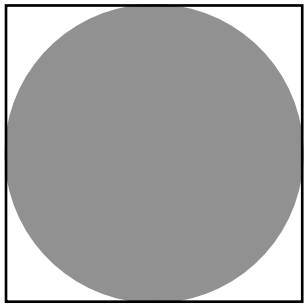
The Demo Programs in Folder Assignment3

Pseudorandom Numbers

The Java class **`PiApproximation`** uses the **`RandomGenerator`** class in **`acm.util`** to solve the following exercise from our text:

> Although it is often easiest to think of random numbers in the context of games of chance, they have other, more practical uses in computer science and mathematics. For example, you can use random numbers to generate a rough approximation of the constant π by writing a simple program that simulates a dart board. Imagine that you have a dart board hanging on your wall. It consists of a circle painted on a square backdrop, as in the following diagram:



> If you throw darts at this board in a random fashion, some will fall inside the circle, but some will fall outside. If the tosses are truly random, the ratio of the number of darts that land inside the circle to the total number of darts hitting the square should be roughly equal to the ratio between the two areas. The ratio of the areas is independent of the actual size of the dart board, as illustrated by the following formula:

$$\frac{\text{darts falling inside the circle}}{\text{darts falling inside the square}} \cong \frac{\text{area of the circle}}{\text{area of the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

> To simulate this process in a program, imagine that the dart board is drawn in the standard Cartesian coordinate plane you learned about in high school. The process of throwing a dart randomly at the square can be modeled by generating two random numbers, x and y, each of which lies between –1 and 1. This (x, y) point always lies somewhere inside the square. The point (x, y) lies inside the circle if

$$\sqrt{x^2 + y^2} < 1$$

> This condition, however, can be simplified considerably by squaring each side of the inequality, which gives the following more efficient test:

$$x^2 + y^2 < 1$$

> If you perform this simulation many times and compute the fraction of darts that fall within the circle, the result will be somewhere in the neighborhood of π/4.

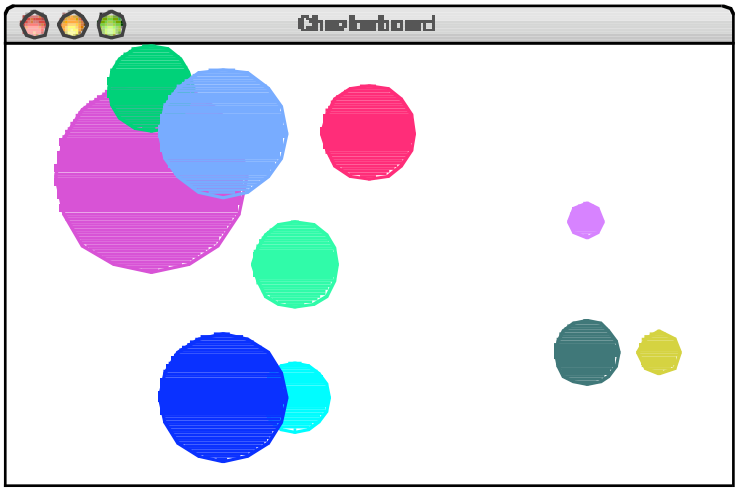
Write a program that simulates throwing 10,000 darts and then uses the simulation technique described in this exercise to generate and display an approximate value of π. Don't worry if your answer is correct only in the first few digits. The strategy used in this problem is not particularly accurate, even though it occasionally proves useful as a technique for making rough approximations. In mathematics, this technique is called *Monte Carlo integration*, after the gambling center that is the capital city of Monaco.

Pseudorandomly Changing Colors

The Java class **`ColorChangingSquare`** similarly creates a colored square and then changes its color randomly every second.

Pseudorandom Circles

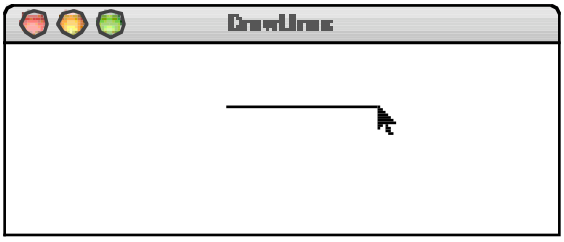
`RandomCircles` is a subclass of **`GraphicsProgram`** that draws a set of ten circles with different sizes, positions, and colors. Each circle has a randomly chosen color, a randomly chosen radius between 5 and 50 pixels, and a randomly chosen position on the canvas, subject to the condition that the entire circle must fit inside the canvas without extending past the edge. The following sample run shows one possible outcome:



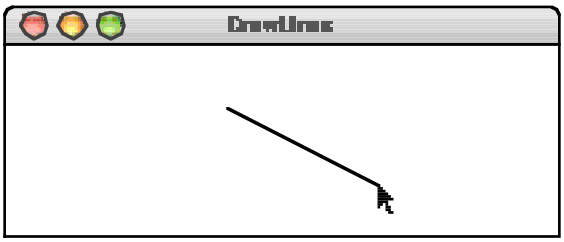
On some runs of this program you might not see ten circles. Why?

