Chapter 4: Interfaces and Polymorphism

Chapter Topics

- Interface
- Polymorphism
- The Comparable Interface
- The Comparator Interface
- Anonymous Classes
- GUI Programming
- Event Handling
- Simple animation using Timer

4.1 Interface

4.1.1 What is an Interface ?

- It is a promise that your class will implement certain method with certain signatures.

```
public interface Icon
{
  int getIconWidth();
  int getIconHeight();
  void paintIcon(Component c, Graphics g, int x, int y);
}
```

- Like classes,
  - can be a standard interface from the library or a user-defined interface.
  - they define types.
  - an interface type can extend another by adding additional requirement.

```
public interface Movable extends Icon
{
    void translate(int x, int y);
}
```

- Unlike classes, they
  - don't have instance variables. (If you define a variable inside an interface, it is automatically declared as a public static final variable. A subclass that implements the interface or a sub interface that extends it will inherit this constant.)

```
public interface ImageObserver
{
      int ABORT = 128; // a public static final constant
}
```

  - contain method definitions only (no implementation)
  - can never be instantiated

- all methods in an interface are automatically `public abstract`
- A class can implement as many interfaces as it likes

```
public class MarsIcon implements Icon, Shape
{
}
```

4.1.2 How to use an interface ?

A class that implements an interface must complete all the methods defined in the interface. Otherwise, it should be an abstract class.

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

/**
   An icon that has the shape of the planet Mars.
*/
public class MarsIcon implements Icon
{  private int size;
   /**
      Constructs a Mars icon of a given size.
      @param aSize the size of the icon
   */
   public MarsIcon(int aSize)
   {  size = aSize; }

   public int getIconWidth()
   {  return size; }

   public int getIconHeight()
   {  return size; }

   public void paintIcon(Component c, Graphics g, int x, int y)
   {  Graphics2D g2 = (Graphics2D) g;
      Ellipse2D.Double planet = new Ellipse2D.Double(x, y, size, size);
      g2.setColor(Color.RED);
      g2.fill(planet);
   }
}
```

The `paintIcon` method

An icon, say *img*, can be drawn on components, such as JComponent or JPanel, using

```
img.paintIcon(Component c, Graphics g, int x, int y)
```

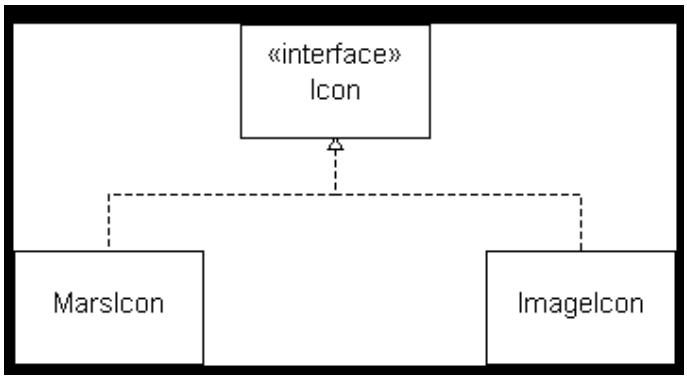
- `Component`: the user interface component containing the icon
- `Graphics`: graphics context, an object to encapsulate drawing property
- `x` and `y`: the location of the icon inside the component

To display the icon on a component, put the paintIcon call in the paintComponent method of a component.

```
class MyComponent extends JComponent
{    Icon img = new MarsIcon(50);
     public void paintComponent(Graphics g)
     {      super.paintComponent(g);
            img.paintIcon(this, g, 100, 100);
     }
}
```

4.1.3 Benefits of using Interfaces

An interface and a class that implements the interface model the "*is-a*" relationship in the real world. In the following UML diagram, MarsIcon is an Icon, and ImageIcon is also an Icon.



Let's take an example to learn the benefits of using interfaces. Let's consider the showMessageDialog method of JOptionPane.

```
public static void showMessageDialog(Component parentComponent,
   Object message, String title, int messageType, Icon icon)
```

Parameters:

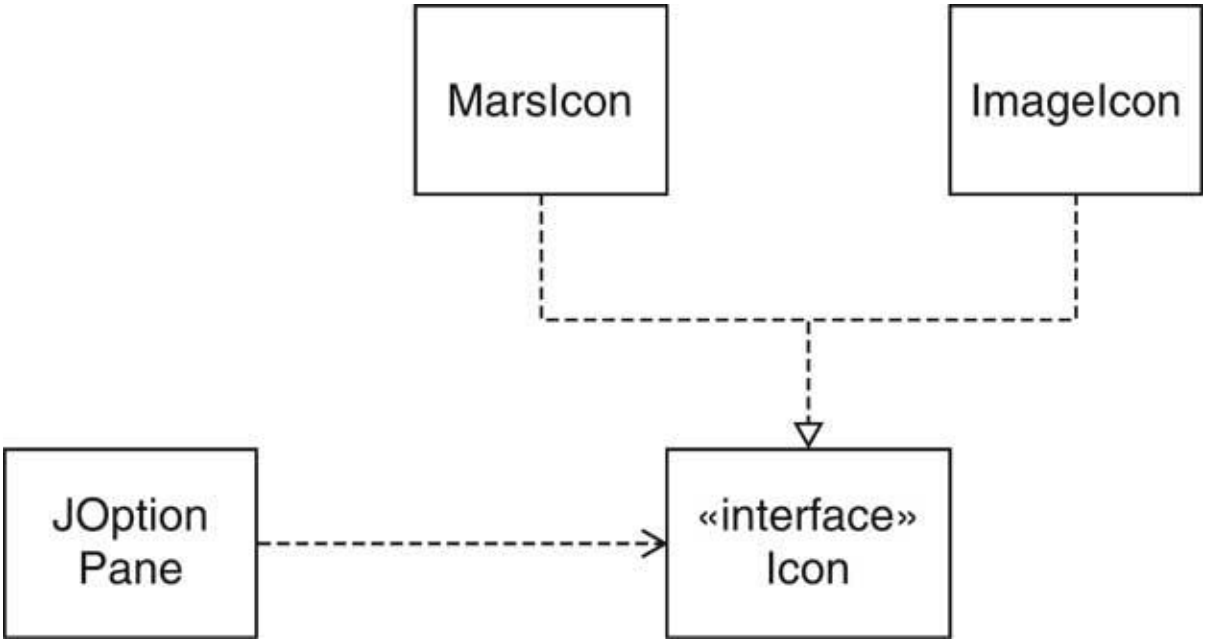
- parentComponent - determines the Frame in which the dialog is displayed; if null, or if the parentComponent has no Frame, a default Frame is used
- message - the Object to display
- title - the title string for the dialog
- messageType - the type of message to be displayed: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, or PLAIN_MESSAGE
- icon - an icon to display in the dialog that helps the user identify the kind of message that is being displayed

Our interest is in the parameter icon. The parameter is defined in terms of Icon so that any thing that *is an* icon can be passed to the parameter.

