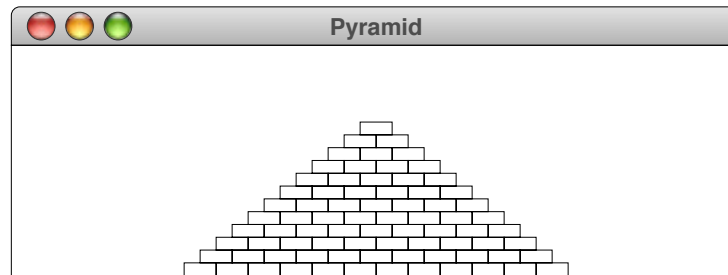# Assignment #2—Simple Java Programs

Due: Friday, January 22

Your job in this assignment is to write programs to solve each of these problems.

1. Write a **GraphicsProgram** subclass that draws a pyramid consisting of bricks arranged in horizontal rows, so that the number of bricks in each row decreases by one as you move up the pyramid, as shown in the following sample run:


   The pyramid should be centered at the bottom of the window and should use constants for the following parameters:

BRICK_WIDTH	The width of each brick (30 pixels)
BRICK_HEIGHT	The height of each brick (12 pixels)
BRICKS_IN_BASE	The number of bricks in the base (12)

   The numbers in parentheses show the values for this diagram, but you must be able to change those values in your program.

2. Write a **GraphicsProgram** subclass that draws a rainbow that looks like this:
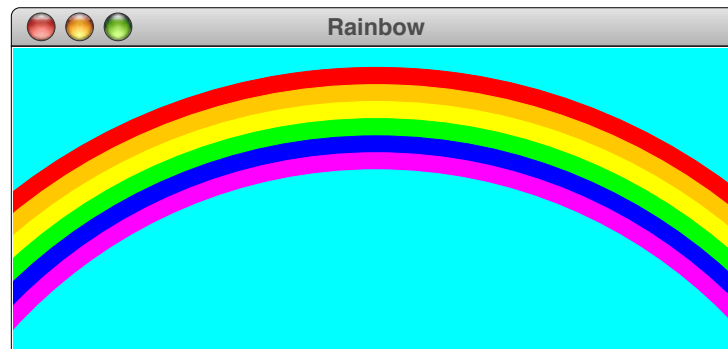
   The colors of the stripes are clear in the web version of the picture, but are hard to see in the black-and-white handout. Starting at the top, the six arcs are red, orange, yellow, green, blue, and magenta, respectively; cyan makes a lovely color for the sky.

   At first glance, it might seem as if you need to draw arcs on the screen, even though you won't actually learn about the **GArc** class until Chapter 8. As it turns out, that class doesn't really help much. The program that produced the diagram shown at the bottom of the previous page uses only circles, although seeing how this is possible forces you to think outside the box—in a literal rather than a figurative sense. The

common center for each circle is some distance below the bottom of the window, and the diameters of the circles are wider than the screen. The **GraphicsProgram** shows only the part of the figure that actually appears in the window. This process of reducing a picture to the visible area is called **clipping.**

Rather than specify the exact dimensions of each circle, play around with the sizes and positioning of the circles until you get something that matches your aesthetic sensibilities. The only things we'll be concerned about are:

- The top of the arc should not be off the screen.
- Each of the arcs in the rainbow should get clipped along the sides of the window, and not along the bottom.

3. Write a **GraphicsProgram** subclass that draws a partial diagram of the **acm.graphics** class hierarchy, as follows:
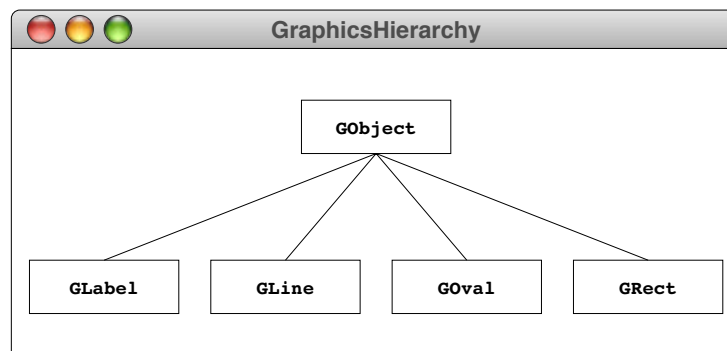
The only classes you need to create this picture are **GRect**, **GLabel**, and **GLine**. The tricky part is specifying the coordinates so that the different elements of the picture are aligned properly. The aspects of the alignment for which you are responsible are:

- The width and height of the class boxes should be specified as named constants so that they are easy to change.
- The labels should be centered in their boxes. You can find the width of a label by calling **label.getWidth()** and the height it extends above the baseline by calling **label.getAscent()**. If you want to center a label, you need to shift its origin by half of these distances in each direction.
- The connecting lines should start and end at the center of the appropriate edge of the box.
- The entire figure should be centered in the window.

4. In high-school algebra, you learned that the standard quadratic equation

$$a x^2 + b x + c = 0$$

has two solutions given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

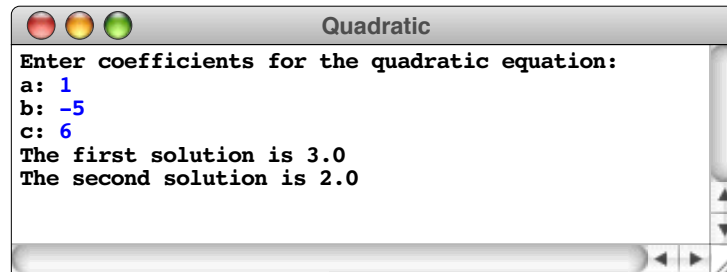The first solution is obtained by using + in place of ±; the second is obtained by using – in place of ±. Most of this expression contains simple operators covered in Chapter 3. The one piece that's missing is taking square roots, which you can do by

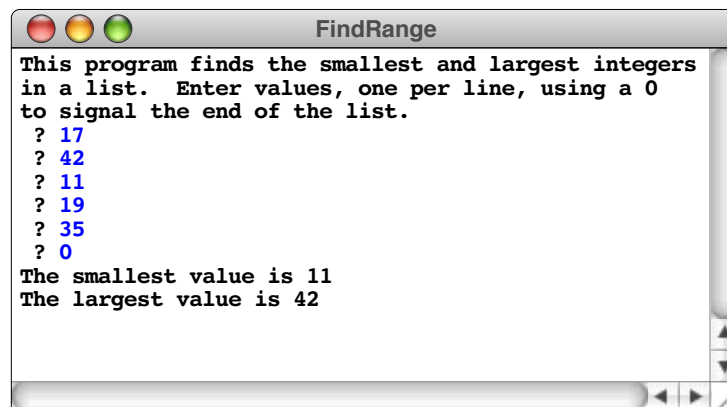calling the standard function `Math.sqrt`. For example, the statement

```
double y = Math.sqrt(x);
```

sets `y` to the square root of `x`.

Write a `ConsoleProgram` that accepts values for `a`, `b`, and `c`, and then calculates the two solutions (which may both be the same). If the quantity under the square root sign is negative, the equation has no real solutions, and your program should display a message to that effect. You may assume that the value for `a` is nonzero. Your program should be able to duplicate the following sample run:

```
● ● ●                      Quadratic
Enter coefficients for the quadratic equation:
a: 1
b: −5
c: 6
The first solution is 3.0
The second solution is 2.0
```

5. Write a `ConsoleProgram` that reads in a list of integers, one per line, until a sentinel value of 0 (which you should be able to change easily to some other value). When the sentinel is read, your program should display the smallest and largest values in the list, as illustrated in this sample run:

```
● ● ●                      FindRange
This program finds the smallest and largest integers
in a list.  Enter values, one per line, using a 0
to signal the end of the list.
 ? 17
 ? 42
 ? 11
 ? 19
 ? 35
 ? 0
The smallest value is 11
The largest value is 42
```

Your program should handle the following special cases:

*   If the user enters only one value before the sentinel, the program should report that value as both the largest and smallest.

