GUI Components: Part 2

Introduction to Layout Managers

- There are three ways for you to arrange components in a GUI:
  - Absolute positioning
    - Greatest level of control.
    - Set `Container`’s layout to `null`.
    - Specify the absolute position of each GUI component with respect to the upper-left corner of the `Container` by using `Component` methods `setSize` and `setLocation` or `setBounds`.
    - Must specify each GUI component’s size.

Introduction to Layout Managers (cont.)

- Layout managers
  - Simpler and faster than absolute positioning.
  - Lose some control over the size and the precise positioning of GUI components.
- Visual programming in an IDE
  - Use tools that make it easy to create GUIs.
  - Allows you to drag and drop GUI components from a tool box onto a design area.
  - You can then position, size and align GUI components as you like.

Introduction to Layout Managers (cont.)

- Layout managers arrange GUI components in a container for presentation purposes
- Can use for basic layout capabilities
- Enable you to concentrate on the basic look-and-feel—the layout manager handles the layout details.
- Layout managers implement interface LayoutManager (in package `java.awt`).
- `Container`'s `setLayout` method takes an object that implements the `LayoutManager` interface as an argument.

Layout manager	Description
`FlowLayout`	Default for `javax.swing.JPanel`. Places components sequentially (left to right) in the order they were added. It's also possible to specify the order of the components by using the `Container` method `add`, which takes a `Component` and an integer index position as arguments.
`BorderLayout`	Default for `JFrame`s (and other windows). Arranges the components into five areas: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
`GridLayout`	Arranges the components into rows and columns.

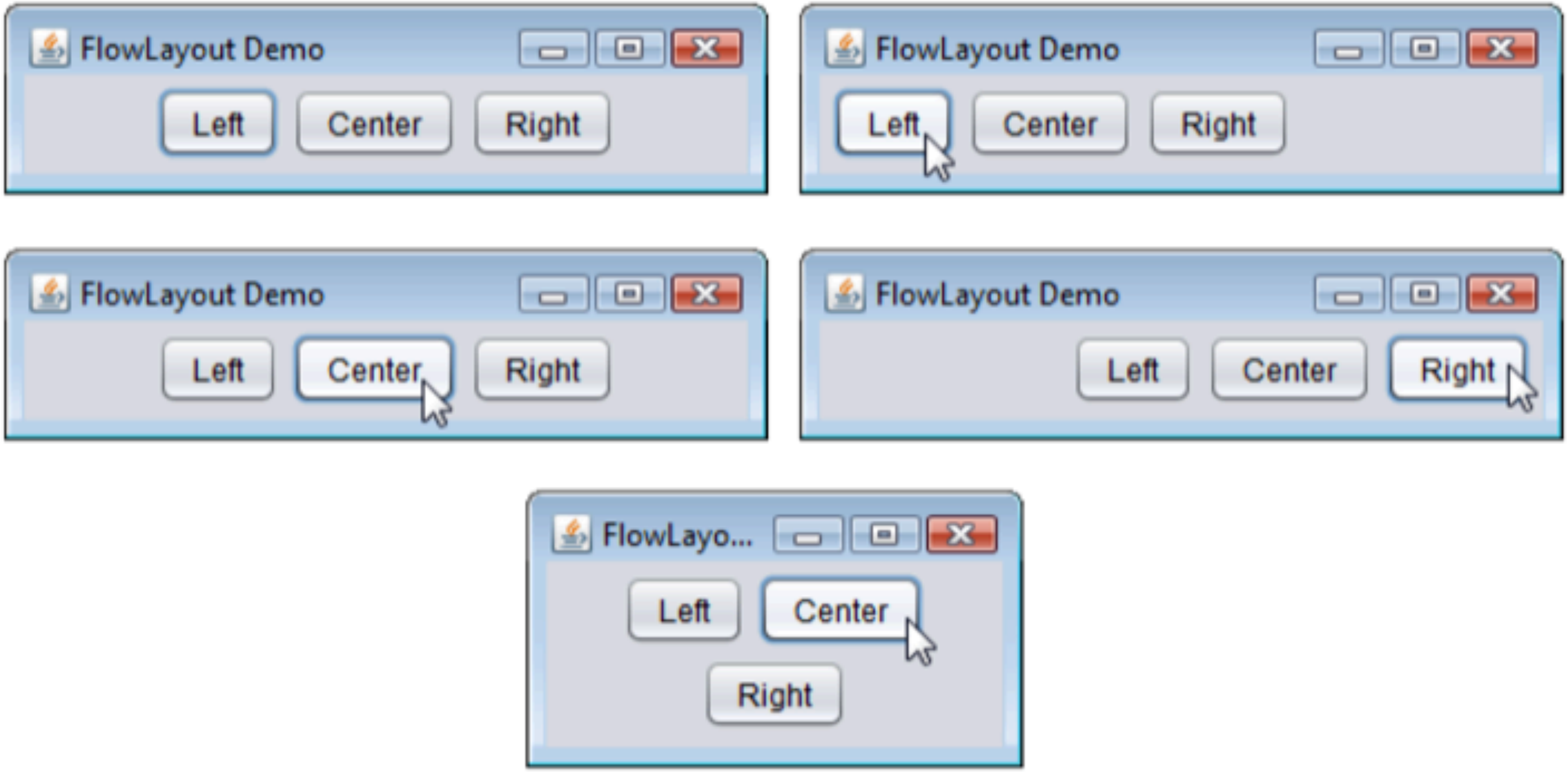
`FlowLayout`

- `FlowLayout` is the simplest layout manager.
- GUI components placed from left to right in the order in which they are added to the container.
- When the edge of the container is reached, components continue to display on the next line.
- `FlowLayout` allows GUI components to be left aligned, centered (the default) and right aligned.

```java
// Fig. 14.39: FlowLayoutFrame.java
// Demonstrating FlowLayout alignments.
import java.awt.FlowLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class FlowLayoutFrame extends JFrame
{
   private JButton leftJButton; // button to set alignment left
   private JButton centerJButton; // button to set alignment center
   private JButton rightJButton; // button to set alignment right
   private FlowLayout layout; // layout object
   private Container container; // container to set layout

   // set up GUI and register button listeners
   public FlowLayoutFrame()
   {
      super( "FlowLayout Demo" );

      layout = new FlowLayout(); // create FlowLayout
      container = getContentPane(); // get container to layout
      setLayout( layout ); // set frame layout

      // set up leftJButton and register listener
      leftJButton = new JButton( "Left" ); // create Left button
      add( leftJButton ); // add Left button to frame
      leftJButton.addActionListener(

         new ActionListener() // anonymous inner class
         {
            // process leftJButton event
            public void actionPerformed( ActionEvent event )
            {
               layout.setAlignment( FlowLayout.LEFT );

               // realign attached components
               layout.layoutContainer( container );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener

      // set up centerJButton and register listener
      centerJButton = new JButton( "Center" ); // create Center button
      add( centerJButton ); // add Center button to frame
      centerJButton.addActionListener(

         new ActionListener() // anonymous inner class
         {
            // process centerJButton event
            public void actionPerformed( ActionEvent event )
            {
               layout.setAlignment( FlowLayout.CENTER );

               // realign attached components
               layout.layoutContainer( container );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener

```

```java
      // set up rightJButton and register listener
      rightJButton = new JButton( "Right" ); // create Right button
      add( rightJButton ); // add Right button to frame
      rightJButton.addActionListener(

         new ActionListener() // anonymous inner class
         {
            // process rightJButton event
            public void actionPerformed( ActionEvent event )
            {
               layout.setAlignment( FlowLayout.RIGHT );

               // realign attached components
               layout.layoutContainer( container );
            } // end method actionPerformed
         } // end anonymous inner class
      ); // end call to addActionListener
   } // end FlowLayoutFrame constructor
} // end class FlowLayoutFrame
```

```java
// Fig. 14.40: FlowLayoutDemo.java
// Testing FlowLayoutFrame.
import javax.swing.JFrame;

public class FlowLayoutDemo
{
   public static void main( String[] args )
   {
      FlowLayoutFrame flowLayoutFrame = new FlowLayoutFrame();
      flowLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      flowLayoutFrame.setSize( 300, 75 ); // set frame size
      flowLayoutFrame.setVisible( true ); // display frame
   } // end main
} // end class FlowLayoutDemo
```


FlowLayout Demo
Left
Center
Right
FlowLayout Demo
Left
Center
Right
FlowLayout Demo
Left
Center
Right
FlowLayout Demo
Left
Center
Right
FlowLayo...
Left
Center
Right

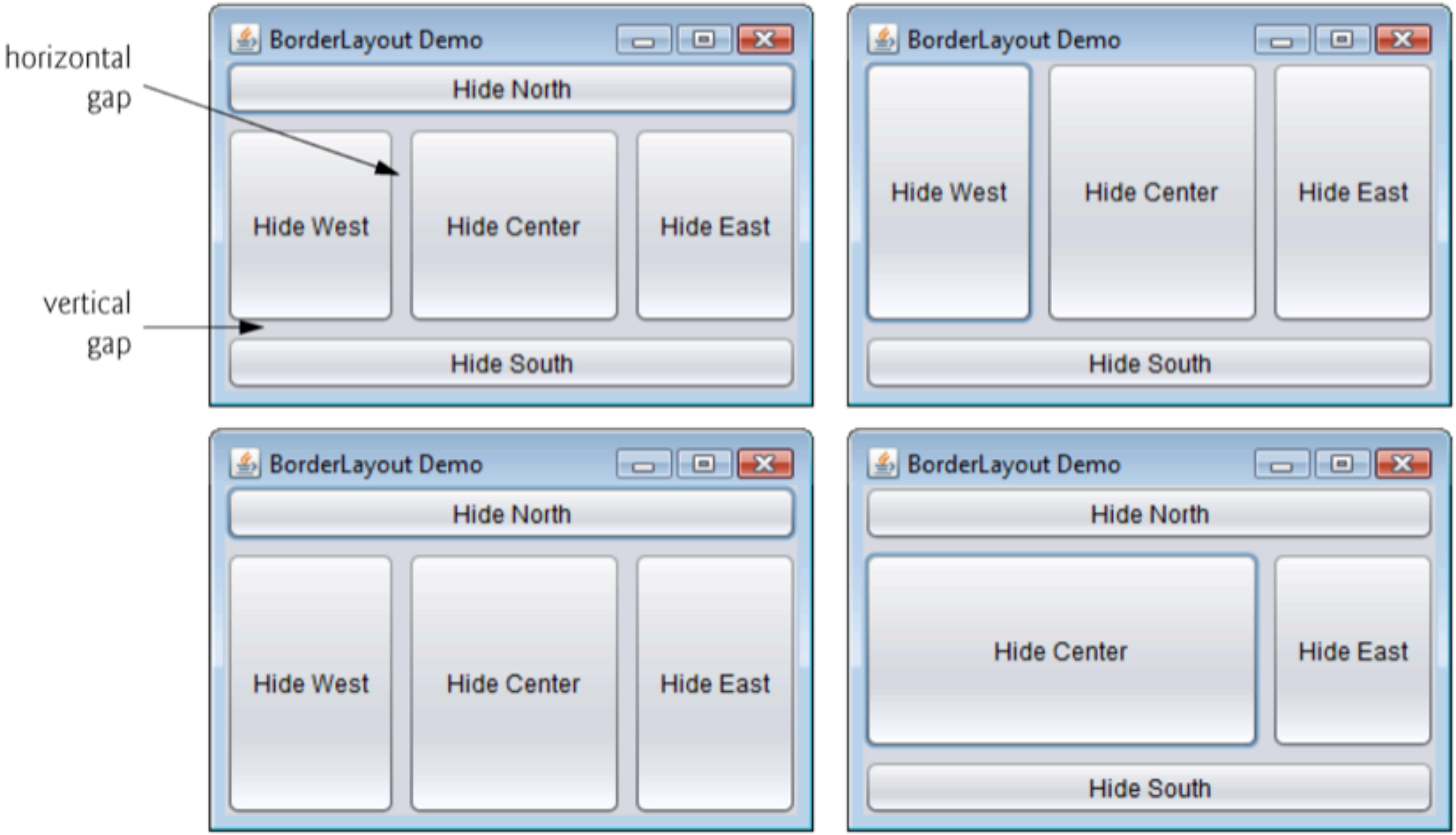
BorderLayout

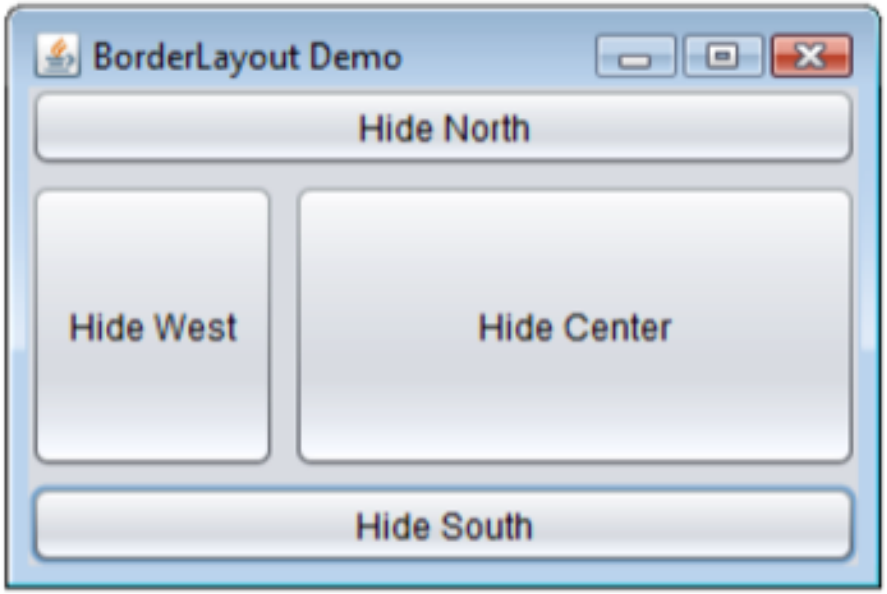
- `BorderLayout`
  - the default layout manager for a `Jframe`
  - arranges components into five regions: `NORTH`, `SOUTH`, `EAST`, `WEST` and `CENTER`.
  - `NORTH` corresponds to the top of the container.