```
(1) JOptionPane.showMessageDialog(null, "Hello, World!","Message",
              JOptionPane.INFORMATION_MESSAGE, new ImageIcon("globe.gif"));
(2) JOptionPane.showMessageDialog(null, "Hello, Mars!","Message",
              JOptionPane.INFORMATION_MESSAGE, new MarsIcon(50));
```

Benefits

- Loose coupling: The showMessageDialog decoupled from ImageIcon or MarsIcon.
- Extensibility: Client can supply new icon types



Polymorphism is an important OO principle that supports the effective use of interfaces.

4.2 Polymorphism

In OOP, a method with the same signature can appear in many different forms. *Polymorphism* is the ability to select the appropriate method for a particular object. This is late binding because it is done at run time by VM.

Let's see the use of polymorphism in the showMessageDialog method. We learned in the previous section that the method takes the parameter of Icon type that can take any Icon object as shown below.

```
public static void showMessageDialog(..., Icon anIcon)
{  …
   int iconWidth = anIcon.getIconWidth();
   int iconHeight = anIcon.getIconHeight();



   // anIcon.method_specific_to_marsIcon(); // compilation error
   …
}
```

The method needs to calculate the size of the dialog box of JoptionPane. As a part of the dialog box, the icon size needs to be figured out by calling anIcon.getIconWidth(). You need to understand two different perspectives on this particular line.

- The compiler checks the *type* of anIcon, which is Icon, and see if the type defines the getIconWidth method.
- The virtual machine calls the most suitable method for anIcon according to the *object* the anIcon actually references, not the type of anIcon. For example, with the method call (1) shown in the previous section, the Virtual Machine calls the getIconWidth method of ImageIcon. With the method call (2), the Virtual Machine calls that of MarsIcon.

4.3 The Comparable Interface Type

The Comparable interface defines the compareTo method as shown below.

```
public interface Comparable<T>
{   int compareTo(T other); }
```

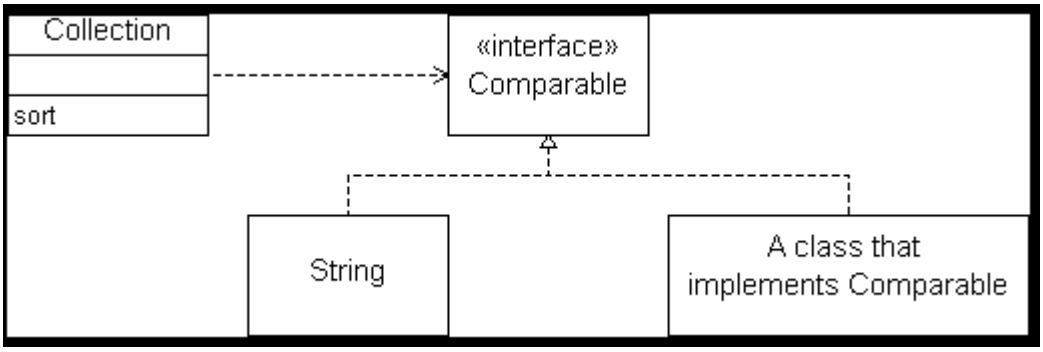
The call object1.compareTo(object2) returns

- a negative number if object1 should come before object2
- zero if the objects are equal
- a positive number otherwise.

The sort method of Collections class expects a list consisting of Comparable objects. Here is the method definition.

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

The sort method takes a List consisting of Comparable objects. The sort method doesn't know any specific Comparable objects. As long as the List elements are Comparable, it should be able to sort them. That is, the method is decoupled from any specific Comparable objects.




```
class Country implements Comparable<Country>
{
   private String name;
   private double area;

   public Country(String aName, double anArea)
   {  name = aName;
      area = anArea;
   }
   public String getName()
   {  return name; }

   public double getArea()
   {  return area; }

   public int compareTo(Country other)
   {  if (area < other.area) return -1;
      if (area > other.area) return 1;
      return 0;
   }
}

public class CountrySortTester
{  public static void main(String[] args)
   {  ArrayList<Country> countries = new ArrayList<Country>();
      countries.add(new Country("Uruguay", 176220));
      countries.add(new Country("Thailand", 514000));
      countries.add(new Country("Belgium", 30510));

      Collections.sort(countries);
      // Now the array list is sorted by area
      for (Country c : countries)
          System.out.println(c.getName() + " " + c.getArea());
   }
}
```

Output

```
Belgium 30510.0
Uruguay 176220.0
Thailand 514000.0
```

4.4 The Comparator Interface Type

Problem

There are drawbacks in the previous approach.

- You need to change the source code of a class, such as Country, if you want to make it Comparable. Therefore, this approach is not applicable for the classes whose source codes are not available, e.g. Java standard classes from the library.
    - You need to explicitly put the java keyword "implements" at the class header. You also need to modify the source code by adding the compareTo method to it.
- It is not practical to redefine the compareTo method every time you want to change the comparison metrics, for example, from areas to names.

Solution

- Use the Comparator interface to resolve the problem.

```
public interface Comparator<T>
{   int compare(T obj1, T obj2); }
```

The compare method returns a negative number, zero, or a positive number depending on whether first is less than, equal to, or greater than second in the particular sort order.

- Write a Comparator class which knows how to compare two Countries by name. Note that you have a full control over the class you are writing.

```
public class CountryComparatorByName implements Comparator<Country>
{   public int compare(Country country1, Country country2)
    {   return country1.getName().compareTo(country2.getName()); }
}
```

- The Collections class provides you with more flexible sort method.

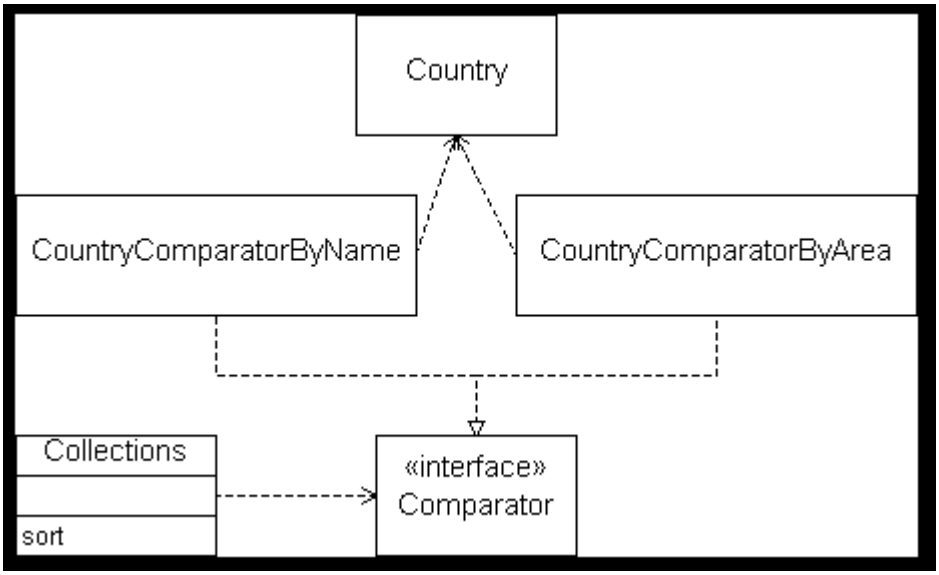
```
public static <T> void sort(List<T> list, Comparator<? super T> c)
```

The method sorts the objects in the list according to the sort order that the Comparator defines. Note that the list can contain any objects.

- Pass the Comparator object to the sort method. An object such as Comparator is called a *function object* because its sole purpose is to execute the comparison function.

