Eric Roberts
CS 106A

Handout #28
April 25, 2012

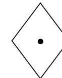
Graphical Structures

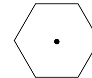
Graphical Structures

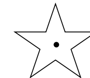
Eric Roberts
CS 106A
April 25, 2012

The GPolygon Class

- The `GPolygon` class is used to represent graphical objects bound by line segments. In mathematics, such figures are called ***polygons*** and consist of a set of ***vertices*** connected by ***edges***. The following figures are examples of polygons:


diamond


regular hexagon


five-pointed star

- Unlike the other shape classes, that location of a polygon is not fixed at the upper left corner. What you do instead is pick a ***reference point*** that is convenient for that particular shape and then position the vertices relative to that reference point.
- The most convenient reference point is often the geometric center of the object.

Constructing a GPolygon Object

- The `GPolygon` constructor creates an empty polygon. Once you have the empty polygon, you then add each vertex to the polygon, one at a time, until the entire polygon is complete.
- The most straightforward way to create a `GPolygon` is to use the method `addVertex(`*x*`, `*y*`)`, which adds a new vertex to the polygon. The *x* and *y* values are measured relative to the reference point for the polygon rather than the origin.
- When you start to build up the polygon, it always makes sense to use `addVertex(`*x*`, `*y*`)` to add the first vertex. Once you have added the first vertex, you can call any of the following methods to add the remaining ones:
  - `addVertex(`*x*`, `*y*`)` adds a new vertex relative to the reference point
  - `addEdge(`*dx*`, `*dy*`)` adds a new vertex relative to the preceding one
  - `addPolarEdge(`*r*`, `*theta*`)` adds a new vertex using polar coordinates

  Each of these strategies is illustrated in a subsequent slide.

Using addVertex and addEdge

- The `addVertex` and `addEdge` methods each add one new vertex to a `GPolygon` object. The only difference is in how you specify the coordinates. The `addVertex` method uses coordinates relative to the reference point, while the `addEdge` method indicates displacements from the previous vertex.
- Your decision about which of these methods to use is based on what information you have readily at hand. If you can easily calculate the coordinates of the vertices, `addVertex` is probably the right choice. If, however, it is much easier to describe each edge, `addEdge` is probably a better strategy.
- No matter which of these methods you use, the `GPolygon` class closes the polygon before displaying it by adding an edge from the last vertex back to the first one, if necessary.
- The next two slides show how to construct a diamond-shaped polygon using the `addVertex` and the `addEdge` strategies.

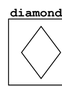
Drawing a Diamond (addVertex)

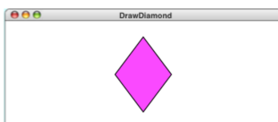
The following program draws a diamond using `addVertex`:

```
public void run() {
```

```
private GPolygon createDiamond(double width, double height) {
   GPolygon diamond = new GPolygon();
   diamond.addVertex(-width / 2, 0);
   diamond.addVertex(0, -height / 2);
   diamond.addVertex(width / 2, 0);
   diamond.addVertex(0, height / 2);
   return diamond;
}
```

diamond





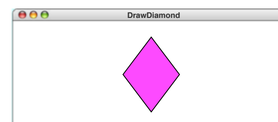
Drawing a Diamond (addEdge)

This program draws the same diamond using `addEdge`:

```
public void run() {
```

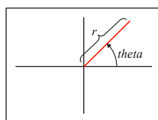
```
private GPolygon createDiamond(double width, double height) {
   GPolygon diamond = new GPolygon();
   diamond.addVertex(-width / 2, 0);
   diamond.addEdge(width / 2, -height / 2);
   diamond.addEdge(width / 2, height / 2);
   diamond.addEdge(-width / 2, height / 2);
   diamond.addEdge(-width / 2, -height / 2);
   return diamond;
}
```

diamond



Using `addPolarEdge`

- In many cases, you can determine the length and direction of a polygon edge more easily than you can compute its *x* and *y* coordinates. In such situations, the best strategy for building up the polygon outline is to call `addPolarEdge(`*r*`,` *theta*`)`, which adds an edge of length *r* at an angle that extends *theta* degrees counterclockwise from the +*x* axis, as illustrated by the following diagram:



- The name of the method reflects the fact that `addPolarEdge` uses what mathematicians call ***polar coordinates***.