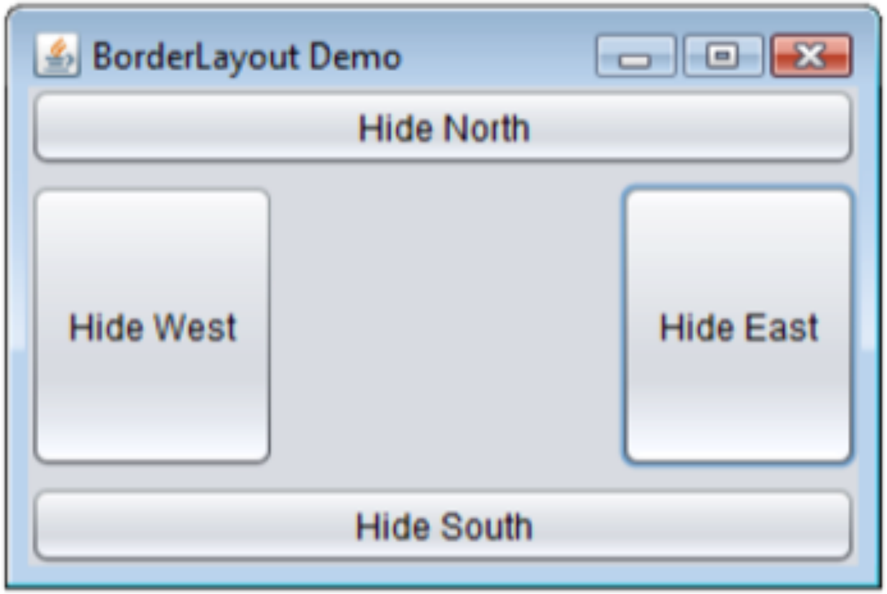
- `BorderLayout` limits a `Container` to at most five components—one in each region.
  - The component placed in each region can be a container to which other components are attached.

```java
// Fig. 14.41: BorderLayoutFrame.java
// Demonstrating BorderLayout.
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class BorderLayoutFrame extends JFrame implements ActionListener
{
   private JButton[] buttons; // array of buttons to hide portions
   private static final String[] names = { "Hide North", "Hide South",
      "Hide East", "Hide West", "Hide Center" };
   private BorderLayout layout; // borderlayout object

   // set up GUI and event handling
   public BorderLayoutFrame()
   {
      super( "BorderLayout Demo" );

      layout = new BorderLayout( 5, 5 ); // 5 pixel gaps
      setLayout( layout ); // set frame layout
      buttons = new JButton[ names.length ]; // set size of array

      // create JButtons and register listeners for them
      for ( int count = 0; count < names.length; count++ )
      {
         buttons[ count ] = new JButton( names[ count ] );
         buttons[ count ].addActionListener( this );
      } // end for

      add( buttons[ 0 ], BorderLayout.NORTH ); // add button to north
      add( buttons[ 1 ], BorderLayout.SOUTH ); // add button to south
      add( buttons[ 2 ], BorderLayout.EAST ); // add button to east
      add( buttons[ 3 ], BorderLayout.WEST ); // add button to west
      add( buttons[ 4 ], BorderLayout.CENTER ); // add button to center
   } // end BorderLayoutFrame constructor

   // handle button events
   public void actionPerformed( ActionEvent event )
   {
      // check event source and lay out content pane correspondingly
      for ( JButton button : buttons )
      {
         if ( event.getSource() == button )
            button.setVisible( false ); // hide button clicked
         else
            button.setVisible( true ); // show other buttons
      } // end for

      layout.layoutContainer( getContentPane() ); // lay out content pane
   } // end method actionPerformed
} // end class BorderLayoutFrame
```

```java
// Fig. 14.42: BorderLayoutDemo.java
// Testing BorderLayoutFrame.
import javax.swing.JFrame;

public class BorderLayoutDemo
{
   public static void main( String[] args )
   {
      BorderLayoutFrame borderLayoutFrame = new BorderLayoutFrame();
      borderLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      borderLayoutFrame.setSize( 300, 200 ); // set frame size
      borderLayoutFrame.setVisible( true ); // display frame
   } // end main
} // end class BorderLayoutDemo
```


horizontal gap
vertical gap
BorderLayout Demo
Hide North
Hide West
Hide Center
Hide East
Hide South
BorderLayout Demo
Hide West
Hide Center
Hide East
Hide South
BorderLayout Demo
Hide North
Hide West
Hide Center
Hide East
BorderLayout Demo
Hide North
Hide Center
Hide East
Hide South


BorderLayout Demo
Hide North
Hide West
Hide Center
Hide South


BorderLayout Demo
Hide North
Hide West
Hide East
Hide South

`BorderLayout` (cont.)

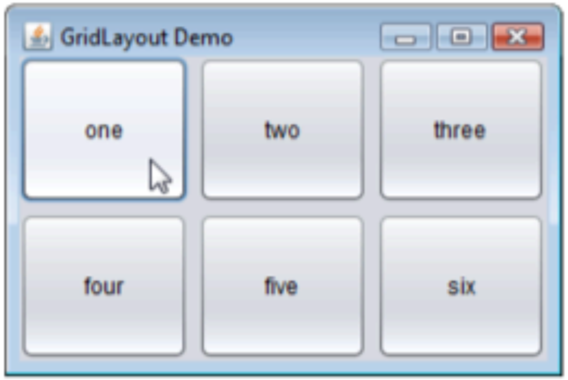
- `BorderLayout` constructor arguments specify the number of pixels between components that are arranged horizontally (horizontal gap space) and between components that are arranged vertically (vertical gap space), respectively.
  - The default is one pixel of gap space horizontally and vertically.