```
Comparator<Country> comp = new CountryComparatorByName();
Collections.sort(countries, comp);
```

- If you want to change the comparison metrics, write another Comparator which knows how to compare two countries by area, and pass it to the sort method.



Example: application to a Java standard class, Rectangle

```
import java.util.*;
import java.awt.Rectangle;

public class RectangleComparatorByArea implements Comparator<Rectangle>
{   public int compare(Rectangle r1, Rectangle r2)
    {   return (int)(r1.getWidth()*r1.getHeight()
                          - r2.getWidth()*r2.getHeight());}
}
public class RectangleComparatorByWidth implements Comparator<Rectangle>
{   public int compare(Rectangle r1, Rectangle r2)
    {   return (int)(r1.getWidth() - r2.getWidth());  }
}
public class ComparatorTester
{  public static void main(String[] args)
   {
      ArrayList<Rectangle> rectangles = new ArrayList<Rectangle>();
      rectangles.add(new Rectangle(0,0,10,30));
      rectangles.add(new Rectangle(0,0,20, 5));

      Comparator<Rectangle> comp = new RectangleComparatorByArea();

      Collections.sort(rectangles, comp);
      for (Rectangle s: rectangles)
           System.out.println(s);  // (0,0,20,5), (0,0,10,30)

      comp = new RectangleComparatorByWidth();
      Collections.sort(rectangles, comp);
      for (Rectangle s: rectangles)
        System.out.println(s);  // (0,0,10,30), (0,0,20,5)
   }
}
```

Example 2: Application to a user defined class, Country

```
public class CountryComparatorByName implements Comparator<Country>
{    public int compare(Country country1, Country country2)
     {  return country1.getName().compareTo(country2.getName()); }
}
public class CountryComparatorByArea implements Comparator<Country>
{    public int compare(Country country1, Country country2)
     {  return (int) (country1.getArea() - country2.getArea()); }
}
public class ComparatorTester
{  public static void main(String[] args)
   {    ArrayList<Country> countries = new ArrayList<Country>();
        countries.add(new Country("Uruguay", 176220));
        countries.add(new Country("Thailand", 514000));
        countries.add(new Country("Belgium", 30510));
        Comparator<Country>  comp = new CountryComparatorByName();
        Collections.sort(countries, comp);
        for (Country c : countries)
           System.out.println(c.getName() + " " + c.getArea());
        comp = new CountryComparatorByArea();
        Collections.sort(countries, comp);
        for (Country c : countries)
           System.out.println(c.getName() + " " + c.getArea());
   }
}
```

4.5. Anonymous Classes

Anonymous object: If an object is used only once in the program, you don't have to name it.

```
Collections.sort(countries, new CountryComparatorByName());
```

Anonymous class: In the same context, you don't need to name classes that are used only once.

```
Comparator<Country> comp = new
   Comparator<Country>()
   {
      public int compare(Country country1, Country country2)
      { return country1.getName().compareTo(country2.getName());}
   };

Collections.sort(countries, comp);
```

The above statement

- defines anonymous class that implements `Comparator<Country>`
- defines compare method of that class
- constructs one object of that class

What it means is this: Create a new object of a class that implements the Comparator<Country> interface, where the required method is the one defined inside the braces {}.

The above code performs the same function as the following code segment does.

```
class MyComparator implements Comparator<Country>
{
   public int compare(Country country1, Country country2)
   {  return country1.getName().compareTo(country2.getName());}
}
Comparator<Country> comp = new MyComparator();
Collections.sort(countries, comp);
```

[Q] What if you want to create multiple objects of the anonymous class?
[A] Put the construction inside a method that returns such an object.

```
public class Country
{  …
   public static Comparator<Country> comparatorByName()
   {
     return new
        Comparator<Country>()
        {
          public int compare(Country country1, Country country2)
          { return country1.getName().compareTo(country2.getName());}
        };
   }
   …
}
```

Then, call the sort method like this:

```
public class ComparatorTester
{  public static void main(String[] args)
   {
      ArrayList<Country> countries1 = new ArrayList<Country>();
      ArrayList<Country> countries2 = new ArrayList<Country>();

      // Assume Country objects are added to these ArrayLists.

      Collections.sort(countries1, Country.comparatorByName());
      Collections.sort(countries2, Country.comparatorByName());
   }
}
```

4.6 Graphic Programming

These notes are based on the text book and "Core Java, Vol 1" by Horstmann and Cornell, chapter 7, graphics programming.

4.6.1 AWT vs. Swing

AWT (Abstract Window Toolkit)

- AWT, a class library for basic GUI programming, has been available since Java 1.0. The basic AWT library takes the *peer-based* approach to deal with user interface elements. That is, it delegates the creations and behaviors of GUI elements to the native GUI toolkit on each target platform.
- Package: `java.awt`
- Example classes: `Frame, Component, Panel`

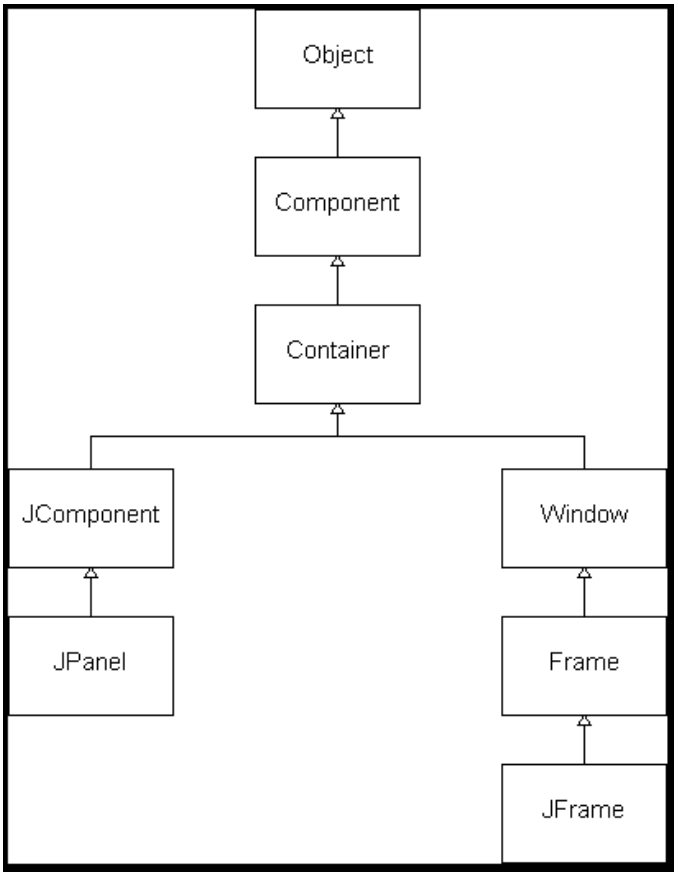
Swing

- Swing is a *non-peer-based* GUI toolkit, and became the part of the standard library in JDK 1.2.
- Swing
    - Has a rich and convenient set of user interface elements.
    - Has few dependencies on the underlying platform; it is therefore less prone to platform-specific bugs.
    - Gives a consistent user experience among platforms.

- Package: `javax.swing`
- Example classes: `JFrame, JComponent, JPanel`

Note 1: In general, AWT components are appropriate for simple applet development or development that targets a specific platform (i.e. the Java program will run on only one platform). For most any other Java GUI development you will want to use Swing components.
Note 2: Swing is not a complete replacement of AWT. It uses the foundations of AWT, in particular, event handling.

4.6.2 Inheritance Hierarchy for the JFrame and JPanel classes



4.6.3 JFrame and JComponent

JFrame

As a top-level window, a frame in Java is designed to be containers for components. Specifically, the add method of JFrame, which is inherited from Container, takes a Component:

```
public Component add(Component comp)
```

JComponent

JComponents model GUI components that can be displayed on the frame, such as JButtons, JLabels, JPanels, and so on.

Example: Creating a frame and adding components to it.

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class FrameTester
5  {  public static void main(String[] args)
6     {  JFrame frame = new JFrame();
7
8        final int FRAME_WIDTH = 300;
9        final int FRAME_HEIGHT = 200;
```

```
10        frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
11
12        JButton helloButton = new JButton("Say Hello");
13
14        final int FIELD_WIDTH = 20;
15        JTextField textField = new JTextField(FIELD_WIDTH);
16        textField.setText("Click a button!");
17
18        frame.setLayout(new FlowLayout());
19
20        frame.add(helloButton);
21        frame.add(textField);
22
23        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24        frame.pack();
25        frame.setVisible(true);
26     }
```

1: The package awt is imported to use the FlowLayout manager.
2: The program is basically using swing.
10: If you don't explicitly size a frame, the frame size will be 0 by 0 by default.
12 ~ 16: Two components, JButton and JTextField, are created.
18: You must set a layout manager for the frame to have multiple components. Otherwise, one component will be drawn on the top of another.
20 ~ 21: The created components are added to the frame.
23: The program exits when the user closes this frame.
24: The pack command sets the size of the frame to the smallest size to display its components.
25: Call the setVisible method to show the frame.

Note: The main thread of the program quits after exiting the main method, but the setVisible method spawns another thread. This thread keeps running until the user closes the frame.

4.6.3 Displaying Information in a Panel

- In Java, frames are designed to be containers for components. Normally, you draw on a *panel* and add it to the frame. *Panels* have two useful properties:
  - They have a surface onto which you can draw.
  - They themselves are containers. That is, panels can hold other user interface components such as buttons, sliders, and so on.
- In particular, to draw on panel, you
  - Define a class that extends JPanel; and
  - Override the paintComponent method in that class.

Example

```
import javax.swing.*;
import java.awt.*;

public class HelloTester
```

```
{ public static void main (String[] args)
  {  JFrame frame = new HelloFrame();
     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

     frame.setVisible(true);
  }
}

class HelloFrame extends JFrame
{
   public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
   public HelloFrame()
   {   setTitle("Hello");
       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
       HelloPanel panel = new HelloPanel();
       add(panel);
   }
}

class HelloPanel extends JPanel
{  public void paintComponent(Graphics g)
   {  super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;
      g2.drawString("Hello !", 100, 100);
   }
}
```

The `paintComponent` method

- JComponent supplies the paintComponent. (Note that a JPanel *is a* JComponent.) Each time a window need to be redrawn, the paintComponent methods of all JComponents will be executed. This is
- You should *never* call the paintComponent method in your program. The method is *automatically* called whenever the part of your program needs to be redrawn.
- When would the paintComponent method be automatically called ?
    - When the window is displayed for the first time.
    - The user resizes the window.
    - The user opens another window which covers the existing window. When the new window disappears, the existing window should be redrawn.

`paintComponent vs. repaint`

If you need to force repainting of the screen, call the `repaint` method instead of `paintComponent`. The repaint method will cause paintComponent to be called for all components, with a properly configured Graphics object.

4.6.4 Working with 2D Shapes

(1) Graphics Context

- Graphics class: The class has existed since Java 1.0, and supplies methods to draw basic shapes.
- Graphics2D class: To draw shapes in Java 2 (or later version), you need to obtain an object of the Graphic2D class. The Graphics2D, which extends Graphics, supplies you a powerful set of methods to draw shapes.

If you use Java 2 or later version, the paintComponent, which is called by the system, automatically receives a Graphics2D object. Therefore, you need to cast the parameter to Graphics2D in the method in order to use the features of Graphics2D.

```
public void paintComponent(Graphics g)
{   Graphics2D g2 = (Graphics2D) g;
    …
}
```

(2) Shape Primitives

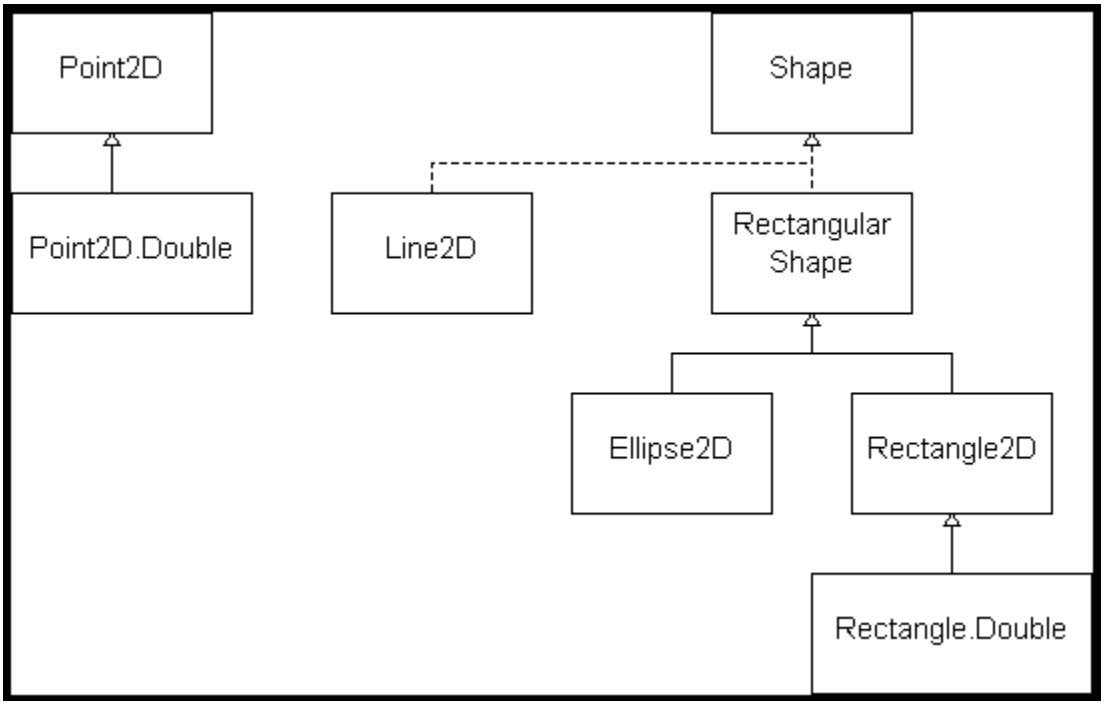
- The shape primitives are contained almost entirely in the `java.awt.geom` package and can be used to draw pretty much anything in two dimensions. All shape primitives implement the java.awt.`Shape` interface, a set of methods for describing shapes.
- All shape primitives implements the java.awt.geom.`PathIterator` object that specifies the outline of the shape.

