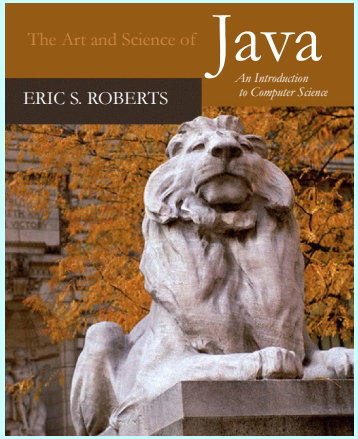
CHAPTER 9

Object-oriented Graphics

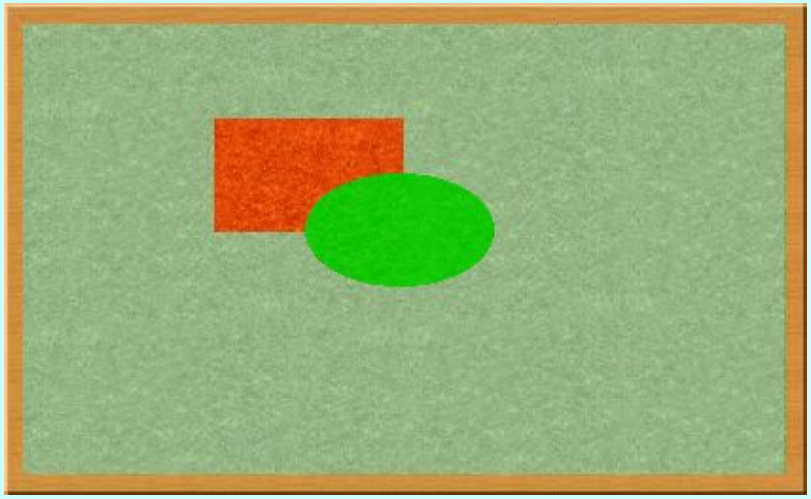
Yea, from the table of my memory
I'll wipe away all trivial fond records.

—William Shakespeare, *Hamlet,* c. 1600



The `acm.graphics` Model

- The **`acm.graphics`** package uses a **collage** model in which you create an image by adding various objects to a canvas.
- A collage is similar to a child's felt board that serves as a backdrop for colored shapes that stick to the felt surface. As an example, the following diagram illustrates the process of adding a red rectangle and a green oval to a felt board:



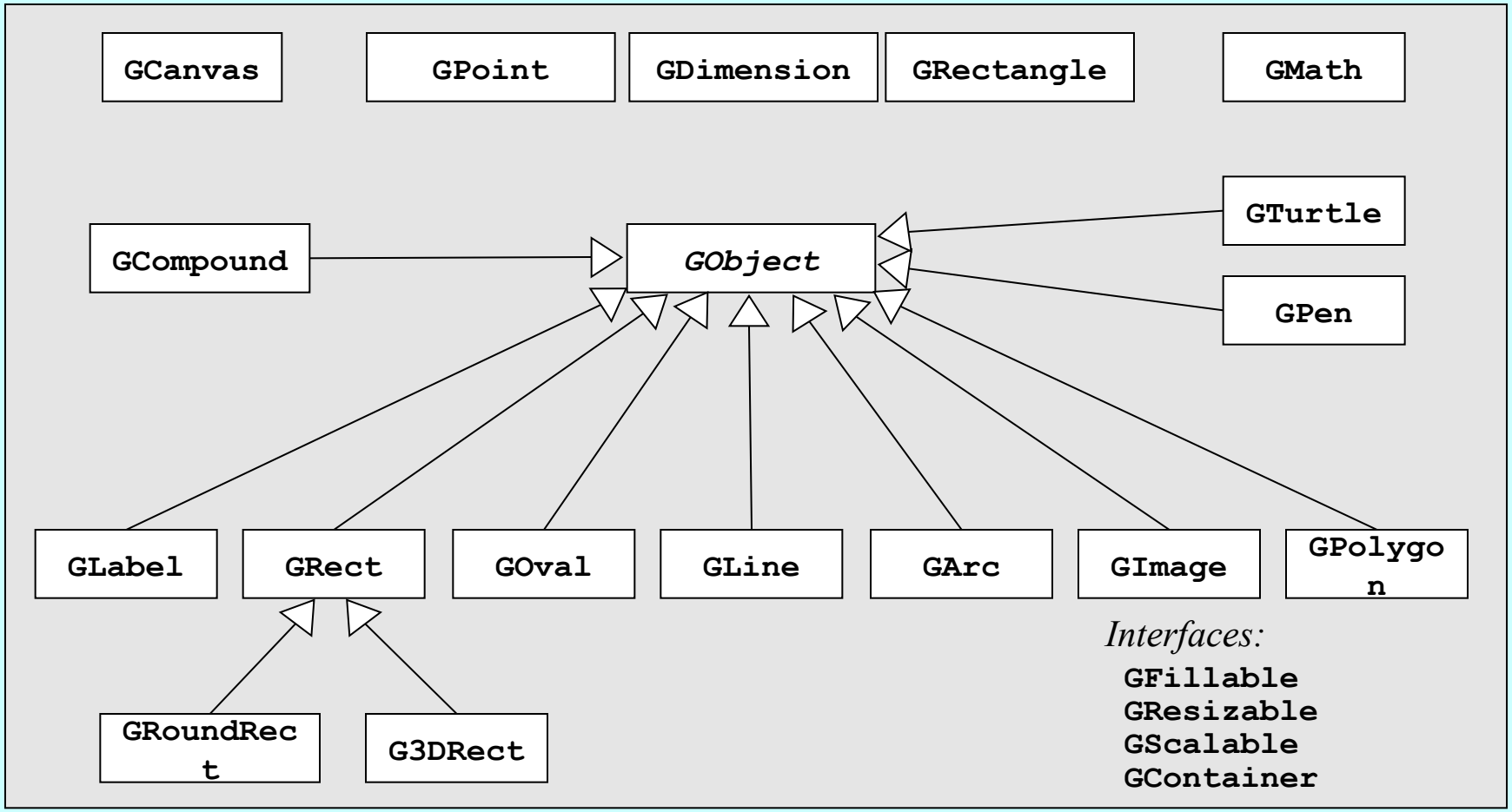
- Note that newer objects can obscure those added earlier. This layering arrangement is called the **stacking order**.

The Java Coordinate System

- As you know from your experience with the **`acm.graphics`** package, all distances and coordinates in the graphics library are measured in **pixels**, which are the tiny dots that cover the surface of the screen.
- Coordinates in the graphics model are specified relative to the **origin** in the upper left corner of the screen.
- Coordinate values are specified as a pair of floating-point values (x, y) where the values for x increase as you move rightward across the screen and the values for y increase as you move downward.
- If you are familiar with traditional Cartesian geometry, it is important to remember that Java treats y values differently, inverting them from their standard interpretation.

Structure of the `acm.graphics` Package

The following diagram shows the classes in the **`acm.graphics`** package and their relationship in the Java class hierarchy:



The **GCanvas** Class

- The **GCanvas** class is used to represent the background canvas for the collage model and therefore serves as a virtual felt board. When you use the **acm.graphics** package, you create pictures by adding various **GObject**s to a **GCanvas**.
- For simple applications, you won't actually need to work with an explicit **GCanvas** object. Whenever you run a program that extends **GraphicsProgram**, the initialization process for the program automatically creates a **GCanvas** and resizes it so that it fills the program window.
- Most of the methods defined for the **GCanvas** class are also available in a **GraphicsProgram**, thanks to an important strategy called **forwarding**. If, for example, you call the **add** method in a **GraphicsProgram**, the program passes that message along to the underlying **GCanvas** object by calling its **add** method with the same arguments.