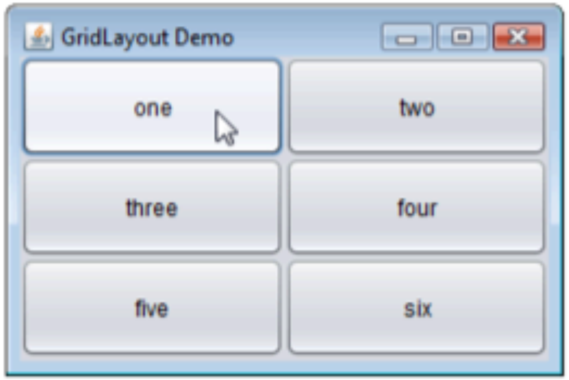
GridLayout

- GridLayout divides the container into a grid of rows and columns.
  - Every Component has the same width and height.
  - Components are added starting at the top-left cell of the grid and proceeding left to right until the row is full. Then the process continues left to right on the next row of the grid, and so on.

```java
// Fig. 14.43: GridLayoutFrame.java
// Demonstrating GridLayout.
import java.awt.GridLayout;
import java.awt.Container;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JFrame;
import javax.swing.JButton;

public class GridLayoutFrame extends JFrame implements ActionListener
{
   private JButton[] buttons; // array of buttons
   private static final String[] names =
      { "one", "two", "three", "four", "five", "six" };
   private boolean toggle = true; // toggle between two layouts
   private Container container; // frame container
   private GridLayout gridLayout1; // first gridlayout
   private GridLayout gridLayout2; // second gridlayout

   // no-argument constructor
   public GridLayoutFrame()
   {
      super( "GridLayout Demo" );
      gridLayout1 = new GridLayout( 2, 3, 5, 5 ); // 2 by 3; gaps of 5
      gridLayout2 = new GridLayout( 3, 2 ); // 3 by 2; no gaps
      container = getContentPane(); // get content pane
      setLayout( gridLayout1 ); // set JFrame layout
      buttons = new JButton[ names.length ]; // create array of JButtons

      for ( int count = 0; count < names.length; count++ )
      {
         buttons[ count ] = new JButton( names[ count ] );
         buttons[ count ].addActionListener( this ); // register listener
         add( buttons[ count ] ); // add button to JFrame
      } // end for
   } // end GridLayoutFrame constructor

   // handle button events by toggling between layouts
   public void actionPerformed( ActionEvent event )
   {
      if ( toggle )
         container.setLayout( gridLayout2 ); // set layout to second
      else
         container.setLayout( gridLayout1 ); // set layout to first

      toggle = !toggle; // set toggle to opposite value
      container.validate(); // re-lay out container
   } // end method actionPerformed
} // end class GridLayoutFrame
```

```java
// Fig. 14.44: GridLayoutDemo.java
// Testing GridLayoutFrame.
import javax.swing.JFrame;

public class GridLayoutDemo
{
   public static void main( String[] args )
   {
      GridLayoutFrame gridLayoutFrame = new GridLayoutFrame();
      gridLayoutFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      gridLayoutFrame.setSize( 300, 200 ); // set frame size
      gridLayoutFrame.setVisible( true ); // display frame
   } // end main
} // end class GridLayoutDemo
```





Using Panels to Manage More Complex Layouts

- Complex GUIs require that each component be placed in an exact location.
  - Often consist of multiple panels, with each panel' s components arranged in a specific layout.

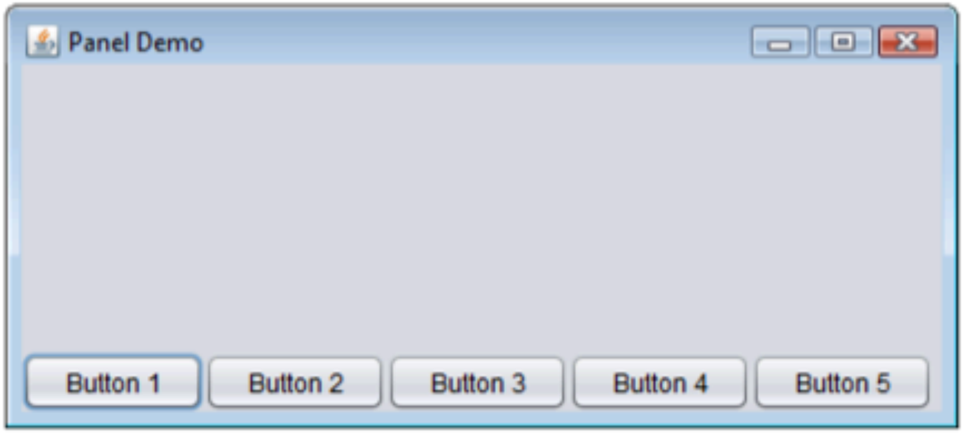
- Class `JPanel` extends `JComponent` and `JComponent` extends class `Container`, so every `JPanel` is a `Container`.

- Every `JPanel` may have components, including other panels, attached to it with `Container` method `add`.

- `JPanel` can be used to create a more complex layout in which several components are in a specific area of another container.

```java
// Fig. 14.45: PanelFrame.java
// Using a JPanel to help lay out components.
import java.awt.GridLayout;
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JButton;

public class PanelFrame extends JFrame
{
   private JPanel buttonJPanel; // panel to hold buttons
   private JButton[] buttons; // array of buttons

   // no-argument constructor
   public PanelFrame()
   {
      super( "Panel Demo" );
      buttons = new JButton[ 5 ]; // create buttons array
      buttonJPanel = new JPanel(); // set up panel
      buttonJPanel.setLayout( new GridLayout( 1, buttons.length ) );

      // create and add buttons
      for ( int count = 0; count < buttons.length; count++ )
      {
         buttons[ count ] = new JButton( "Button " + ( count + 1 ) );
         buttonJPanel.add( buttons[ count ] ); // add button to panel
      } // end for

      add( buttonJPanel, BorderLayout.SOUTH ); // add panel to JFrame
   } // end PanelFrame constructor
} // end class PanelFrame
```

```java
// Fig. 14.46: PanelDemo.java
// Testing PanelFrame.
import javax.swing.JFrame;

public class PanelDemo extends JFrame
{
   public static void main( String[] args )
   {
      PanelFrame panelFrame = new PanelFrame();
      panelFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      panelFrame.setSize( 450, 200 ); // set frame size
      panelFrame.setVisible( true ); // display frame
   } // end main
} // end class PanelDemo
```



`JPanel` Subclass for Drawing with the Mouse

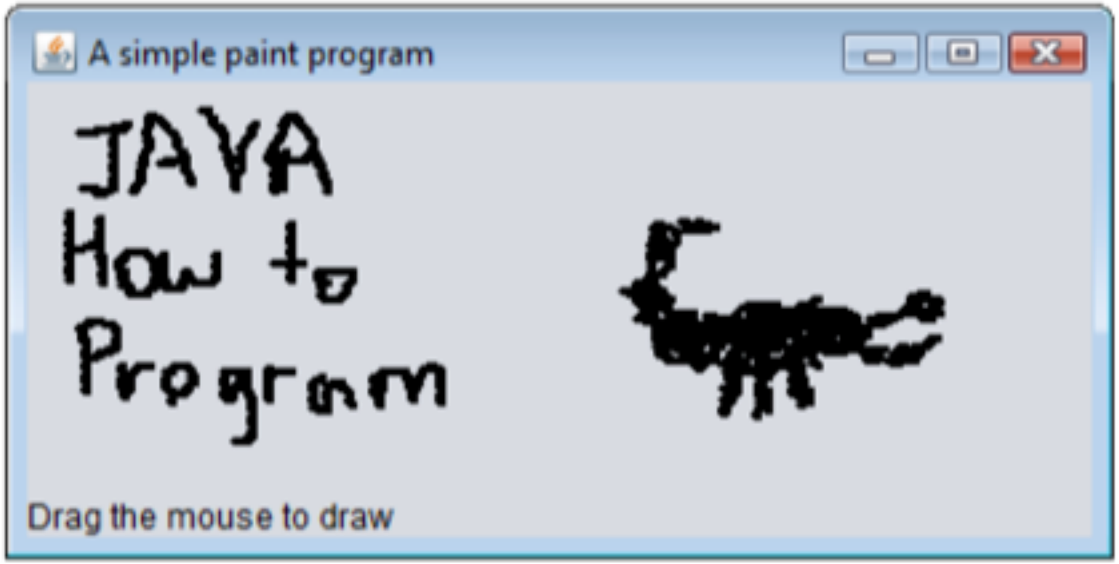
- Use a `JPanel` as a dedicated drawing area in which the user can draw by dragging the mouse.