- 2D shape primitives

  - `Rectangle2D, RoundRectangle2D, Arc2D, and Ellipse2D` are subclasses of RectangularShape. You can describe these primitive through a rectangular bounding box.
  - `Line2D, Quadcurve2D, and CubicCurve2D` are line segments. You need to describe their two endpoints with the requisite control points.
  - `Area` allows the creation of new shapes through the use of intersection, union, and subtraction of other shapes.
  - `Path2D` classes provide full implementations of a general geometric path that support all of the functionality of the `Shape` and `PathIterator` interfaces with the ability to explicitly select different levels of internal coordinate precision.

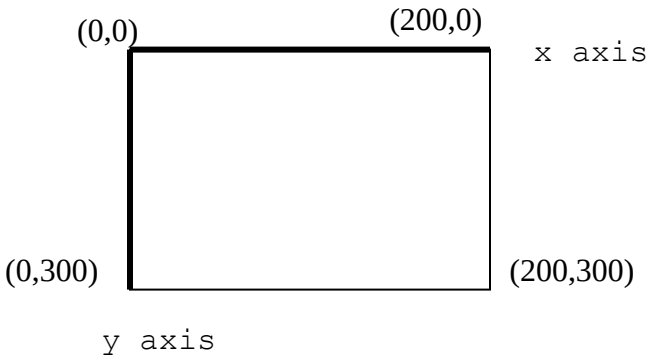
- Note that these classes are abstract classes except for Area. That is, they cannot be instantiated directly but rather are instantiated through a subclass. The class names followed by `.Double` and `.Float` are such sub classes you can instantiate. For example,

```
Rectangle2D r = new Rectangle2D.Double(0, 0, 75, 75));
g2.draw(r);
```



(3) Drawing shapes

To draw a shape, you first create an object of a class that implements the Shape interface and then call the draw method of Graphics2D class. To use graphics methods you need to understand coordinate system. The origin (0,0) is at the top left corner of a frame.



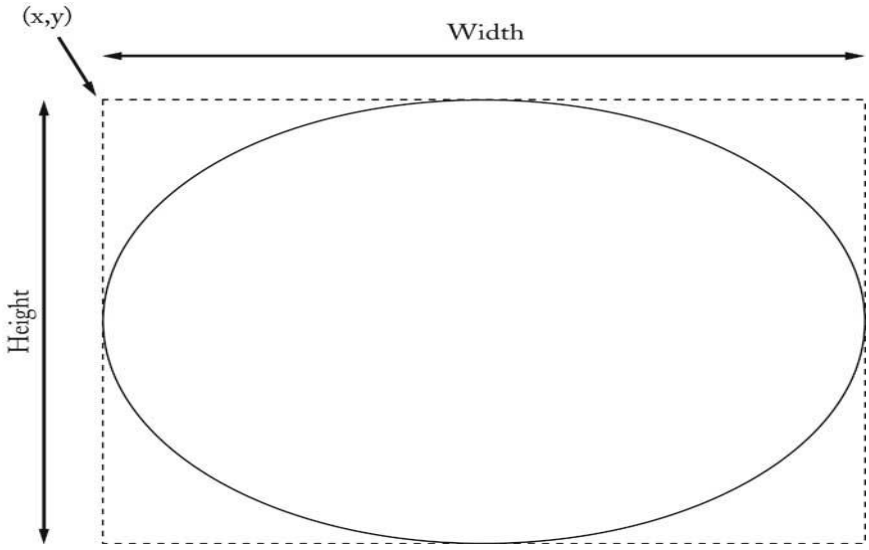
This frame is 200 pixels high, 300 wide

Rectangle and Ellipse

- Construct a Rectangle2D.Double with the top left corner, width, and height, and call g2.draw().
  - Example: `g2.draw(new Rectangle2D.Double(x, y, width, height));`

- For Ellipse2D.Double, specify bounding box

```
g2.draw(new Ellipse2D.Double(x, y, width, height));
```



Line Segments

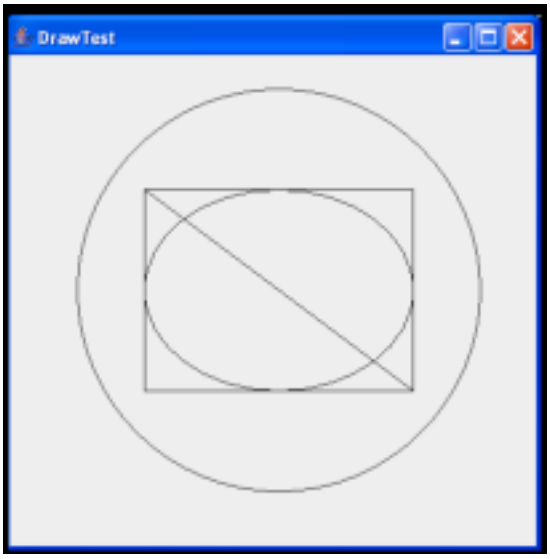
Line2D.Double joins to points.

```
Point2D.Double start = new Point2D.Double(x1, y1);
Point2D.Double end = new Point2D.Double(x2, y2);
Shape segment = new Line2D.Double(start, end);
//or Shape segment = new Line2D.Double(x1, y1, x2, y2);
g2.draw(segment);
```

Filling Shapes

First, set the color in the graphics context: `g2.setPaint(Color.red);`
To fill interior of shape, call `g2.fill(shape);`

Example: To draw primitive shapes on Panel (from Core Java 2, Volume 1)