Methods in the `GCanvas` Class

The following methods are available in both the **`GCanvas`** and **`GraphicsProgram`** classes:

`add(`*object*`)`**	Adds the object to the canvas at the front of the stack
`add(`*object*`, `*x*`, `*y*`)`**	Moves the object to (*x*, *y*) and then adds it to the canvas
`remove(`*object*`)`**	Removes the object from the canvas
`removeAll()`	Removes all objects from the canvas
`getElementAt(`*x*`, `*y*`)`**	Returns the frontmost object at (*x*, *y*), or **`null`** if none
`getWidth()`	Returns the width in pixels of the entire canvas
`getHeight()`	Returns the height in pixels of the entire canvas
`setBackground(`*c*`)`**	Sets the background color of the canvas to *c*.

The following methods are available in **`GraphicsProgram`** only:

`pause(`*milliseconds*`)`**	Pauses the program for the specified time in milliseconds
`waitForClick()`	Suspends the program until the user clicks the mouse

The Two Forms of the **`add`** Method

- The **`add`** method comes in two forms. The first is simply

```
add(object);
```

  which adds the object at the location currently stored in its internal structure. You use this form when you have already set the coordinates of the object, which usually happens at the time you create it.

- The second form is

```
add(object, x, y);
```

  which first moves the object to the point (*x*, *y*) and then adds it there. This form is useful when you need to determine some property of the object before you know where to put it. If, for example, you want to center a **`GLabel`**, you must first create it and then use its size to determine its location.

Encapsulated Coordinates

- The **acm.graphics** package defines three classes—**GPoint**, **GDimension**, and **GRectangle**—that combine geometric information about coordinates and sizes into a single object.
- The **GPoint** class encapsulates the *x* and *y* coordinates of a point. You can create a new **GPoint** object by calling **new GPoint(***x***,** *y***)**. Given a **GPoint**, you can retrieve its coordinates by calling the getter methods **getX** and **getY**.
- The **GDimension** class encapsulates *width* and *height* values that specify an object's size. You create a new **GDimension** by invoking **new GDimension(***width***,** *height***)** and retrieve its components by calling **getWidth** and **getHeight**.
- The **GRectangle** class encapsulates all four of these values by specifying both the origin and size of a rectangle. The constructor form is **new GRectangle(***x***,** *y***,** *width***,** *height***)** and the getters are **getX**, **getY**, **getWidth**, and **getHeight**.

The **GMath** Class

The **GMath** class exports the following static methods:

GMath.sinDegrees(*theta***)**	Returns the sine of *theta,* measured in degrees
GMath.cosDegrees(*theta***)**	Returns the cosine of *theta*
GMath.tanDegrees(*theta***)**	Returns the tangent of *theta*
GMath.angle(*x***, *y*)**	Returns the angle in degrees formed by the line connecting the origin to the point (x, y)
GMath.angle(x_0**, **y_0**, **x_1**, **y_1**)**	Returns the angle in degrees formed by the line connecting the points (x_0, y_0) and (x_1, y_1)
GMath.distance(*x***, *y***)**	Returns the distance from the origin to (x, y)
GMath.distance(x_0**, **y_0**, **x_1**, **y_1**)**	Returns the distance from (x_0, y_0) to (x_1, y_1)
GMath.toRadians(*degrees***)**	Converts an angle from degrees to radians
GMath.toDegrees(*radians***)**	Converts an angle from radians to degrees
GMath.round(*x***)**	Returns the closest **int** to x

Methods Common to All `GObject`s

`setLocation(`*x*`,` *y*`)`**	Resets the location of the object to the specified point
`move(`*dx*`,` *dy*`)`**	Moves the object *dx* and *dy* pixels from its current position
`movePolar(`*r*`,` *theta*`)`**	Moves the object *r* pixel units in direction *theta*
`getX()`	Returns the *x* coordinate of the object
`getY()`	Returns the *y* coordinate of the object
`getWidth()`	Returns the horizontal width of the object in pixels
`getHeight()`	Returns the vertical height of the object in pixels
`contains(`*x*`,` *y*`)`**	Returns **`true`** if the object contains the specified point
`setColor(`*c*`)`**	Sets the color of the object to the **`Color`** *c*
`getColor()`	Returns the color currently assigned to the object
`setVisible(`*flag*`)`**	Sets the visibility flag (**`false`**=invisible, **`true`**=visible)
`isVisible()`	Returns **`true`** if the object is visible
`sendToFront()`	Sends the object to the front of the stacking order
`sendToBack()`	Sends the object to the back of the stacking order
`sendForward()`	Sends the object forward one position in the stacking order
`sendBackward()`	Sends the object backward one position in the stacking order

Sharing Behavior through Interfaces

- In addition to the methods defined for all **`GObject`**s shown on the preceding slide, there are a few methods that apply to some **`GObject`** subclasses but not others. You already know, for example, that you can call **`setFilled`** on either a **`GOval`** or a **`GRect`**. At the same time, it doesn't make sense to call **`setFilled`** on a **`GLine`** because there is no interior to fill.
- In Java, the best strategy when you have methods that apply to some subclasses but not others is to define an **interface** that specifies the shared methods. An interface definition is similar to a class definition except that the interface omits the implementation of each method, retaining only the header line that shows the types of each argument.
- In the **`acm.graphics`** package, there are three interfaces that define methods for certain **`GObject`** subclasses: **`GFillable`**, **`GResizable`**, and **`GScalable`**. The methods in these interfaces appear on the next slide.

Methods Defined by Interfaces

`GFillable` (**`GArc`**, **`GOval`**, **`GPolygon`**, **`GRect`**)

`setFilled(`*flag***`)`**	Sets the fill state for the object (**`false`**=outlined, **`true`**=filled)
`isFilled()`	Returns the fill state for the object
`setFillColor(`*c***`)`**	Sets the color used to fill the interior of the object to *c*
`getFillColor()`	Returns the fill color

`GResizable` (**`GImage`**, **`GOval`**, **`GRect`**)

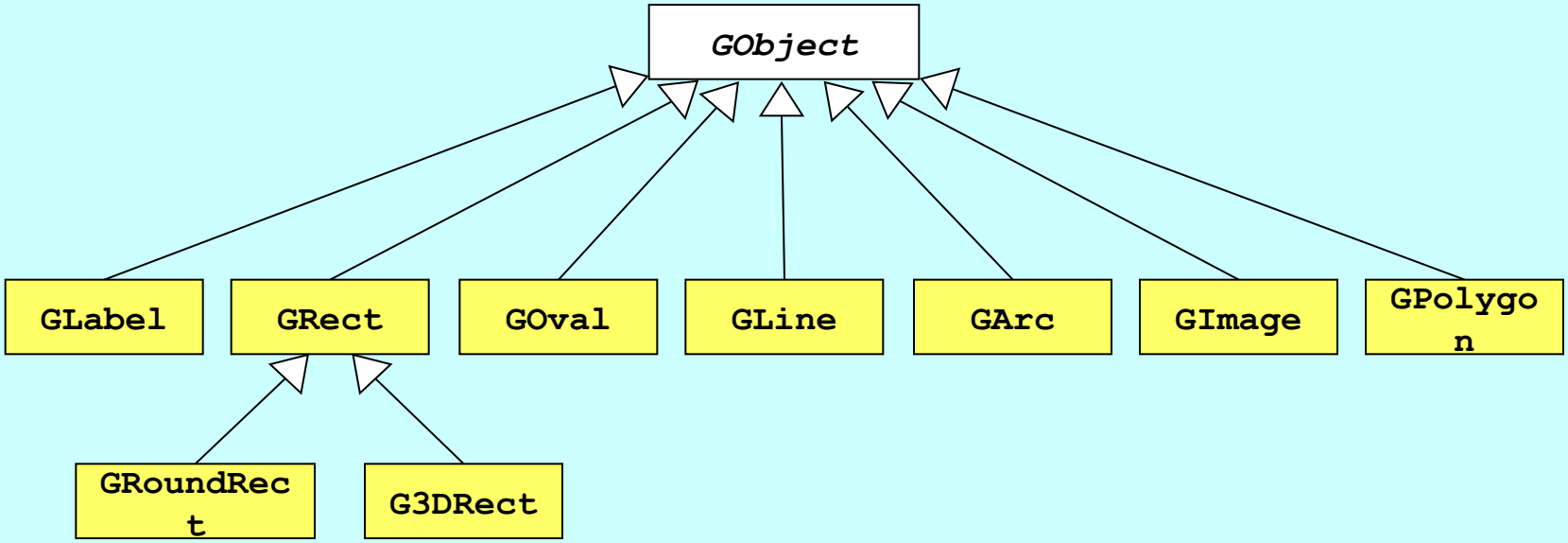
`setSize(`*width***`,` *height***`)`**	Sets the dimensions of the object as specified
`setBounds(`*x***`,` *y***`,` *width***`,` *height***`)`**	Sets the location and dimensions together

`GScalable` (**`GArc`**, **`GCompound`**, **`GLine`**, **`GImage`**, **`GOval`**, **`GPolygon`**, **`GRect`**)

`scale(`*sf***`)`**	Scales both dimensions of the object by *sf*
`scale(`*sx***`,` *sy***`)`**	Scales the object by *sx* horizontally and *sy* vertically

Using the Shape Classes

- The shape classes are the **GObject** subclasses that appear in yellow at the bottom of the hierarchy diagram.