- Lightweight Swing components that extend class `JComponent` (such as `JPanel`) contain method paintComponent
    - called when a lightweight Swing component is displayed

- Override this method to specify how to draw.

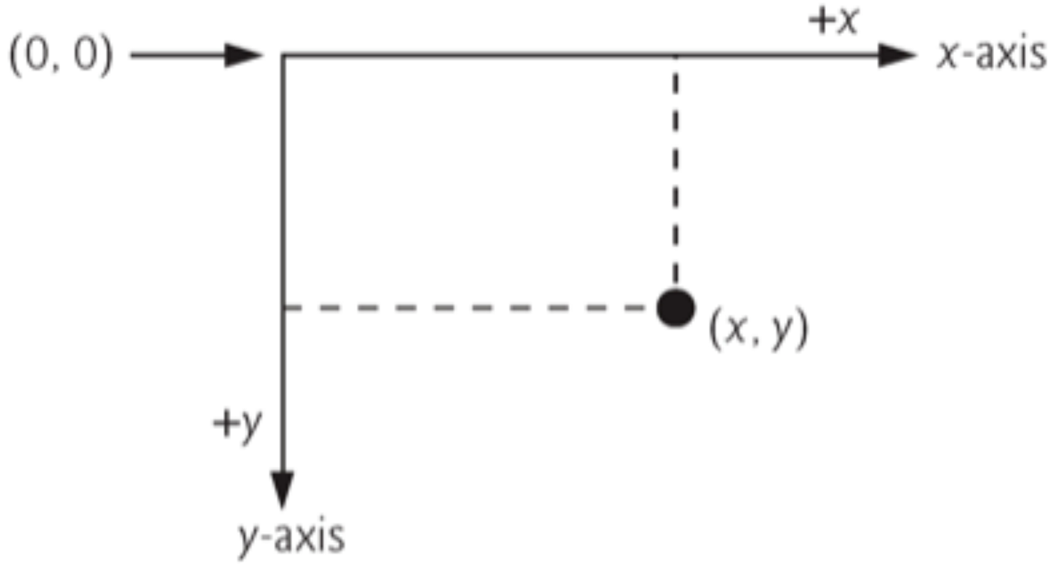
```java
// Fig. 14.34: PaintPanel.java
// Using class MouseMotionAdapter.
import java.awt.Point;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
import javax.swing.JPanel;

public class PaintPanel extends JPanel
{
   private int pointCount = 0; // count number of points

   // array of 10000 java.awt.Point references
   private Point[] points = new Point[ 10000 ];

   // set up GUI and register mouse event handler
   public PaintPanel()
   {
      // handle frame mouse motion event
      addMouseMotionListener(

         new MouseMotionAdapter() // anonymous inner class
         {
            // store drag coordinates and repaint
            public void mouseDragged( MouseEvent event )
            {
               if ( pointCount < points.length )
               {
                  points[ pointCount ] = event.getPoint(); // find point
                  ++pointCount; // increment number of points in array
                  repaint(); // repaint JFrame
               } // end if
            } // end method mouseDragged
         } // end anonymous inner class
      ); // end call to addMouseMotionListener
   } // end PaintPanel constructor

   // draw ovals in a 4-by-4 bounding box at specified locations on window
   public void paintComponent( Graphics g )
   {
      super.paintComponent( g ); // clears drawing area

      // draw all points in array
      for ( int i = 0; i < pointCount; i++ )
         g.fillOval( points[ i ].x, points[ i ].y, 4, 4 );
   } // end method paintComponent
} // end class PaintPanel
```

```java
// Fig. 14.35: Painter.java
// Testing PaintPanel.
import java.awt.BorderLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class Painter
{
   public static void main( String[] args )
   {
      // create JFrame
      JFrame application = new JFrame( "A simple paint program" );

      PaintPanel paintPanel = new PaintPanel(); // create paint panel
      application.add( paintPanel, BorderLayout.CENTER ); // in center

      // create a label and place it in SOUTH of BorderLayout
      application.add( new JLabel( "Drag the mouse to draw" ),
         BorderLayout.SOUTH );

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.setSize( 400, 200 ); // set frame size
      application.setVisible( true ); // display frame
   } // end main
} // end class Painter
```



