

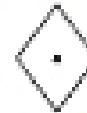
## Graphical Structures Slides

### Graphical Structures

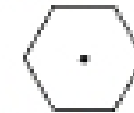
Eric Roberts  
CS 106A  
January 29, 2010

### 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.

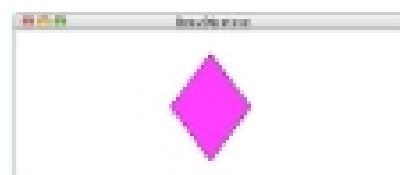
### Using addVertex and addEdge

- 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`:

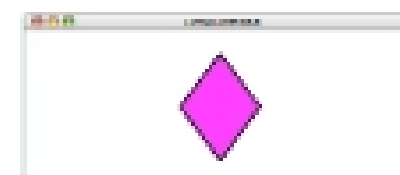
```
public void run() {
    private GPolygon makeDiamond(double width, double height) {
        GPolygon diamond = new GPolygon();
        diamond.addVertex(-width / 2, 0);
        diamond.addVertex(0, -height / 2);
        diamond.addVertex(width / 2, 0);
        diamond.addVertex(0, height / 2);
 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



### Drawing a Diamond (addEdge)

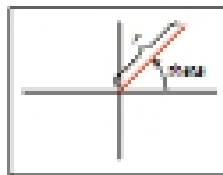
This program draws the same diamond using `addEdge`:

```
public void run() {
    private GPolygon makeDiamond(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;
    }
}
```



## 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:

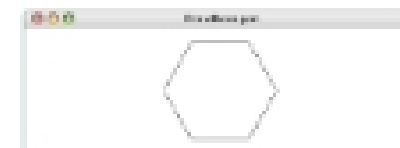
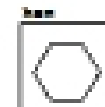


- The name of the method reflects the fact that `addPolarEdge` uses what mathematicians call **polar coordinates**.

## 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);
        hex.angle = 60;
        for (int k = 0; k < 6; k++) {
            hex.addPolarEdge(side, angle);
            angle += 60;
        }
        return hex;
    }
}
```



Copyright © 2004

## 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);
        hex.angle = 60;
        for (int k = 0; k < 6; k++) {
            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.setFill(Color.RED);
    add(cross, getWidth() / 2, getHeight() / 2);
}
```

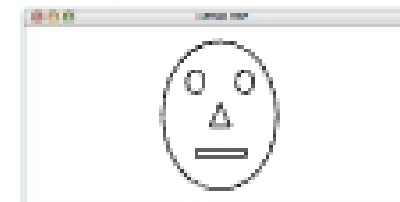


## Creating Compound Objects

- 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 appear in the right locations relative to that point.

## Creating a Face Object

- The first example of the `GCompound` class is the `drawFace` program, which is illustrated at the bottom of this slide.
- The figure consists of a `G oval` for the face and each of the eyes, a `GPolygon` for the nose, and a `GRect` for the mouth. These objects, however, are not added directly to the canvas but to a `GCompound` that represents the face as a whole.
- This primary advantage of using the `GCompound` strategy is that doing so allows you to manipulate the face as a unit.



## The GFace Class

```
import java.awt.*;

/** Defines a compound GFace class */
public class GFace extends GCompound {

    /** Creates a new GFace object with the specified dimensions */
    public GFace(double width, double height) {
        head = new COval(width, height);
        leftEye = new COval(EYE_WIDTH * width, EYE_HEIGHT * height);
        rightEye = new COval(EYE_WIDTH * width, EYE_HEIGHT * height);
        nose = new GRect(NOSE_WIDTH * width, NOSE_HEIGHT * height);
        mouth = new GRect(MOUTH_WIDTH * width, MOUTH_HEIGHT * height);
        add(head, 0, 0);
        add(leftEye, 0.25 * width - EYE_WIDTH * width / 2,
            0.25 * height - EYE_HEIGHT * height / 2);
        add(rightEye, 0.75 * width - EYE_WIDTH * width / 2,
            0.25 * height - EYE_HEIGHT * height / 2);
        add(nose, 0.50 * width, 0.50 * height);
        add(mouth, 0.50 * width - MOUTH_WIDTH * width / 2,
            0.75 * height - MOUTH_HEIGHT * height / 2);
    }
}
```

## The GFace Class

```
/** Creates a triangle for the nose */
private CPolygon nose(double width, double height) {
    CPolygon poly = new CPolygon();
    poly.addVertex(0, -height / 2);
    poly.addVertex(width / 2, height / 2);
    poly.addVertex(-width / 2, height / 2);
    return poly;
}

/** Constants specifying feature size as a fraction of the head size */
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 = 0.15;
private static final double MOUTH_WIDTH = 0.20;
private static final double MOUTH_HEIGHT = 0.20;

/** Private instance variables */
private COval head;
private COval leftEye, rightEye;
private CPolygon nose;
private GRect mouth;
}
```

## Exercise: Labeled Rectangles

Define a class `GLabelRect` 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() {
    GLabelRect rect = new GLabelRect(100, 50, "hello");
    rect.setFont("SansSerif-18");
    add(rect, 100, 50);
}
}
```



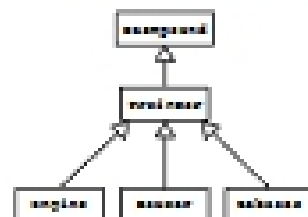
## Graphical Object Decomposition

- 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 java.awt.*;
import java.awt.geom.*;

/** 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 car body
     */
    public TrainCar(Color color) {
        double xLoc = CONNECTION;
        double yLoc = -CAR_BASELINE;
        add(new GRect(0, yLoc, CAR_WIDTH + 2 * CONNECTION, yLoc));
        addHead(xLoc + WHEEL_OFFSET, -WHEEL_RADIUS);
        addHead(xLoc + CAR_WIDTH - WHEEL_OFFSET, -WHEEL_RADIUS);
        double yTop = yLoc + CAR_HEIGHT;
        GRect r = new GRect(xLoc, yTop, CAR_WIDTH, CAR_HEIGHT);
        r.setFill(color);
        add(r);
    }
}
```