- Each of the shape classes corresponds precisely to a method in the **Graphics** class in the **java.awt** package. Once you have learned to use the shape classes, you will easily be able to transfer that knowledge to Java's standard graphics tools.

The `GLabel` Class

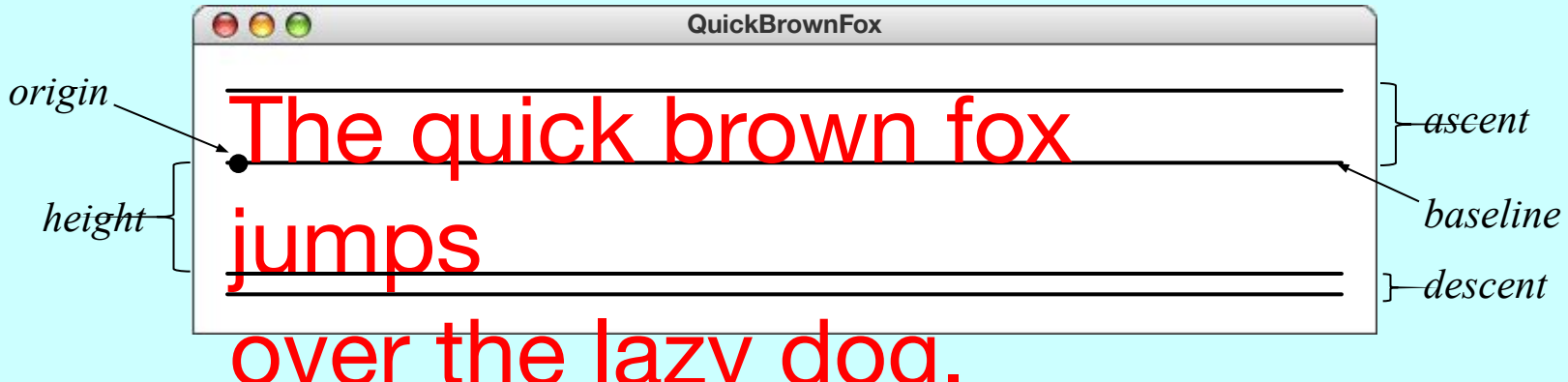
You've been using the **`GLabel`** class ever since Chapter 2 and already know how to change the font and color, as shown in the most recent version of the "Hello World" program:

```
public class HelloProgram extends GraphicsProgram {
   public void run() {
      GLabel label = new GLabel("hello, world", 100, 75);
      label.setFont("SansSerif-36");
      label.setColor(Color.RED);
      add(label);
   }
}
```

The Geometry of the `GLabel` Class

- The **`GLabel`** class relies on a set of geometrical concepts that are derived from classical typesetting:
  - The **baseline** is the imaginary line on which the characters rest.
  - The **origin** is the point on the baseline at which the label begins.
  - The **height** of the font is the distance between successive baselines.
  - The **ascent** is the distance characters rise above the baseline.
  - The **descent** is the distance characters drop below the baseline.

- You can use the **`getHeight`**, **`getAscent`**, and **`getDescent`** methods to determine the corresponding property of the font. You can use the **`getWidth`** method to determine the width of the entire label, which depends on both the font and the text.



Centering Labels

The following update to the "Hello World" program centers the label in the window:

```
public class HelloProgram extends GraphicsProgram {
   public void run() {
      GLabel label = new GLabel("hello, world");
      label.setFont("SansSerif-36");
      label.setColor(Color.RED);
      double x = (getWidth() - label.getWidth()) / 2;
      double y = (getHeight() - label.getAscent()) / 2;
      add(label, x, y);
   }
}
```

The `GRect` Class

- The **`GRect`** class implements the **`GFillable`**, **`GResizable`**, and **`GScalable`** interfaces but does not otherwise extend the facilities of **`GObject`**.
- Like every other shape class, the **`GRect`** constructor comes in two forms. The first includes both the location and the size:

```
new GRect(x, y, width, height)
```

  This form makes sense when you know in advance where the rectangle belongs.

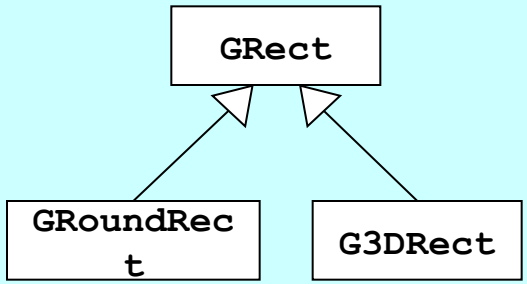
- The second constructor defers setting the location:

```
new GRect(width, height)
```

  This form is more convenient when you want to create a rectangle and then decide where to put it later.

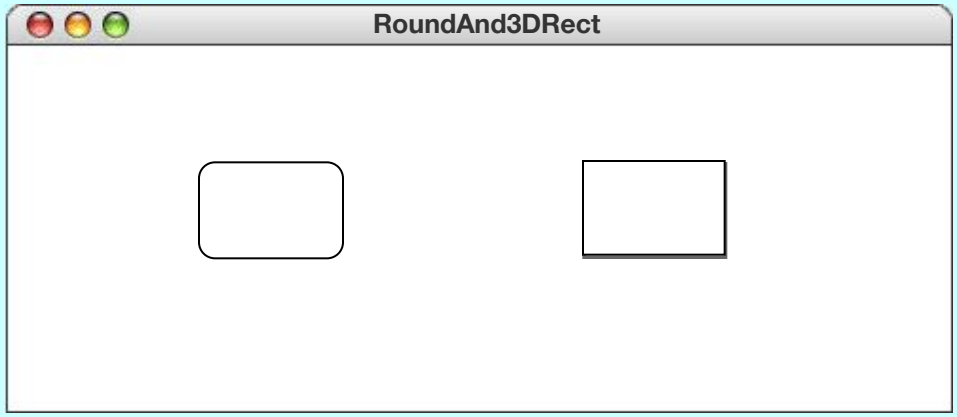
GRoundRect and G3DRect

- As the class hierarchy diagram indicates, the **GRect** class has two subclasses. In keeping with the rules of inheritance, any instance of **GRoundRect** or **G3DRect** is also a **GRect** and inherits its behavior.



