover.")
```

your method should return 1. The word `Love` counts as a match because your method should ignore the fact that `Love` starts with an uppercase `L`. The word `clover` doesn't match because the letters `love` are merely part of a larger word.

Although you have enough information to write the method already, some people find it helpful to see the method used in context. The following `run` method, for example, counts all the occurrences of `love` in a paragraph of text ending with a blank line:

```
public void run() {
   int count = 0;
   while (true) {
      String line = readLine();
      if (line.length() == 0) break;
      count += countLove(line);
   }
   println("Love occurs " + count + " times.");
}
```

This program might produce the following sample run, which uses Shakespeare's Sonnet 116 as its input text:



There are two matches in the second line, one in the ninth, and one in the eleventh. Note that `Love's` at the beginning of the ninth line counts because the apostrophe is not a letter, but that `loved` at the very end of the sonnet does not.