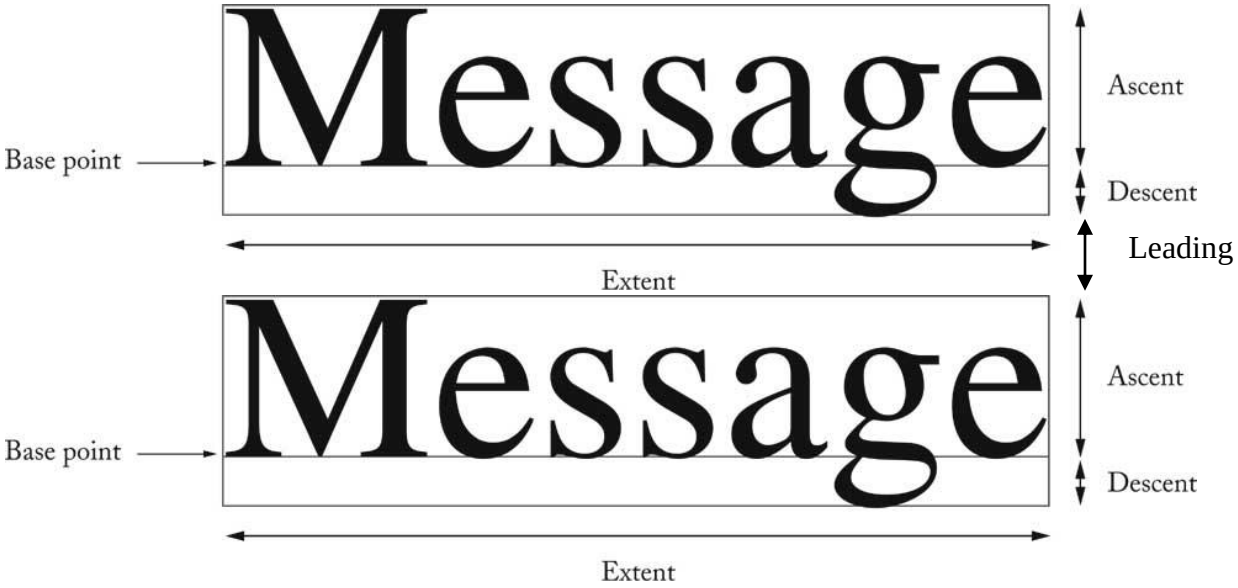

```
import java.awt.*;
import java.awt.geom.*;
import javax.swing.*;

public class DrawTest
{
   public static void main(String[] args)
   {
      DrawFrame frame = new DrawFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   A frame that contains a panel with drawings
*/
class DrawFrame extends JFrame
{
   public static final int DEFAULT_WIDTH = 400;
   public static final int DEFAULT_HEIGHT = 400;

   public DrawFrame()
   {
      setTitle("DrawTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add panel to frame

      DrawPanel panel = new DrawPanel();
      add(panel);
   }
}

/**
   A panel that displays rectangles and ellipses.
*/
```

```
class DrawPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      // draw a rectangle
      double leftX = 100;
      double topY = 100;
      double width = 200;
      double height = 150;

      Rectangle2D rect =
          new Rectangle2D.Double(leftX, topY, width, height);
      g2.draw(rect);

      // draw the enclosed ellipse
      Ellipse2D ellipse = new Ellipse2D.Double();
      ellipse.setFrame(rect);
      g2.draw(ellipse);

      // draw a diagonal line
      g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY +
height));

      // draw a circle with the same center
      double centerX = rect.getCenterX();
      double centerY = rect.getCenterY();
      double radius = 150;

      Ellipse2D circle = new Ellipse2D.Double();
      circle.setFrameFromCenter(centerX, centerY, centerX + radius,
centerY + radius);
      g2.draw(circle);
   }
}
```

Drawing Text

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);String text = "Message";
g2.drawString(text, x, y); // x, y are base point coordinates
```

Logical font names defined by AWT: SansSerif, Serif, Monospaced, Dialog, DialogInput.



Example: To draw a string, its base line and bounds.


```
import java.awt.*;
import java.awt.font.*;
import java.awt.geom.*;
import javax.swing.*;

public class FontTest
{
   public static void main(String[] args)
   {
      FontFrame frame = new FontFrame();
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}

/**
   A frame with a text message panel
*/
class FontFrame extends JFrame
{  public static final int DEFAULT_WIDTH = 300;
   public static final int DEFAULT_HEIGHT = 200;
```

```
   public FontFrame()
   {
      setTitle("FontTest");
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      // add panel to frame

      FontPanel panel = new FontPanel();
      add(panel);
   }
}

/**
   A panel that shows a centered message in a box.
*/
class FontPanel extends JPanel
{
   public void paintComponent(Graphics g)
   {
      super.paintComponent(g);
      Graphics2D g2 = (Graphics2D) g;

      String message = "Hello, World!";

      Font f = new Font("Serif", Font.BOLD, 36);
      g2.setFont(f);

      // measure the size of the message

      FontRenderContext context = g2.getFontRenderContext();
      Rectangle2D bounds = f.getStringBounds(message, context);

      // set (x,y) = top left corner of text

      double x = (getWidth() - bounds.getWidth()) / 2;
      double y = (getHeight() - bounds.getHeight()) / 2;

      // add ascentleading to y to reach the baseline

      double ascentleading = -bounds.getY();
      double baseY = y + ascentleading;

      // draw the message

      g2.drawString(message, (int) x, (int) baseY);

      g2.setPaint(Color.GRAY);


      // draw the baseline
```

```
      g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(),
baseY));

      // draw the enclosing rectangle

      Rectangle2D rect =
        new Rectangle2D.Double(x, y, bounds.getWidth(), bounds.getHeight());
      g2.draw(rect);
   }
}
```

(5) The draw method of Graphics2D takes and draws a `Shape` object. All primitive shapes are `Shapes`. Therefore, you can simply call `g2.draw` (a primitive shape such as Rectangle, Ellipse, Line). What if you want to draw a user-defined shape such as car or house ? There are various ways to do it.

- Directly draw the user-defined shapes in the `paintComponent` or `paintIcon` method. Because a user-defined shape consists of primitive shapes, you can draw it by subsequently calling `g2.draw()` for each primitive shape.

  Example: DrawPanel of the previous example

- Implement an ADT of such a class and have it implement the `Shape` interface.

```
public class Car implements Shape
{

}
```

  You can call `g2.draw (new Car(…));` in the `paintComponent` or `paintIcon` method. But, then the Car class should supply all the methods of the Shape interface.

- Implement an ADT of such a class that defines its own draw method. The draw method should know how to draw the user defined shape. And, have the `paintComponent` or `paintIcon` call the draw method of the user defined shape class. Note that the graphics context should be passed from the paint method to the user-defined draw method.

```
public class Car
{  …
     public void draw(Graphics2D g2)
     {
          Rectangle2D.Double body = …
          Ellipse2D.Double front tire = …
          …
          g2.draw(body);
          g2.draw(tire);
     }
}
```

In the `paintComponent` or `paintIcon` method,

```
Car car = new Car(…);
car.draw(g2);
```

Example: The `CarShape` class (pp. 168) and the `paintIcon` method of `ShapeIcon` (pp.167)

4.7 User Interface Actions

4.7.1 Overview

- ActionListener Interface

```
public interface ActionListener
{  void actionPerformed(ActionEvent event); }
```

- A listener object is an instance of a class that implements the ActionListener interface.