- The sample run at the bottom of the screen shows what happens if you execute the following calls:

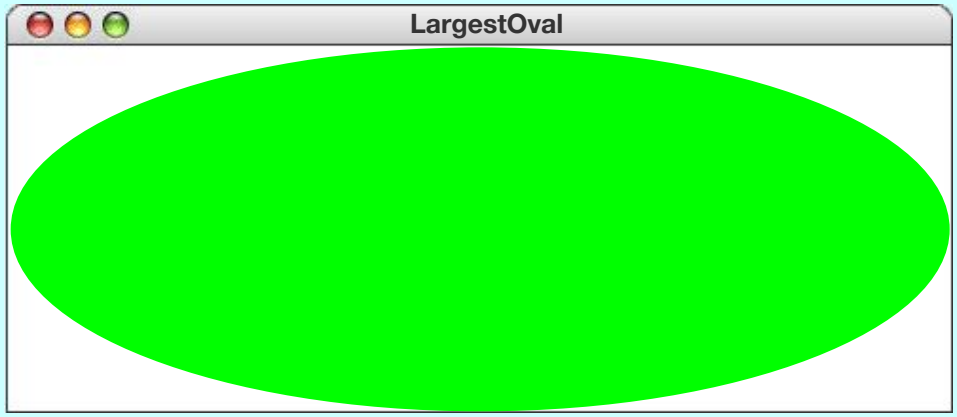
```
add(new GRoundRect(100, 60, 75, 50));
add(new G3DRect(300, 60, 75, 50));
```



The `GOval` Class

- The **`GOval`** class represents an elliptical shape defined by the boundaries of its enclosing rectangle.
- As an example, the following **`run`** method creates the largest oval that fits within the canvas:

```
public void run() {
   GOval oval = new GOval(getWidth(), getHeight());
   oval.setFilled(true);
   oval.setColor(Color.GREEN);
   add(oval, 0, 0);
}
```



The **GLine** Class

- The **GLine** class represents a line segment that connects two points. The constructor call looks like this:

  new GLine(x_0**,** y_0**,** x_1**,** y_1**)**

  The points (x_0,y_0) and (x_1,y_1) are called the **start point** and the **end point**, respectively.

- The **GLine** class does not support filling or resizing but does implement the **GScalable** interface. When you scale a line, its start point remains fixed.

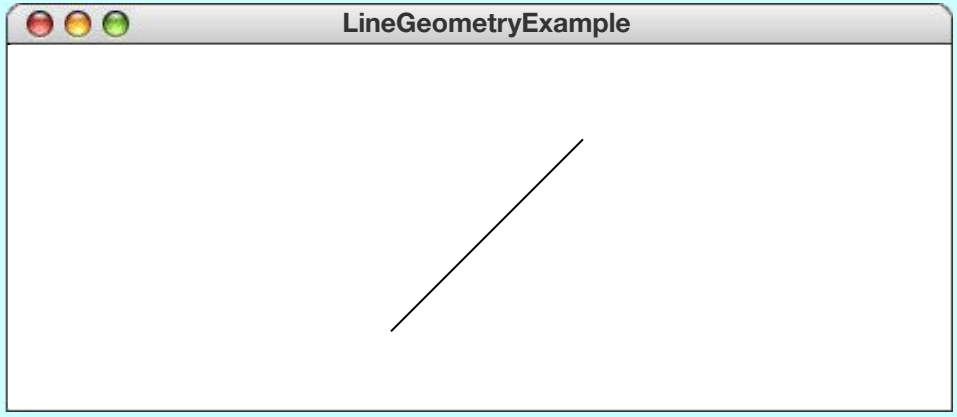
- Given a **GLine** object, you can get the coordinates of the two points by calling **getStartPoint** and **getEndPoint**. Both of these methods return a **GPoint** object.

- The **GLine** class also exports the methods **setStartPoint** and **setEndPoint**, which are illustrated on the next slide.

Setting Points in a `GLine`

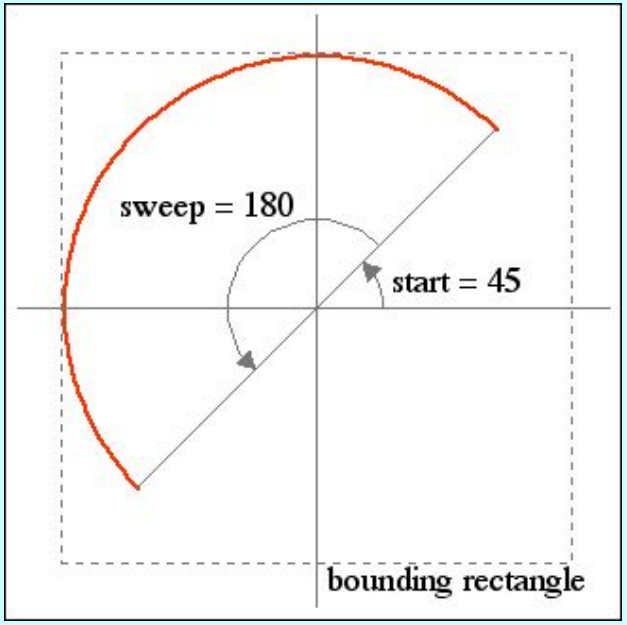
The following **`run`** method illustrates the difference between the **`setLocation`** method (which moves both points together) and **`setStartPoint`**/**`setEndPoint`** (which move only one):

```
public void run() {
   GLine line = new GLine(0, 0, 100, 100);
   add(line);
   line.setLocation(200, 50);
   line.setStartPoint(200, 150);
   line.setEndPoint(300, 50);
}
```



The `GArc` Class

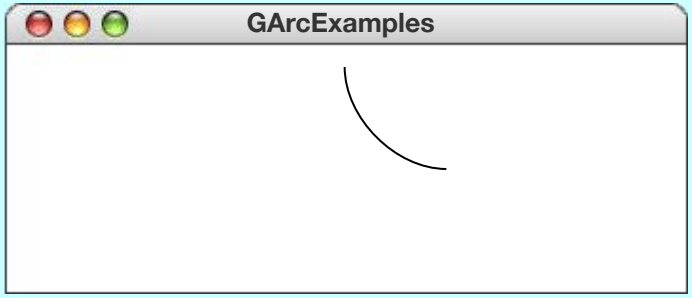
- The **`GArc`** class represents an arc formed by taking a section from the perimeter of an oval.
- Conceptually, the steps necessary to define an arc are:
  - Specify the coordinates and size of the bounding rectangle.
  - Specify the **start angle**, which is the angle at which the arc begins.
  - Specify the **sweep angle**, which indicates how far the arc extends.
- The geometry used by the **`GArc`** class is shown in the diagram on the right.
- In keeping with Java's graphics model, angles are measured in degrees starting at the $+x$ axis (the 3:00 o'clock position) and increasing counterclockwise.
- Negative values for the *start* and *sweep* angles signify a clockwise direction.



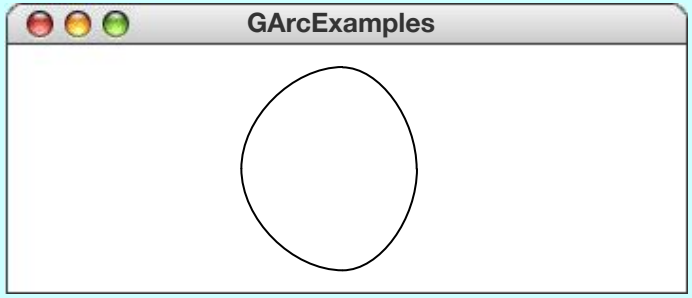
Exercise: `GArc` Geometry

Suppose that the variables **`cx`** and **`cy`** contain the coordinates of the center of the window and that the variable **`d`** is 0.8 times the screen height. Sketch the arcs that result from each of the following code sequences:

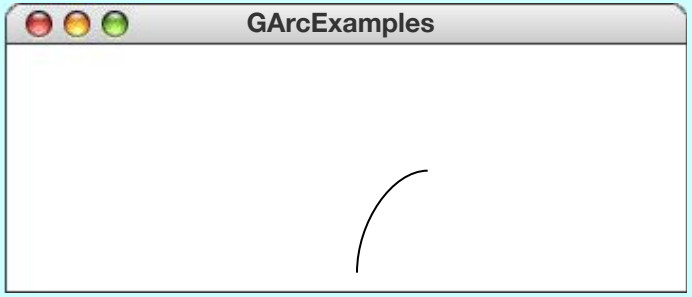
```
GArc a1 = new GArc(d, d, 0, 90);
add(a1, cx - d / 2, cy - d / 2);
```