*   If the user enters the sentinel on the very first input line, then no values have been entered, and your program should display a message to that effect.

6. In my philosophical excursion on the topics of holism and reductionism, I referred to Douglas Hofstadter's Pulitzer-prize-winning book *Gödel, Escher, Bach*. Hofstadter's book contains many interesting mathematical puzzles, many of which can be expressed in the form of computer programs. Of these, most require programming skills well beyond the second week of CS 106A. In Chapter XII, Hofstadter mentions a wonderful problem that is well within the scope of the control statements from Chapter 4. The problem can be expressed as follows:
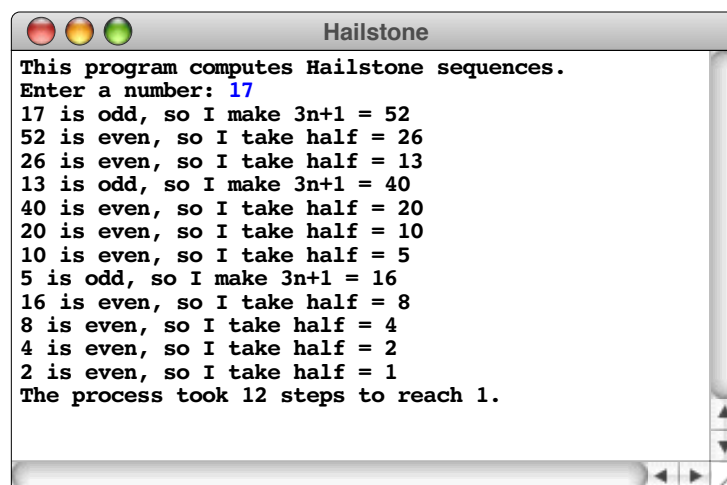
Pick some positive integer and call it n.
If n is even, divide it by two.
If n is odd, multiply it by three and add one.
Continue this process until n is equal to one.

On page 401 of the Vintage edition, Hofstadter illustrates this process with the following example, starting with the number 15:

15	is odd, so I make $3n+1$:	46
46	is even, so I take half:	23
23	is odd, so I make $3n+1$:	70
70	is even, so I take half:	35
35	is odd, so I make $3n+1$:	106
106	is even, so I take half:	53
53	is odd, so I make $3n+1$:	160
160	is even, so I take half:	80
80	is even, so I take half:	40
40	is even, so I take half:	20
20	is even, so I take half:	10
10	is even, so I take half:	5
5	is odd, so I make $3n+1$:	16
16	is even, so I take half:	8
8	is even, so I take half:	4
4	is even, so I take half:	2
2	is even, so I take half:	1

As you can see from this example, the numbers go up and down, but eventually—at least for all numbers that have ever been tried—comes down to end in 1. In some respects, this process is reminiscent of the formation of hailstones, which get carried upward by the winds over and over again before they finally descend to the ground. Because of this analogy, this sequence of numbers is usually called the **Hailstone sequence,** although it goes by many other names as well.

Write a **ConsoleProgram** that reads in a number from the user and then displays the Hailstone sequence for that number, just as in Hofstadter's book, followed by a line showing the number of steps taken to reach 1. For example, your program should be able to produce a sample run that looks like this:

```
●●●                      Hailstone
This program computes Hailstone sequences.
Enter a number: 17
17 is odd, so I make 3n+1 = 52
52 is even, so I take half = 26
26 is even, so I take half = 13
13 is odd, so I make 3n+1 = 40
40 is even, so I take half = 20
20 is even, so I take half = 10
10 is even, so I take half = 5
5 is odd, so I make 3n+1 = 16
16 is even, so I take half = 8
8 is even, so I take half = 4
4 is even, so I take half = 2
2 is even, so I take half = 1
The process took 12 steps to reach 1.
```

The fascinating thing about this problem is that no one has yet been able to prove that it always stops. The number of steps in the process can certainly get very large. How many steps, for example, does your program take when n is 27?