Drawing Lines in Response to the Mouse

`RubberBanding` is a subclass of **`GraphicsProgram`** that allows the user to draw lines on the canvas. Pressing the mouse button sets the starting point for the line. Dragging the mouse moves the other endpoint around as the drag proceeds. Releasing the mouse fixes the line in its current position and gets ready to start a new line. For example, suppose that you press the mouse button somewhere on the screen and then drag it rightward an inch, holding the button down. What you'd like to see is the following picture:



If you then move the mouse downward without releasing the button, the displayed line will track the mouse, so that you might see the following picture:



Because the original point and the mouse position appear to be joined by some elastic string, this technique is called **rubber-banding.**

Although this program may seem quite powerful (and indeed it allows you to draw any figure on the screen that can be drawn with lines), it is also simple to implement. The entire program requires fewer than 20 lines of code.

Using the `GArc` and `GPolygon` Graphics Classes

The Java classes **`YinYang`** and **`StopSign`** demonstrate two of the "shape classes" in the **`acm.graphics`** package. These classes are described in the text. Most of them are relatively easy to use; the ones that seem to cause difficulty are **`GArc`** and **`GPolygon`**. For these classes, the important things to remember are:

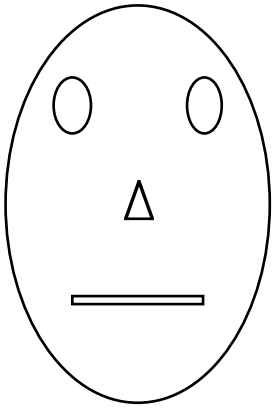
- The **`GArc`** constructor takes the boundary of the rectangle that contains the ellipse in which the arc appears, along with two parameters—**`start`** and **`sweep`**—that give the angle at which the arc begins (measured in degrees counterclockwise from the +*x* axis) and the number of degrees through which the arc extends, respectively.
- The coordinates of the vertices of a **`GPolygon`** are measured relative to a point defined to be the **origin** of that polygon. The origin need not actually be a vertex; in many cases, for example, the most convenient origin for the polygon is its center. When you add a **`GPolygon`** to the canvas, the entire figure is then shifted to the location of the entire polygon and drawn relative to that point.

The **`YinYang`** class uses the **`GArc`** class to draw the following Yin-Yang symbol:

The **StopSign** class uses the **GPolygon** class to draw a picture of a stop sign, as follows:

Graphical Object Decomposition

The Java classes **DrawFace1** through **DrawFace3** solve a graphics problem while focusing on the idea of decomposition. The problem is to generate an abstract picture of a face that looks like this:



The parameters that control the size of the face are **width** and **height**, which can be chosen by the programmer and which can be different for different faces. The dimensions of the eyes, nose, and mouth are all calculated from the dimensions of the face as a whole using the following constant definitions, each of which is multiplied by the **width** or the **height** parameter, as appropriate:

```
private static final double EYE_WIDTH    = 0.15;
private static final double EYE_HEIGHT   = 0.15;
private static final double NOSE_WIDTH   = 0.15;
private static final double NOSE_HEIGHT  = 0.10;
private static final double MOUTH_WIDTH  = 0.50;
private static final double MOUTH_HEIGHT = 0.03;
```

The Java classes show three different solutions to the problem:

1. A decomposition strategy similar to the style used in Karel in which the face is assembled on the canvas by calling a procedure

```
void drawFace(double x, double y, double width, double height)
```

where **x** and **y** are the *center* of the face rather than its upper-left corner. The code for **drawFace** then uses stepwise refinement to break the problem up into similar calls to **drawEye**, **drawNose**, and **drawMouth**.

2. A parallel decomposition strategy in the object domain where the face is a **GFace** object, which is a subclass of **GCompound**. That compound is initialized by the constructor to contain two **GEye**s, a **GNose**, and a **GMouth**.

3. A possibly amusing example in which we extend the **GEye** implementation to include a pupil that tracks mouse motions.