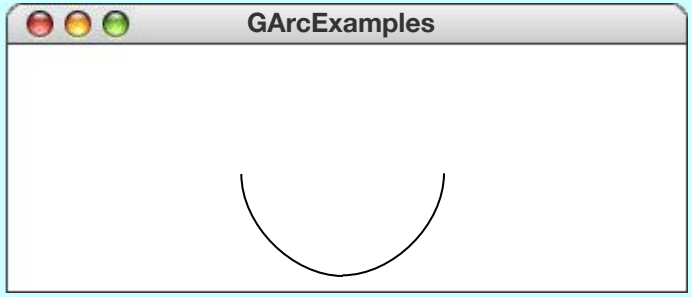

```
GArc a2 = new GArc(d, d, 45, 270);
add(a2, cx - d / 2, cy - d / 2);
```




```
GArc a3 = new GArc(d, d, -90, 45);
add(a3, cx - d / 2, cy - d / 2);
```



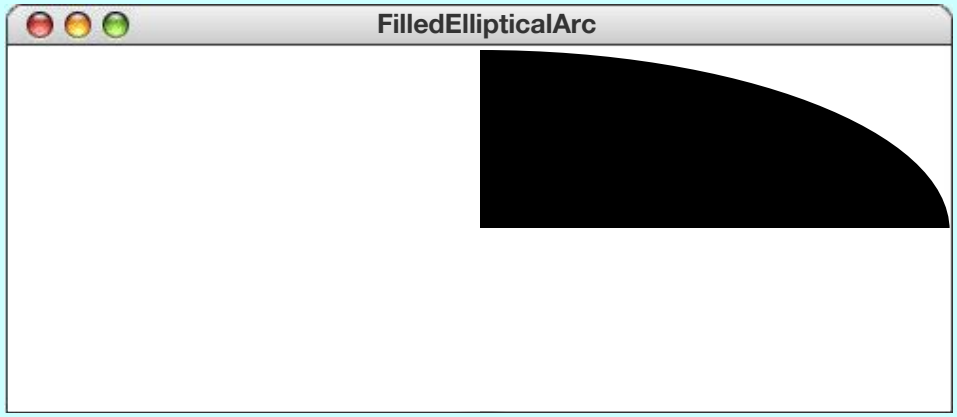

```
GArc a4 = new GArc(d, d, 0, -180);
add(a4, cx - d / 2, cy - d / 2);
```



Filled Arcs

- The **GArc** class implements the **GFillable** interface, which means that you can call **setFilled** on a **GArc** object.
- A filled **GArc** is displayed as the pie-shaped wedge formed by the center and the endpoints of the arc, as illustrated below:

```
public void run() {
   GArc arc = new GArc(0, 0, getWidth(), getHeight(),
                       0, 90);
   arc.setFilled(true);
   add(arc);
}
```



The `GImage` Class

- The **`GImage`** class is used to display an image from a file. The constructor has the form

```
new GImage(image file, x, y)
```

  where *image file* is the name of a file containing a stored image and *x* and *y* are the coordinates of the upper left corner of the image.

- When Java executes the constructor, it looks for the file in the current directory and then in a subdirectory named **`images`**.

- To make sure that your programs will run on a wide variety of platforms, it is best to use one of the two most common image formats: the Graphical Interchange Format (GIF) and the Joint Photographic Experts Group (JPEG) format. Typically, your image file name will end with the suffix **`.gif`** for GIF files and either **`.jpg`** or **`.jpeg`** for JPEG files.

Images and Copyrights

- Most images that you find on the web are protected by copyright under international law.
- Before you use a copyrighted image, you should make sure that you have the necessary permissions. For images that appear of the web, the hosting site often specifies what rules apply for the use of that image. For example, images from the **`www.nasa.gov`** site can be used freely as long as you include the following citation identifying the source:

  Courtesy
  NASA/JPL-Caltech

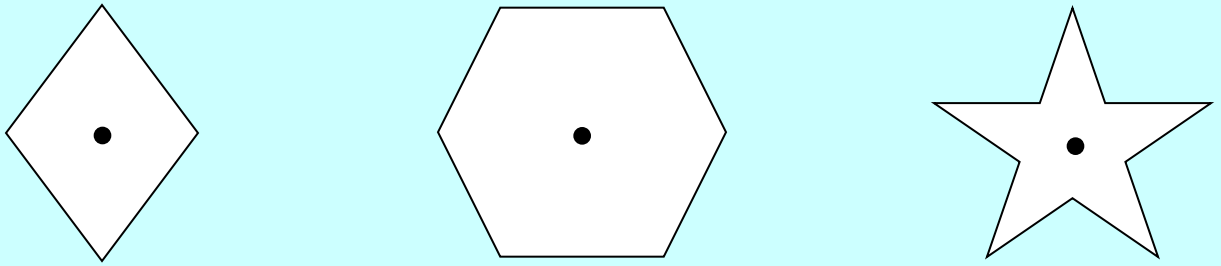
- In some cases, noncommercial use of an image may fall under the "fair use" doctrine, which allows some uses of proprietary material. Even in those cases, however, academic integrity and common courtesy both demand that you cite the source of any material that you have obtained from others.

Example of the `GImage` Class

```
public void run() {
   add(new GImage("EarthFromApollo17.jpg"));
   addCitation("Courtesy NASA/JPL-Caltech");
}
```

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:



- 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.

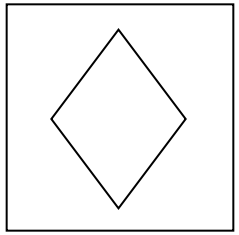
Drawing a Diamond (`addVertex`)

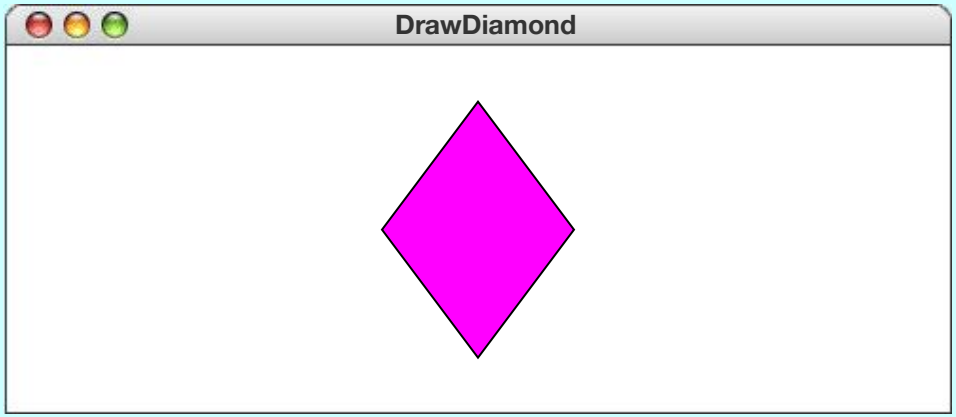
The following program draws a diamond using **`addVertex`**:

```
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





skip simulation

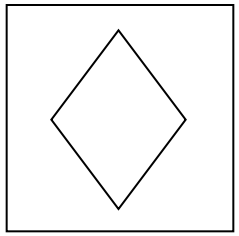
Drawing a Diamond (`addEdge`)

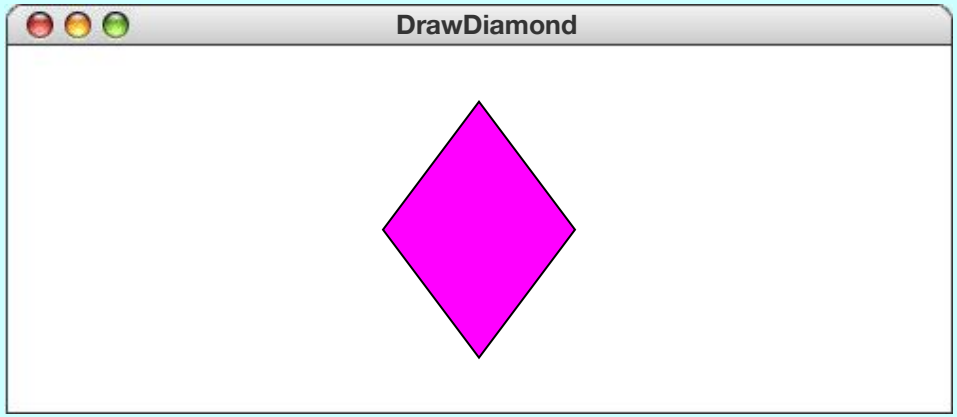
This program draws the same diamond using **`addEdge`**:

```
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

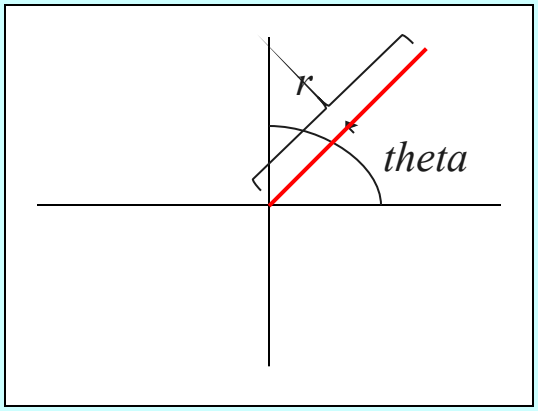




skip simulation

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**.

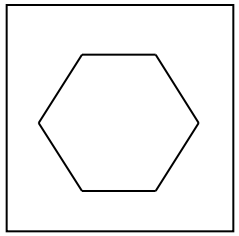
Drawing a Hexagon

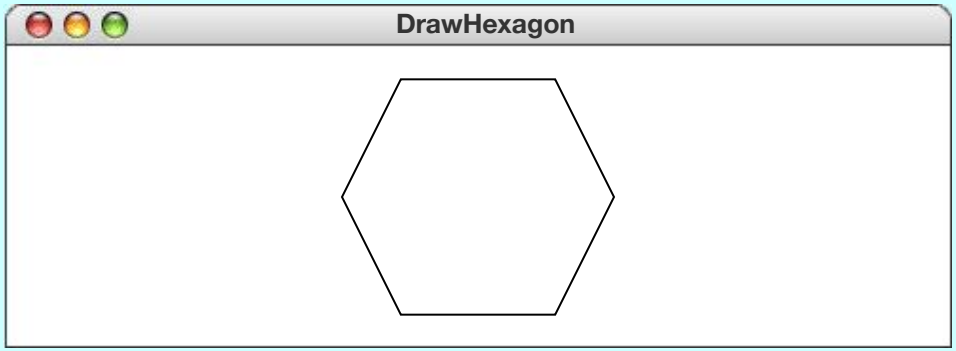
This program draws a regular hexagon using **`addPolarEdge`**:

```
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

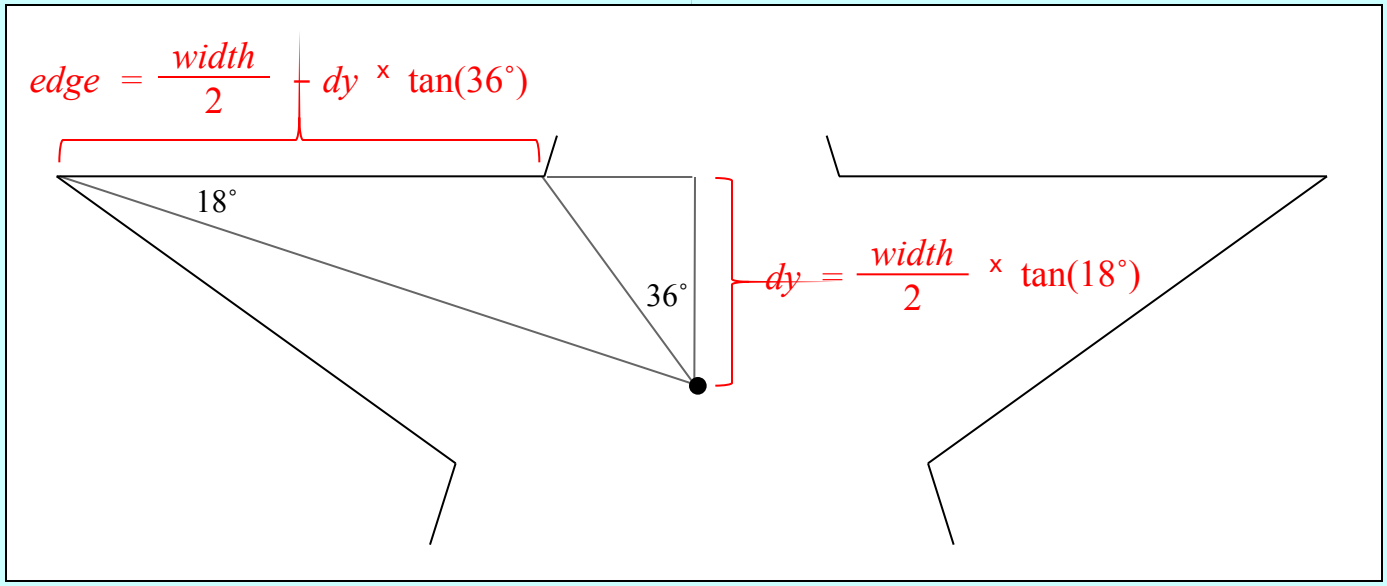
- The **`GPolygon`** class can also serve as the superclass for new types of graphical objects. For example, instead of calling a method like the **`createHexagon`** method from the preceding slide, you could also define a **`GHexagon`** class like this:

```
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.

Drawing a Five-Pointed Star

- As a second example of a new class that extends **`GPolygon`**, the **`GStar`** class on the next slide represents a graphical object that appears as a five-pointed star. The size is determined by the **`width`** parameter to the constructor.
- The only real complexity in the code involves computing the location of the initial vertex and the length of each edge in the star. This calculation requires some simple trigonometry:



The GStar Class

```
/**
 * Defines a new GObject class that appears as a
 * five-pointed star.
 */
public class GStar extends GPolygon {

   public GStar(double width) {
      double dx = width / 2;
      double dy = dx * GMath.tanDegrees(18);
      double edge = width / 2 - dy * GMath.tanDegrees(36);
      addVertex(-dx, -dy);
      int angle = 0;
      for (int i = 0; i < 5; i++) {
         addPolarEdge(edge, angle);
         addPolarEdge(edge, angle + 72);
         angle -= 72;
      }
   }

}
```

Using the `GStar` Class

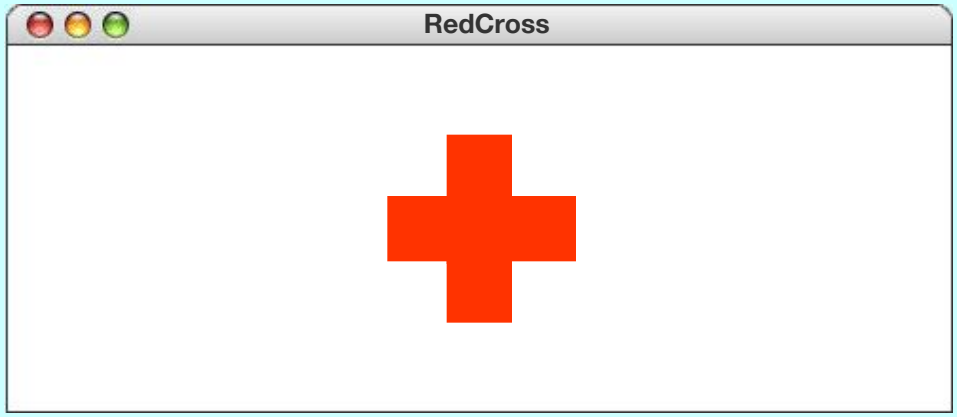
The following program draws five random stars:

```
public void run() {
   for (int i = 0; i < 5; i++) {
      GStar star = new GStar(rgen.nextDouble(20, 100));
      star.setFilled(true);
      star.setColor(rgen.nextColor());
      double x = rgen.nextDouble(50, getWidth() - 50);
      double y = rgen.nextDouble(50, getHeight() - 50);
      add(star, x, y);
      pause(500);
   }
}
```