```
new
  ActionListener()
  {  public void actionPerformed(ActionEvent event)
     { textField.setText("Hello, World!");}
  });
```

- An event source, such as button, is an object that can register listener objects and send them event objects, such as ActionEvent.

- You need to add listener objects to an event source.

```
helloButton.addActionListener(new
    ActionListener()
    {
       public void actionPerformed(ActionEvent event)
       {
          textField.setText("Hello, World!");
       }
    });
```

- The event source sends out event objects to all registered listeners when that event occurs. (You can add multiple listeners to one event source.)
- The listener object can use the information in the event object, the ActionEvent parameter, to determine their reaction to the event.

Example: ActionTester.java

```
import java.awt.*;
import java.awt.event.*;
```

```
import javax.swing.*;

public class ActionTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      final int FIELD_WIDTH = 20;
      final JTextField textField = new JTextField(FIELD_WIDTH);
      textField.setText("Click a button!");

      JButton helloButton = new JButton("Say Hello");

      helloButton.addActionListener(new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               textField.setText("Hello, World!");
            }
         });

      frame.setLayout(new FlowLayout());

      frame.add(helloButton);
      frame.add(textField);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

4.7.2 Nested Classes

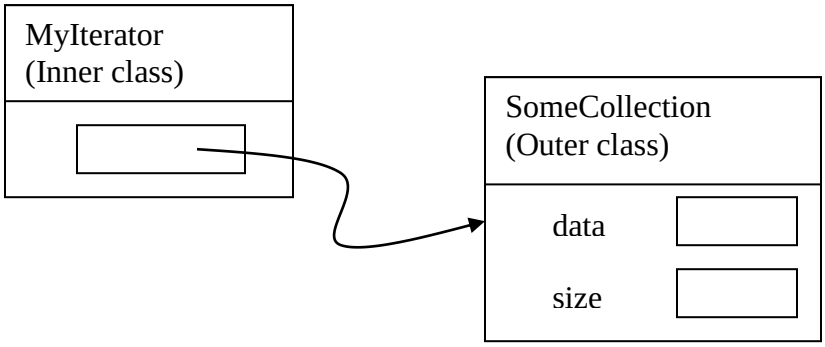
- Inner class (= non-static nested class)
- Static nested class
- Local class
  - Anonymous class (local class without name) – covered in 4.5

Inner class

- A non-static class that is defined inside another class but outside of its methods.
- The inner class is available to all instance methods of the outer class.

- An inner class method gets to access to both its own data fields and those of the outer object creating it.

- An object of an inner class always gets an implicit reference to the object that created it. In this way, an inner class can refer to all data fields and methods of the outer class.
- Note that the field must belong to the object that constructed the inner class object.



- An inner class *cannot* have any *static* method or variable.
- Example

```
import java.util.*;
public class SomeCollection<E> implements Collection<E>
{  private E[] data;
   private int size ;

   public SomeCollection()
   { data = (E[]) new Object[10];
     size = 0;
   }
   public void add(E e)
   {  data[size] = e;
      size++;
   }
   public Iterator iterator()
   {  return new MyIterator(); }

// Assume other methods required by Collection<E> are
implemented here. !

   class MyIterator implements Iterator<E>
   { int count = 0;
     public boolean hasNext() { return count < size; }
     public E next()
     {
        return data[count++];
     }
     public void remove()
     {  throw new UnsupportedOperationException();
     }
    } // inner class
```

```
      public static void main (String [] args)
      {  SomeCollection<String> sc =
                              new SomeCollection<String>();
         sc.add("A");
         sc.add("B");
         Iterator iter = sc.iterator();
         while(iter.hasNext())
         { System.out.println(iter.next() + " "); }
      }
   }
```

Local inner class

- An inner class that is defined locally in a single method.
- Their scope is always restricted to the block in which they are declared.
- Local inner class is completed hidden from the outside world-not even other code in the outer class. For example, if ActionTester defines other methods, they cannot recognize the local inner class defined in the main method method.
- Not only can they access the fields of their outer classes, they can even access local variables of the method that defines it. However, those local variables must be declared *final*.
- If the local inner class is defined inside of a static method, it can only access static members of the outer class.
- Example: `ActionTester.java` in 4.7.1 (Example of local anonymous inner class)

Static nested class

- If you want to use a nested class simply to hide it inside a logically related outer class, but you don't need the nested class to have a reference to the outer class object, then define it static. That is, use a static inner class whenever the nested class does not need to access of the outer class object.
- A static nested class is exactly like inner classes, except that an object of a static nested class doesn't have a reference to the outer class object. As a result, a static nested class can access only static members of outer class. That is, static nested class cannot refer directly to instance variables or instance methods defined in the outer class — it can use them only after creating an object of outer class and then through the object reference.

```
public class LinkedList <E>
{ private Node <E> head = null;
  private Node <E> tail = null;
  private int size = 0;

  private static class Node <E>
  {  private E data;
     private Node <E> next = null;
     private Node <E> prev = null;
     private Node(E dataItem) { data = dataItem;}
  } //end class Node
```

```
  // Other methods of LinkedList are defined here.
}
```

4.7.3 Several action listeners with similar action?

- Write helper method that constructs listener objects and pass variable information as parameters.
- You should declare parameters *final* so that it can be referenced in the method of anonymous class.
- Example

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ActionTester
{  private static JTextField textField;

   public static void main(String[] args)
   {
     JFrame frame = new JFrame();

     final int FIELD_WIDTH = 20;
     textField = new JTextField(FIELD_WIDTH);
     textField.setText("Click a button!");

     JButton helloButton = new JButton("Say Hello");

     helloButton.addActionListener(createGreetingButtonListener(
     "Hello, World!"));

     JButton goodbyeButton = new JButton("Say Goodbye");

     goodbyeButton.addActionListener(createGreetingButtonListene
     r("Goodbye, World!"));

     frame.setLayout(new FlowLayout());

     frame.add(helloButton);
     frame.add(goodbyeButton);
     frame.add(textField);

     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
     frame.pack();
     frame.setVisible(true);
   }

 public static ActionListener createGreetingButtonListener(final
String message)
```

```
 {
     return new
         ActionListener()
         {  public void actionPerformed(ActionEvent event)
             {   textField.setText(message); }
         };
 }
}
```

4.8 Simple Animation

Ten times per second, the car shape will move and the window will be repainted in a way that the new position is displayed.

4.8.1 Timer (javax.swing.Timer)

To use a timer, you need to

1) define an action listener or action listeners
2) create a timer object and associate it with listeners.

   `public` **`Timer`**`(int delay, ActionListener listener)`

   The constructor initializes the timer with a delay and one listener so that this Timer will notify the listener every delay milliseconds. If you want to add more listeners to this Timer, use the `addActionListener` method of the Timer. All the listeners registered with this Timer will be called after the same delay given by the constructor.

3) activate the Timer by calling start() on the timer object.

Example

```
/**
  This program shows a clock that is updated once per second.
*/
public class TimerTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();

      final int FIELD_WIDTH = 20;
      final JTextField textField = new JTextField(FIELD_WIDTH);

      frame.setLayout(new FlowLayout());
      frame.add(textField);
```

```
      ActionListener listener = new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Date now = new Date();
               textField.setText(now.toString());
            }
         };
      final int DELAY = 1000;