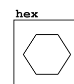
Drawing a Hexagon

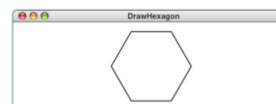
This program draws a regular hexagon using `addPolarEdge`:

```
public void run() {
```

```
private GPolygon createHexagon(double side) {
   GPolygon hex = new GPolygon();
   hex.addVertex(-side, 0);
   int angle = 60;
   for (int i = 0; i < 6; i++) {
      hex.addPolarEdge(side, angle);
      angle -= 60;
   }
   return hex;
}
```

hex





skip simulation

Defining `GPolygon` Subclasses

- The `GPolygon` class can also serve as the superclass for new types of graphical objects. For example, instead of calling a method like the `createHexagon` method from the preceding slide, you could also define a `GHexagon` class like this:

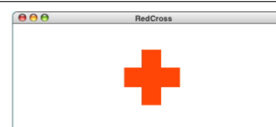
```
public class GHexagon extends GPolygon {
   public GHexagon(double side) {
      addVertex(-side, 0);
      int angle = 60;
      for (int i = 0; i < 6; i++) {
         addPolarEdge(side, angle);
         angle -= 60;
      }
   }
}
```

- The `addVertex` and `addPolarEdge` calls in the `GHexagon` constructor operate on the object being created, which is set to an empty `GPolygon` by the superclass constructor.

Exercise: Using the `GPolygon` Class

Define a class `GCross` that represents a cross-shaped figure. The constructor should take a single parameter `size` that indicates both the width and height of the cross. Your definition should make it possible to execute the following program to produce the diagram at the bottom of the slide:

```
public void run() {
   GCross cross = new GCross(100);
   cross.setFilled(true);
   cross.setColor(Color.RED);
   add(cross, getWidth() / 2, getHeight() / 2);
}
```



Creating Compound Objects

- The `GCompound` class in the `acm.graphics` package makes it possible to combine several graphical objects so that the resulting structure behaves as a single `GObject`.
- The easiest way to think about the `GCompound` class is as a combination of a `GCanvas` and a `GObject`. A `GCompound` is like a `GCanvas` in that you can add objects to it, but it is also like a `GObject` in that you can add it to a canvas.
- As was true in the case of the `GPolygon` class, a `GCompound` object has its own coordinate system that is expressed relative to a ***reference point***. When you add new objects to the `GCompound`, you use the local coordinate system based on the reference point. When you add the `GCompound` to the canvas as a whole, all you have to do is set the location of the reference point; the individual components will automatically appear in the right locations relative to that point.

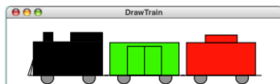
Exercise: Labeled Rectangles

Define a class `GLabeledRect` that consists of an outlined rectangle with a label centered inside. Your class should include constructors that are similar to those for `GRect` but include an extra argument for the label. It should also export `setLabel`, `getLabel`, and `setFont` methods. The following `run` method illustrates the use of the class:

```
public void run() {
   GLabeledRect rect = new GLabeledRect(100, 50, "hello");
   rect.setFont("SansSerif-18");
   add(rect, 150, 50);
}
```

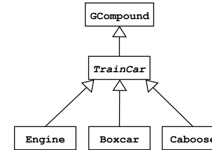
Graphical Object Decomposition

- The most important advantage of using the `GCompound` class is that doing so makes it possible to apply the strategy of decomposition in the domain of graphical objects. Just as you use stepwise refinement to break a problem down into smaller and smaller pieces, you can use it to decompose a graphical display into successively simpler pieces.
- The text illustrates this technique by returning to the example of train cars from Chapter 5, where the goal is to produce the picture at the bottom of this slide.
- In Chapter 5, the decomposition strategy led to a hierarchy of methods. The goal now is to produce a hierarchy of classes.