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);
}
```



Solution: The `GCross` Class

```
class GCross extends GPolygon {
   public GCross(double size) {
      double edge = size / 3;
      addVertex(-size / 2, -edge / 2);
      addEdge(edge, 0);
      addEdge(0, -edge);
      addEdge(edge, 0);
      addEdge(0, edge);
      addEdge(edge, 0);
      addEdge(0, edge);
      addEdge(-edge, 0);
      addEdge(0, edge);
      addEdge(-edge, 0);
      addEdge(0, -edge);
      addEdge(-edge, 0);
      addEdge(0, -edge);
   }
}
```

The **`addArc`** Method

- To make it easier to display shapes that combine straight and curved segments, the **`GPolygon`** class includes a method called **`addArc`** that adds an entire series of small edges to a polygon that simulate an arc.
- A call to the **`addArc`** method has the form

```
polygon.addArc(width, height, start, sweep);
```

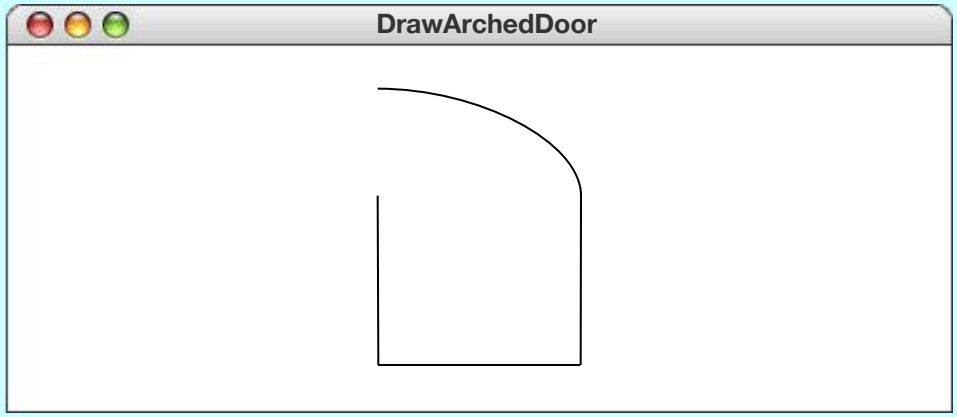
  where the arguments are interpreted as they are for the **`GArc`** constructor: the *width* and *height* parameters specify the size of the bounding rectangle, and the *start* and *sweep* parameters indicate the starting point and extent of the arc.

- The coordinates for the arc are not specified explicitly in the **`addArc`** call but are instead chosen so that the starting point of the arc is the current point on the polygon outline.

Using the **addArc** Method

The following class definition creates a new **GPolygon** subclass that appears as an arched doorway, as shown in the sample run:

```
public class GArchedDoor extends GPolygon {
   public GArchedDoor(double width, double height) {
      double lengthOfVerticalEdge = height - width / 2;
      addVertex(-width / 2, 0);
      addEdge(width, 0);
      addEdge(0, -lengthOfVerticalEdge);
      addArc(width, width, 0, 180);
      addEdge(0, lengthOfVerticalEdge);
   }
}
```

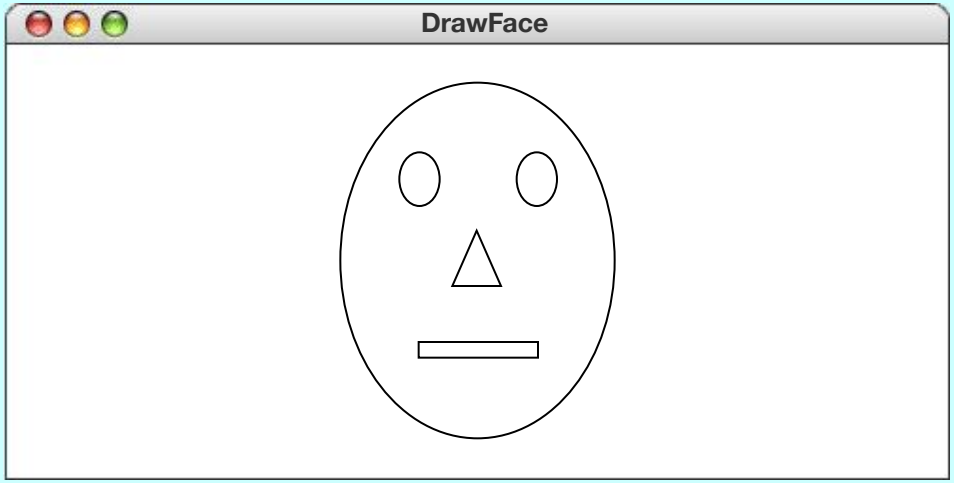


Creating Compound Objects

- 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.

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 **`GOval`** 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 acm.graphics.*;

/** 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 GOval(width, height);
      leftEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      rightEye = new GOval(EYE_WIDTH * width, EYE_HEIGHT * height);
      nose = createNose(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);
   }
```

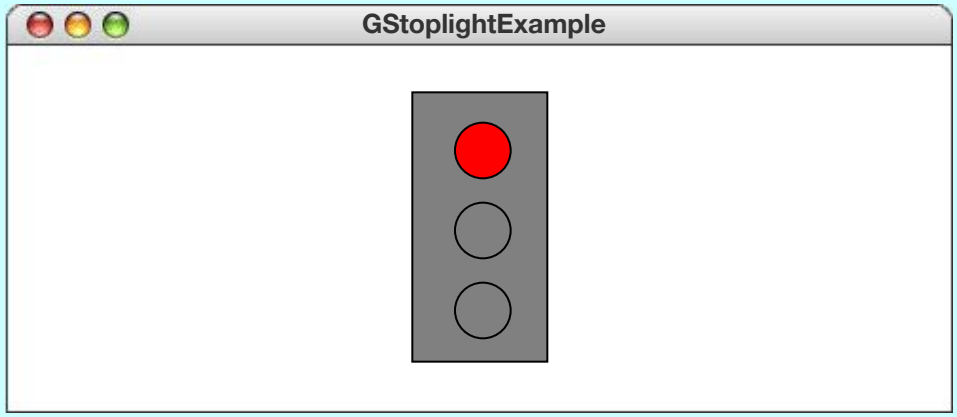
The GFace Class

```
/* Creates a triangle for the nose */
   private GPolygon createNose(double width, double height) {
      GPolygon poly = new GPolygon();
      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  = 0.10;
   private static final double MOUTH_WIDTH  = 0.50;
   private static final double MOUTH_HEIGHT = 0.03;

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

Specifying Behavior of a `GCompound`

- The **`GCompound`** class is useful for defining graphical objects that involve behavior beyond that common to all **`GObject`**s.
- The **`GStoplight`** on the next slide implements a stoplight object that exports methods to set and get which lamp is on. The following code illustrates its use:

```
public void run() {
   GStoplight stoplight = new GStoplight();
   add(stoplight, getWidth() / 2, getHeight() / 2);
   stoplight.setColor("RED");
}
```



The GStoplight Class

```
/**
 * Defines a GObject subclass that displays a stoplight.  The
 * state of the stoplight must be one of the Color values RED,
 * YELLOW, or GREEN.
 */
public class GStoplight extends GCompound {

/** Creates a new Stoplight object, which is initially GREEN */
   public GStoplight() {
      GRect frame = new GRect(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setFilled(true);
      frame.setFillColor(Color.GRAY);
      add(frame, -FRAME_WIDTH / 2, -FRAME_HEIGHT / 2);
      double dy = FRAME_HEIGHT / 4 + LAMP_RADIUS / 2;
      redLamp = createFilledCircle(0, -dy, LAMP_RADIUS);
      add(redLamp);
      yellowLamp = createFilledCircle(0, 0, LAMP_RADIUS);
      add(yellowLamp);
      greenLamp = createFilledCircle(0, dy, LAMP_RADIUS);
      add(greenLamp);
      setState(Color.GREEN);
   }
```

The `GStoplight` Class