   // Milliseconds between timer ticks

      Timer t = new Timer(DELAY, listener);
      t.start();

          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

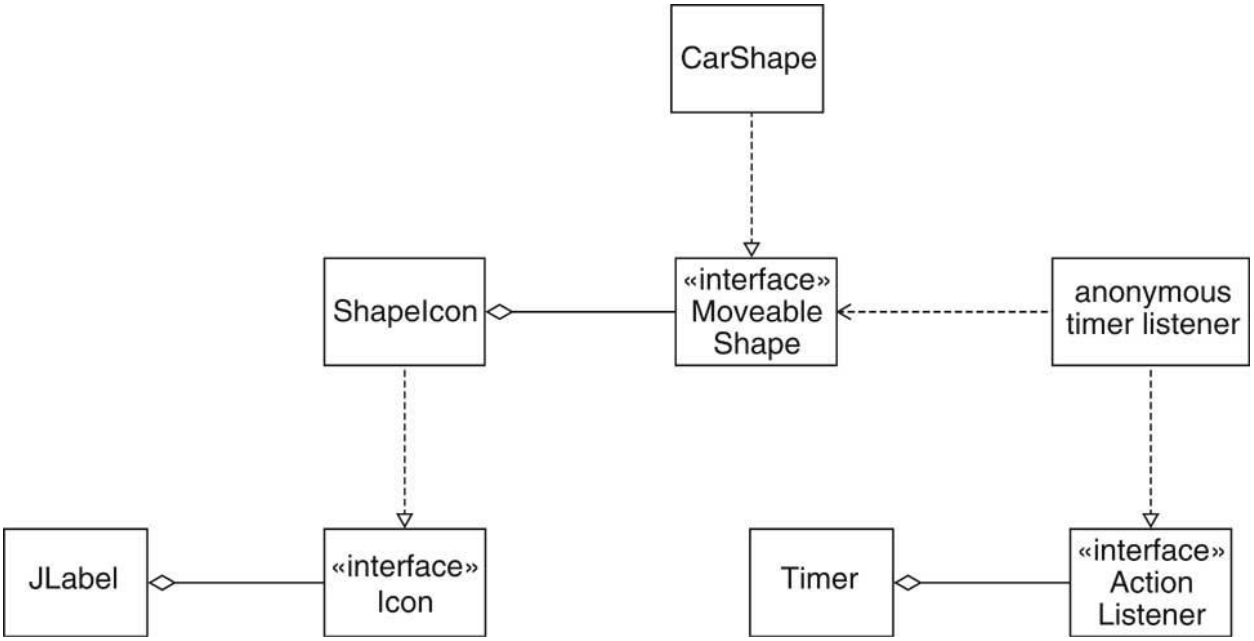
4.8.2 Defining a New Interface Type

We want to define an interface for any shape that can be movable. The interface defines two methods

- to draw shape
- to move it by a given amount shape

```
public interface MoveableShape
{
   void draw(Graphics2D g2);
   void translate(int dx, int dy);
}
```

By defining such an interface and writing an animation program in terms of MoveableShape, we can decouple the animation program from any specific shape such as CarShape. That is, the same program can be used to animate any shape as long as it is a MoveableShape.



4.8.3 Implementing the Animation

- Ch4/animation/MoveableShape.java
- Ch4/animation/ShapeIcon.java
- Ch4/animation/AnimationTester.java
- Ch4/animation/CarShape.java

```
import java.awt.*;

/**
   A shape that can be moved around.
*/
public interface MoveableShape
{
   /**
      Draws the shape.
      @param g2 the graphics context
   */
   void draw(Graphics2D g2);
   /**
      Moves the shape by a given amount.
      @param dx the amount to translate in x-direction
      @param dy the amount to translate in y-direction
   */
   void translate(int dx, int dy);
}

import java.awt.*;
import java.util.*;
import javax.swing.*;
```

```
/**
   An icon that contains a moveable shape.
*/
public class ShapeIcon implements Icon
{  private int width;
   private int height;
   private MoveableShape shape;

   public ShapeIcon(MoveableShape shape,int width, int height)
   {
      this.shape = shape;
      this.width = width;
      this.height = height;
   }

   public int getIconWidth()
   {  return width; }

   public int getIconHeight()
   {  return height; }

   public void paintIcon(Component c, Graphics g, int x, int y)
   {
      Graphics2D g2 = (Graphics2D) g;
      shape.draw(g2);
   }


}
public class CarShape implements MoveableShape
{    private int x;
     private int y;
     private int width;

   public void translate(int dx, int dy)
   {  x += dx;
      y += dy;
   }

   public void draw(Graphics2D g2)
   {
   // Create the parts of this car and draw them here.

      Rectangle2D.Double body
            = new Rectangle2D.Double(x, y + width / 6,
                  width - 1, width / 6);
      Ellipse2D.Double frontTire
            = new Ellipse2D.Double(x + width / 6, y + width / 3,
```

```
                  width / 6, width / 6);

. . .

   }
}


public class AnimationTester
{  private static final int ICON_WIDTH = 400;
   private static final int ICON_HEIGHT = 100;
   private static final int CAR_WIDTH = 100;

   public static void main(String[] args)
   {  JFrame frame = new JFrame();
      final MoveableShape shape = new CarShape(0, 0, CAR_WIDTH);
      ShapeIcon icon = new ShapeIcon(shape,ICON_WIDTH, ICON_HEIGHT);

      final JLabel label = new JLabel(icon);
      frame.setLayout(new FlowLayout());
      frame.add(label);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);

      final int DELAY = 100;
      // Milliseconds between timer ticks
      Timer t = new Timer(DELAY, new
         ActionListener()
         {  public void actionPerformed(ActionEvent event)
            {  shape.translate(1, 0);
               label.repaint();
            }
         });
      t.start();
   }
}
```

Note that there can be several approaches to achieve the same goal. In the following example, the car shape is drawn on the panel, and the panel is repainted each time the car moves.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
   This program implements an animation that moves a car shape.
*/
class MyPanel extends JPanel
{  MoveableShape s;
```

```
   public MyPanel(MoveableShape m)
   {  s = m; }
   public void paintComponent(Graphics g)
   {    super.paintComponent(g);
        s.draw((Graphics2D)g);
   }
}
public class AnimationTester
{
   public static void main(String[] args)
   {
      JFrame frame = new JFrame();
      int DEFAULT_WIDTH = 400;
      int DEFAULT_HEIGHT = 400;
      frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      int CAR_WIDTH = 100;
      final MoveableShape shape = new CarShape(0, 0, CAR_WIDTH);
      final JPanel panel = new MyPanel(shape);
      frame.add(panel);

      final int DELAY = 100;
      // Milliseconds between timer ticks
      Timer t = new Timer(DELAY, new
         ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               shape.translate(1, 0);
               panel.repaint();
            }
         });
      t.start();

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```