The `TrainCar` Hierarchy

- The critical insight in designing an object-oriented solution to the train problem is that the cars form a hierarchy in which the individual classes `Engine`, `Boxcar`, and `Caboose` are all subclasses of a more general class called `TrainCar`:



- The `TrainCar` class itself is a `GCompound`, which means that it is a graphical object. The constructor at the `TrainCar` level adds the common elements, and the constructors for the individual subclasses adds any remaining details.

The `TrainCar` Class

```
import acm.graphics.*;
import java.awt.*;

/** This abstract class defines what is common to all train cars */
public abstract class TrainCar extends GCompound {

/**
 * Creates the frame of the car using the specified color.
 * @param color The color of the new train car
 */
   public TrainCar(Color color) {
      double xLeft = CONNECTOR;
      double yBase = -CAR_BASELINE;
      add(new GLine(0, yBase, CAR_WIDTH + 2 * CONNECTOR, yBase));
      addWheel(xLeft + WHEEL_INSET, -WHEEL_RADIUS);
      addWheel(xLeft + CAR_WIDTH - WHEEL_INSET, -WHEEL_RADIUS);
      double yTop = yBase - CAR_HEIGHT;
      GRect r = new GRect(xLeft, yTop, CAR_WIDTH, CAR_HEIGHT);
      r.setFilled(true);
      r.setFillColor(color);
      add(r);
   }
```

The `TrainCar` Class

```
/* Adds a wheel centered at (x, y) */
   private void addWheel(double x, double y) {
      GOval wheel = new GOval(x - WHEEL_RADIUS, y - WHEEL_RADIUS,
                              2 * WHEEL_RADIUS, 2 * WHEEL_RADIUS);
      wheel.setFilled(true);
      wheel.setFillColor(Color.GRAY);
      add(wheel);
   }

/* Private constants */
   protected static final double CAR_WIDTH = 75;
   protected static final double CAR_HEIGHT = 36;
   protected static final double CAR_BASELINE = 10;
   protected static final double CONNECTOR = 6;
   protected static final double WHEEL_RADIUS = 8;
   protected static final double WHEEL_INSET = 16;

}
```

The `Boxcar` Class

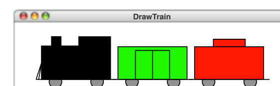
```
/**
 * This class represents a boxcar.  Like all TrainCar subclasses,
 * a Boxcar is a graphical object that you can add to a GCanvas.
 */
public class Boxcar extends TrainCar {

/**
 * Creates a new boxcar with the specified color.
 * @param color The color of the new boxcar
 */
   public Boxcar(Color color) {
      super(color);
      double xRightDoor = CONNECTOR + CAR_WIDTH / 2;
      double xLeftDoor = xRightDoor - DOOR_WIDTH;
      double yDoor = -CAR_BASELINE - DOOR_HEIGHT;
      add(new GRect(xLeftDoor, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
      add(new GRect(xRightDoor, yDoor, DOOR_WIDTH, DOOR_HEIGHT));
   }

/* Dimensions of the door panels on the boxcar */
   private static final double DOOR_WIDTH = 18;
   private static final double DOOR_HEIGHT = 32;

}
```

Nesting Compound Objects

- Given that a `GCompound` is also a `GObject`, you can add a `GCompound` to another `GCompound`.
- The `Train` class on the next slide illustrates this technique by defining an entire train as a compound to which you can append new cars. You can create a three-car train like this:

```
Train train = new Train();
train.append(new Engine());
train.append(new Boxcar(Color.GREEN));
train.append(new Caboose());
```

- One tremendous advantage of making the train a single object is that you can then animate the train as a whole.



The `Train` Class

```
import acm.graphics.*;

/** This class defines a GCompound that represents a train. */
public class Train extends GCompound {

/**
 * Creates a new train that contains no cars.  Clients can add
 * cars at the end by calling append.
 */
   public Train() {
      /* No operations necessary */
   }

/**
 * Adds a new car to the end of the train.
 * @param car The new train car
 */
   public void append(TrainCar car) {
      double width = getWidth();
      double x = (width == 0) ? 0 : width - TrainCar.CONNECTOR;
      add(car, x, 0);
   }
}
```