```
/** Sets the state of the stoplight */
   public void setState(Color color) {
      if (color.equals(Color.RED)) {
         redLamp.setFillColor(Color.RED);
         yellowLamp.setFillColor(Color.GRAY);
         greenLamp.setFillColor(Color.GRAY);
      } else if (color.equals(Color.YELLOW)) {
         redLamp.setFillColor(Color.GRAY);
         yellowLamp.setFillColor(Color.YELLOW);
         greenLamp.setFillColor(Color.GRAY);
      } else if (color.equals(Color.GREEN)) {
         redLamp.setFillColor(Color.GRAY);
         yellowLamp.setFillColor(Color.GRAY);
         greenLamp.setFillColor(Color.GREEN);
      }
      state = color;
   }

/** Returns the current state of the stoplight */
   public Color getState() {
      return state;
   }
```

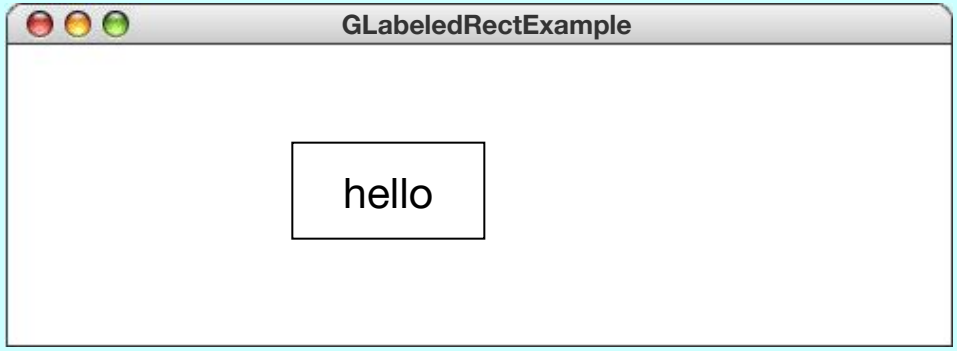
The GStoplight Class

```
/* Creates a filled circle centered at (x, y) with radius r */
   private GOval createFilledCircle(double x, double y, double r) {
      GOval circle = new GOval(x - r, y - r, 2 * r, 2 * r);
      circle.setFilled(true);
      return circle;
   }

/* Private constants */
   private static final double FRAME_WIDTH = 50;
   private static final double FRAME_HEIGHT = 100;
   private static final double LAMP_RADIUS = 10;

/* Private instance variables */
   private Color state;
   private GOval redLamp;
   private GOval yellowLamp;
   private GOval greenLamp;
}
```

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);
}
```



Solution: The `GLabeledRect` Class

```
/** Defines a graphical object combining a rectangle and a label */
public class GLabeledRect extends GCompound {

/** Creates a new GLabeledRect object */
   public GLabeledRect(int width, int height, String text) {
      frame = new GRect(width, height);
      add(frame);
      label = new GLabel(text);
      add(label);
      recenterLabel();
   }

/** Creates a new GLabeledRect object at a given point */
   public GLabeledRect(int x, int y, int width, int height,
                       String text) {
      this(width, height, text);
      setLocation(x, y);
   }

/** Sets the label font */
   public void setFont(String font) {
      label.setFont(font);
      recenterLabel();
   }
```

Solution: The `GLabeledRect` Class

```
/** Sets the text of the label */
   public void setLabel(String text) {
      label.setLabel(text);
      recenterLabel();
   }

/** Gets the text of the label */
   public String getLabel() {
      return label.getLabel();
   }

/* Recenters the label in the window */
   private void recenterLabel() {
      double x = (frame.getWidth() - label.getWidth()) / 2;
      double y = (frame.getHeight() + label.getAscent()) / 2;
      label.setLocation(x, y);
   }

/* Private instance variables */
   private GRect frame;
   private GLabel label;
}
```

The `GCompound` Coordinate System

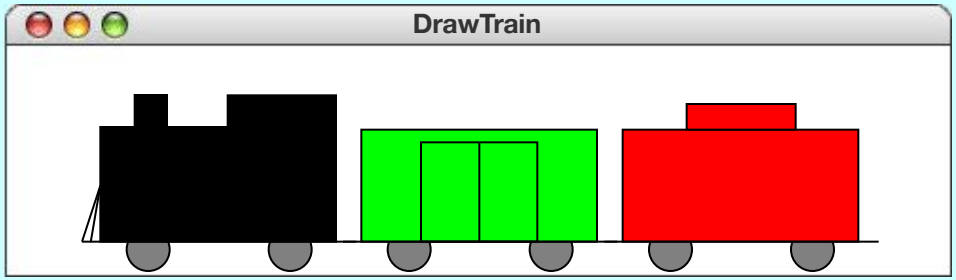
- As noted on an earlier slide, the components of a **`GCompound`** object use a local coordinate system in which *x* and *y* values are interpreted relative to the reference point.
- On some occasions (most notably if you need to work with mouse coordinates as described in Chapter 10), it is useful to be able to convert back and forth between the local coordinate system used within the compound and the coordinate system of the canvas as a whole. This capability is provided by the following methods:

`getCanvasPoint(`*x*`,` *y*`)`**	Converts the local point (*x*, *y*) to canvas coordinates
`getLocalPoint(`*x*`,` *y*`)`**	Converts the canvas point (*x*, *y*) to local coordinates

Each of these methods returns a **`GPoint`** that encapsulates the *x* and *y* coordinates in a single object.

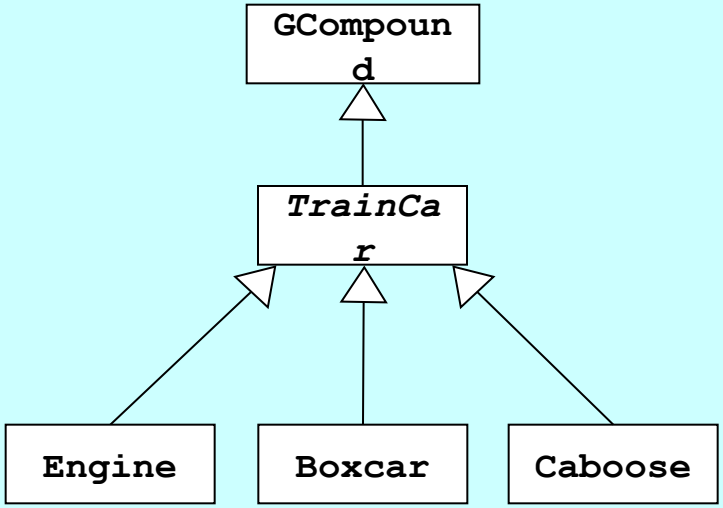
Graphical Object Decomposition

- 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 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

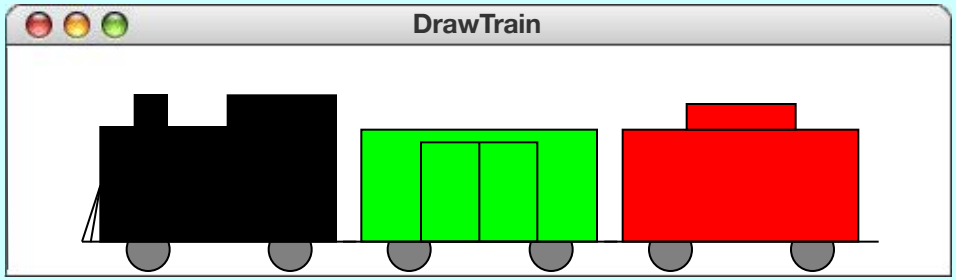
```
/**
 * This class represents a boxcar.  Like all TrainCar subclasses,
 * a Boxcar is a graphical object that you can add to a GCanvas.
 */
public class Boxcar extends TrainCar {

/**
 * 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);
   }

}
```

The End