A simple paint program

Drag the mouse to draw

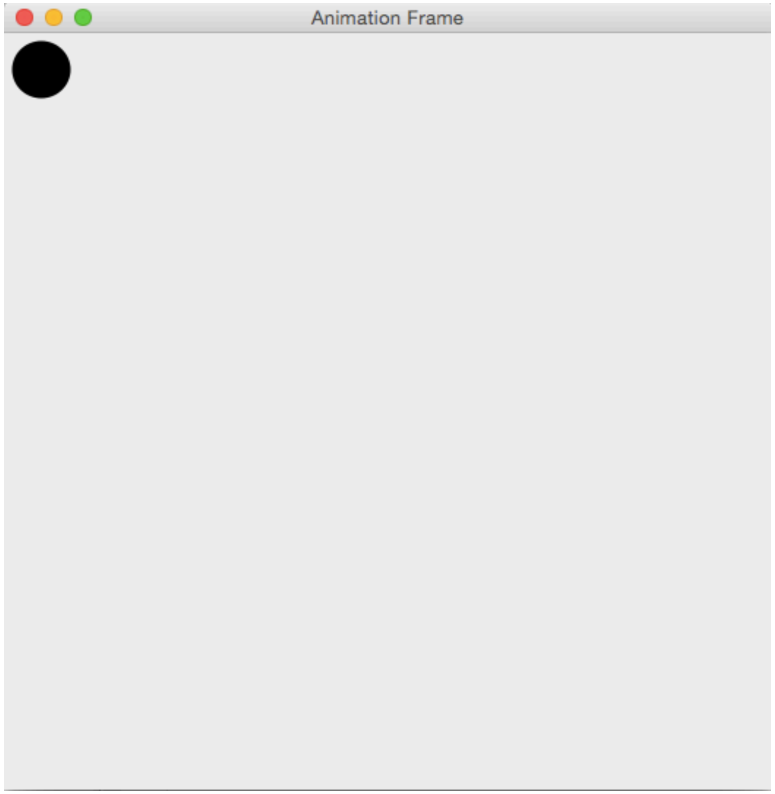


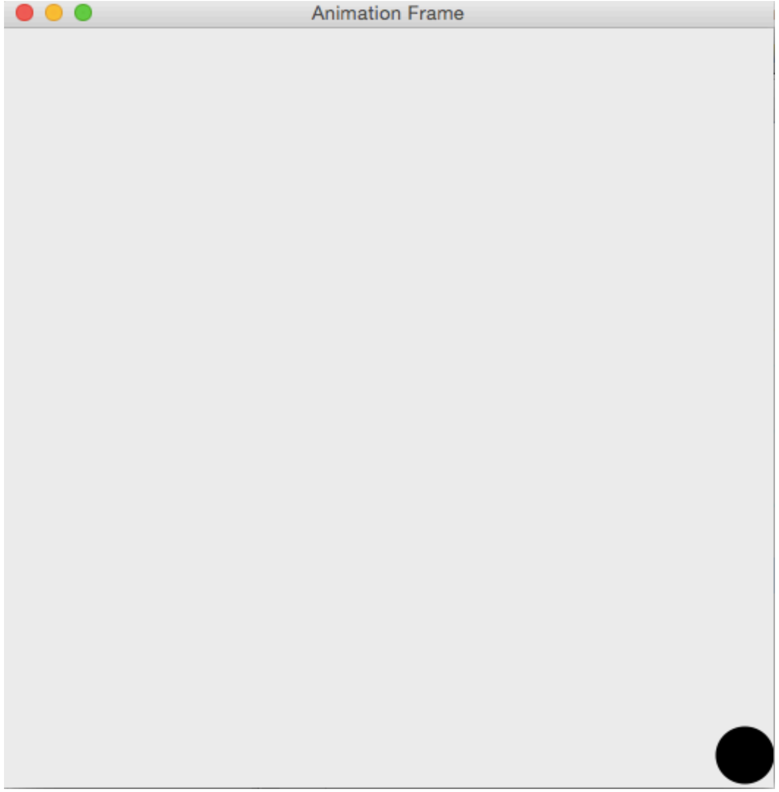
Java coordinate system. Units are measured in pixels.

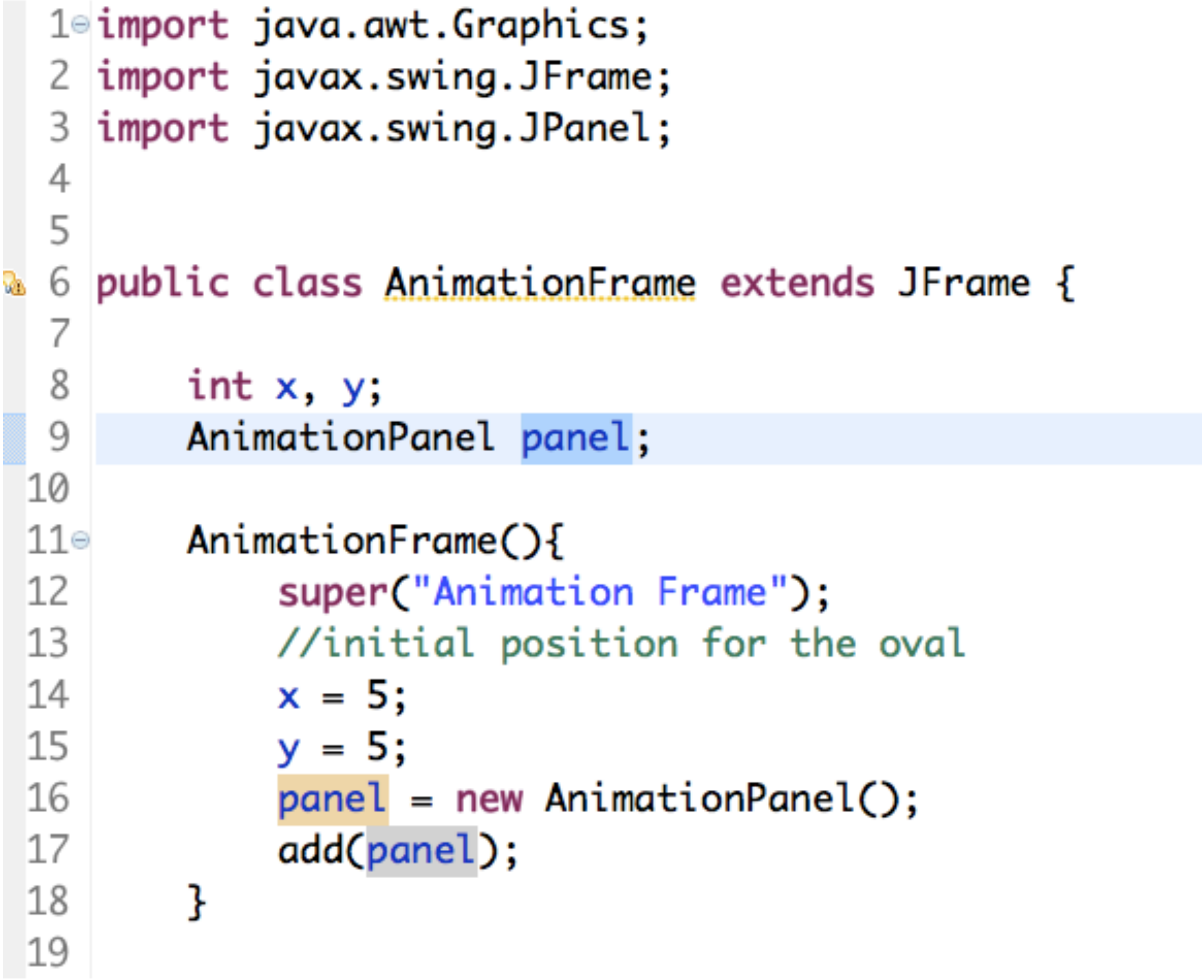
Java Graphics (cont.)

- Class Graphics contains methods for drawing strings, lines, rectangles and other shapes.
- Class Graphics2D, which extends class `Graphics`, is used for drawing with the Java 2D API.
- Class Color contains methods and constants for manipulating colors.
- Class Font contains methods and constants for manipulating fonts.
- Class FontMetrics contains methods for obtaining font information.

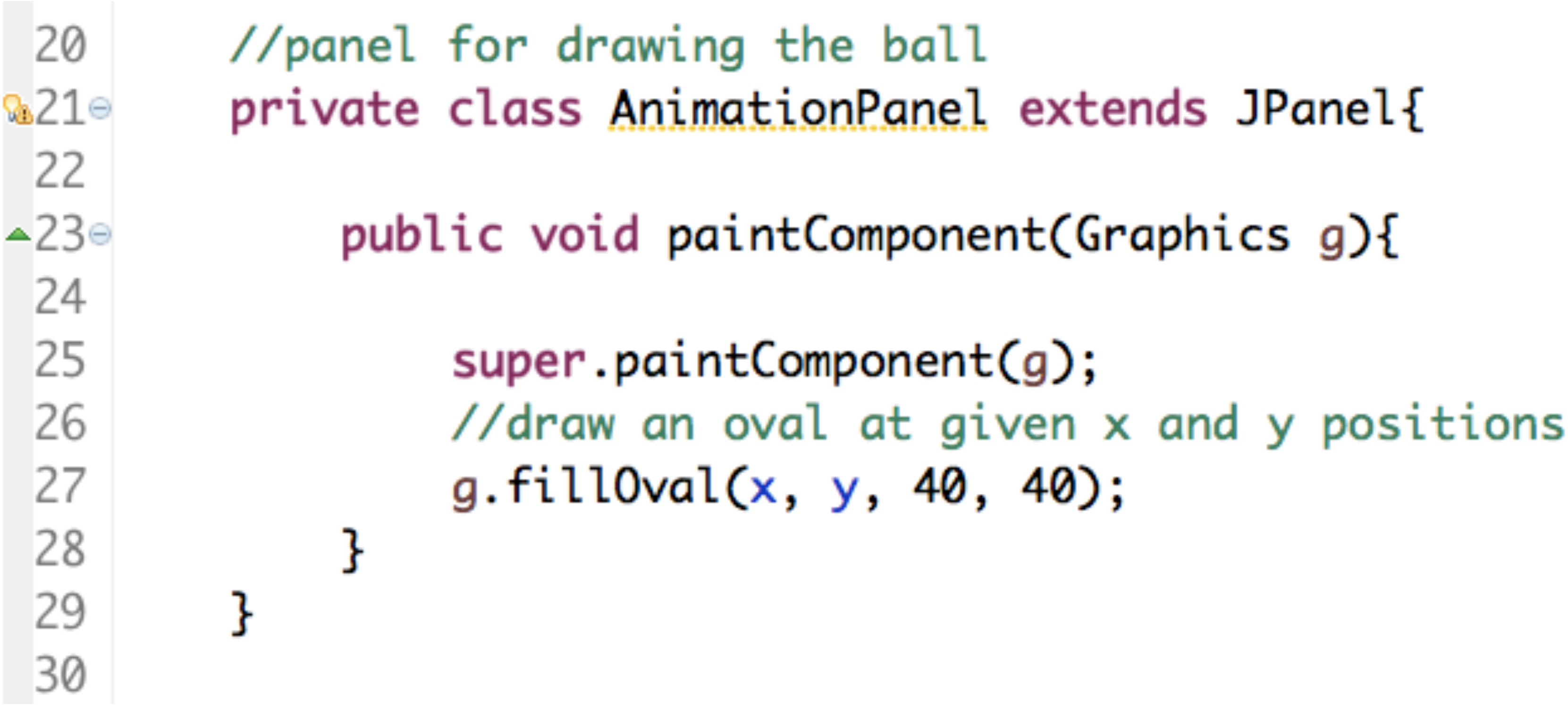
Simple Animation



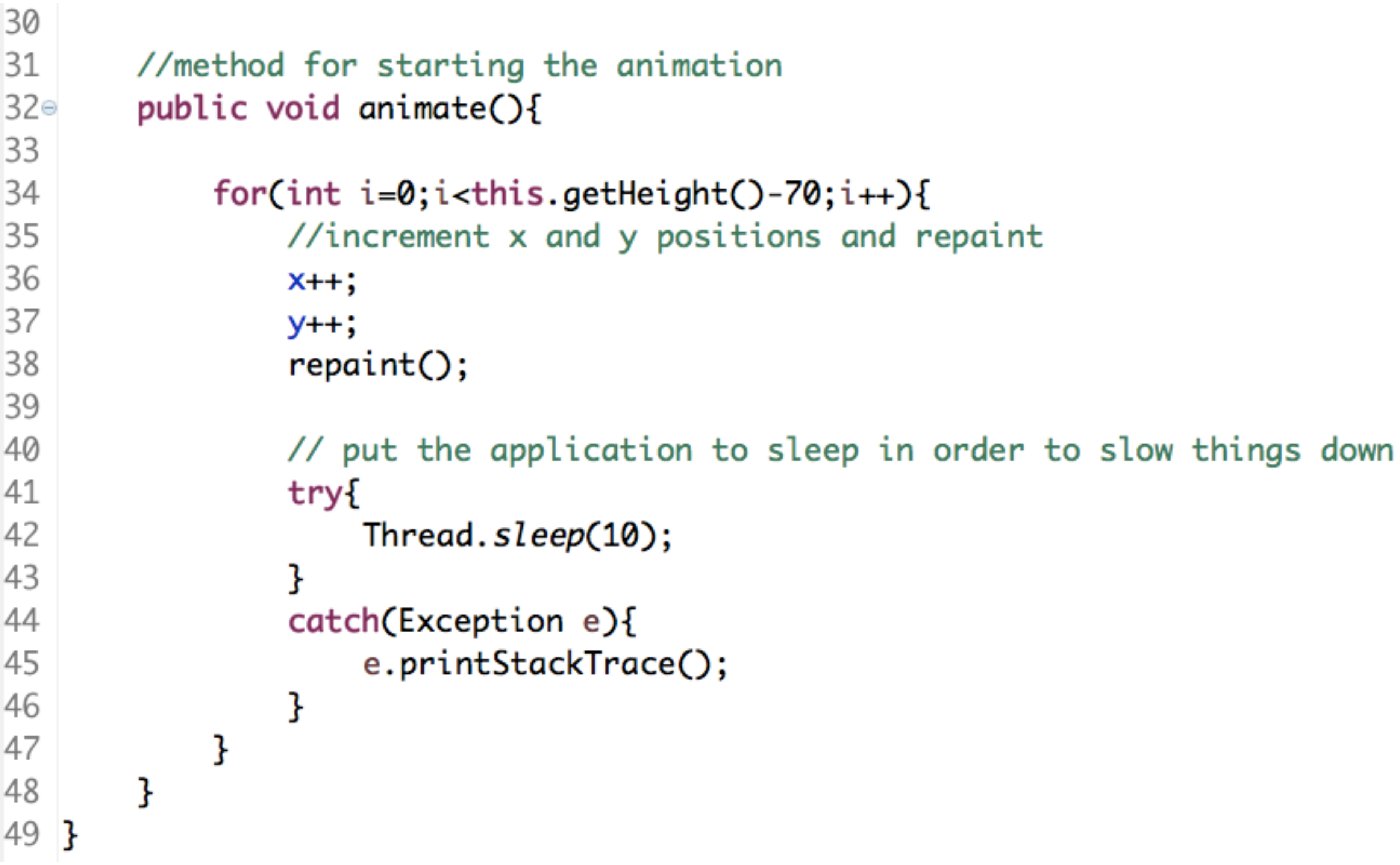




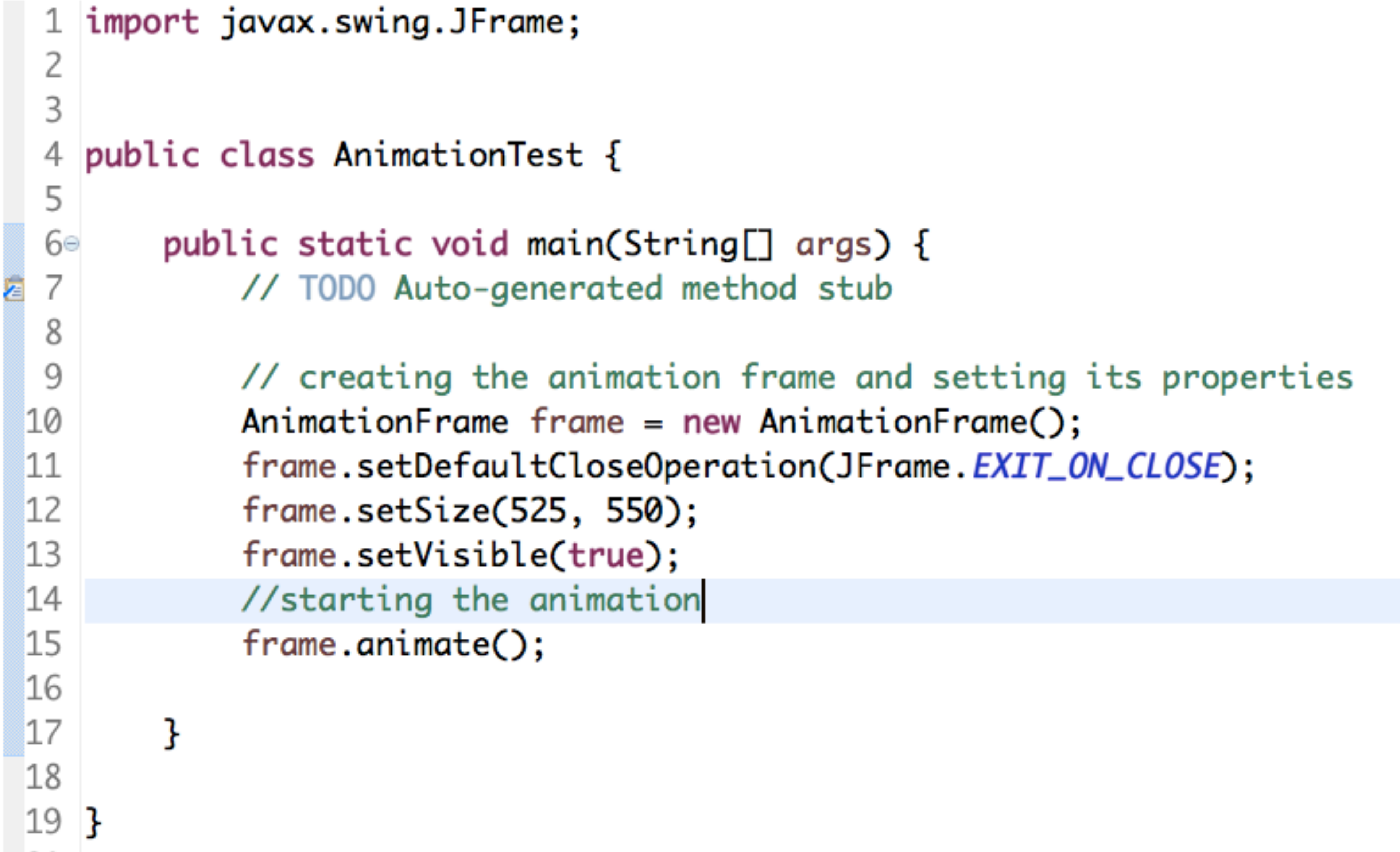
```java
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JPanel;


public class AnimationFrame extends JFrame {

    int x, y;
    AnimationPanel panel;

    AnimationFrame(){
        super("Animation Frame");
        //initial position for the oval
        x = 5;
        y = 5;
        panel = new AnimationPanel();
        add(panel);
    }

```



```java
    //panel for drawing the ball
    private class AnimationPanel extends JPanel{

        public void paintComponent(Graphics g){

            super.paintComponent(g);
            //draw an oval at given x and y positions
            g.fillOval(x, y, 40, 40);
        }
    }

```

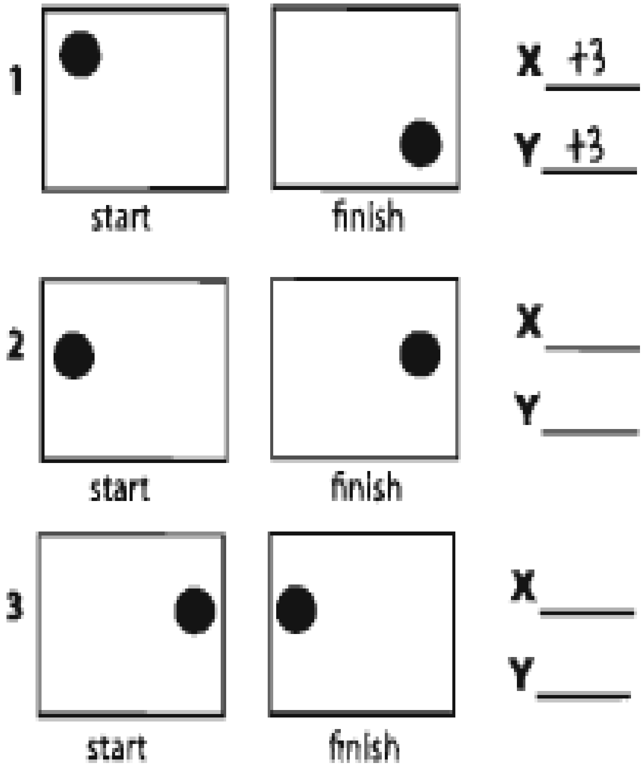


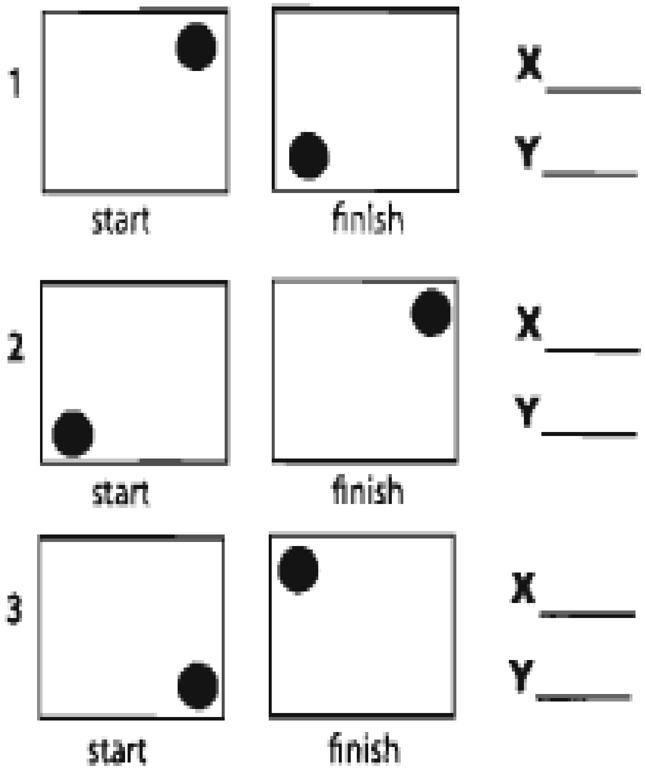
```java

    //method for starting the animation
    public void animate(){

        for(int i=0;i<this.getHeight()-70;i++){
            //increment x and y positions and repaint
            x++;
            y++;
            repaint();

            // put the application to sleep in order to slow things down
            try{
                Thread.sleep(10);
            }
            catch(Exception e){
                e.printStackTrace();
            }
        }
    }
}
```



```java
import javax.swing.JFrame;


public class AnimationTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        // creating the animation frame and setting its properties
        AnimationFrame frame = new AnimationFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(525, 550);
        frame.setVisible(true);
        //starting the animation
        frame.animate();

    }

}
```

- A little exercise


1
start
finish
X +3
Y +3
2
start
finish
X
Y
3
start
finish
X
Y


1
start
finish
X
Y
2
start
finish
X
Y
3
start
finish
X
Y