

Chapter 6: Making dynamic illustrations

Up to this point, we have learned how to use Java2D to create static mathematical illustrations. However, Java provides some tools that make it quite easy to create illustrations that users can reconfigure. In this chapter, we'll look at a few ways to create dynamic illustrations.

1: Using Buttons

Perhaps the simplest way to allow the viewer to modify the diagram is by adding a button that, when pressed, presents another, related diagram. To do this, we need to add two features to our code: the button that is meant to be pressed and a method that causes the redrawing when the button is pressed.

Java Swing provides buttons through the class `javax.swing.JButton`. The easiest, and most common, way to instantiate a `JButton` is to give it a `String` that will be displayed on the button. For instance,

```
JButton move = new JButton("Move");
```

The second piece of the puzzle is a method that is executed when the button is pressed. In particular, we would like to tell the button to execute a particular method when the button is pressed. However, in Java, we cannot pass a method to the button directly. Instead, we pass the implementation of an interface, in this case, an implementation of the `java.awt.event.ActionListener` interface. This interface requires that one method be defined with the signature:

```
public void actionPerformed(ActionEvent event);
```

Let's put everything together now in an example. We will construct a diagram consisting of one rectangle. When the button is pressed, the upper left corner of the rectangle will slide to the right by

three pixels and down by two. Here's the code that does it:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
public class MovingRectangle extends JPanel implements ActionListener {
    double x = 50, y = 50;
    double dx = 3, dy = 2;
    double width = 100, height = 50;
    public MovingRectangle() {
        setBackground(Color.white);
    }
    public void actionPerformed(ActionEvent event) {
        x += dx; y += dy;
        repaint();
    }
    public void paintComponent(Graphics gfx) {
        super.paintComponent(gfx);
        Graphics2D g = (Graphics2D) gfx;
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        g.draw(new Rectangle2D.Double(x, y, width, height));
    }
    public static void main(String[] args) {
        DrawFrame frame = new DrawFrame("MovingRectangle");
        MovingRectangle mr = new MovingRectangle();
        mr.setPreferredSize(new Dimension(300, 200));
        frame.getContentPane().add(mr, BorderLayout.CENTER);
        JButton move = new JButton("Move");
        move.addActionListener(mr);
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(move);
        frame.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        frame.pack();
        frame.show();
    }
}
```

Notice that we define a class `MovingRectangle` that implements `ActionListener`. Two instance variables, `x` and `y`, hold the location of the upper left corner of the rectangle. The method `paintComponent` does nothing more than draw the rectangle at its current position. The method `actionPerformed` is noteworthy as this is the method that will be executed when the button is pressed. You will notice that this method simply increases `x` and `y` and asks that the diagram be redrawn through a call to the `repaint` method, which causes the `paintComponent` method to be executed.

In the method `main`, we instantiate a `JButton` and add it into our frame (more about that in a minute). We now need to tell the button where to find the method `actionPerformed` to execute. This

is done by calling the `addActionListener` method of the `JButton`. It is, of course, possible to add many `ActionListeners` to a given `JButton`.

When the button is pressed, it creates an instance of `java.awt.event.ActionEvent`. This object contains all the information about the event, in this case, a button press, that occurred. For instance, it can give us the `String` that labels the button as well as tell us, for example, whether the control key was held down at the time of the press. The `ActionEvent` is passed to the `ActionListener` in case any of this information is needed in processing the event. In our example, we don't need any of this information so the `ActionEvent` is ignored.

2: A few words about layout

Once we have instantiated our button, we need to specify where it is displayed in our frame. The first thing we do is to instantiate a new `JPanel`

```
JPanel buttonPanel = new JPanel();
```

into which we will put the button

```
buttonPanel.add(move);
```

Finally, we will include the panel containing the button in the frame.

```
frame.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
```

Java provides several types of containers to hold other components, the placement of which is controlled by the container's `LayoutManager`. By default, the frame's `LayoutManager` is an instance of `BorderLayout`, which can hold five components. One component may be placed in the center of the frame and the others along the top, bottom, left and right. To specify which component goes where, we use the constants `BorderLayout.CENTER`, `BorderLayout.SOUTH` and so on. In our example, we want the button to lie below the diagram so we use the `BorderLayout.SOUTH` constant when adding the panel containing the button to the frame.

You may ask why the button needs to be put into a panel before it is added to the frame. The short answer is that it doesn't need to be. However, if the button is added directly to the frame, it will be stretched so that it occupies the entire width of the frame. Generally, this does not look attractive. Placing the button in a panel and the panel in the frame causes the button to have a more appropriate size.

3: Adding another button

Our example that moves the rectangle across the screen is not so good: once the rectangle disappears, there is no way to get it back. We can correct this by adding a `Reset` button as in the following example.

```
public class MovingRectangleWithReset extends JPanel
    implements ActionListener {
    double x0 = 50, y0 = 50;
    double x, y;
    double dx = 3, dy = 2;
    double width = 100, height = 50;
    public MovingRectangleWithReset() {
        setBackground(Color.white);
        x = x0; y = y0;
    }
    public void actionPerformed(ActionEvent event) {
        String actionCommand = event.getActionCommand();
        if (actionCommand.equals("Move")) {
            x += dx; y += dy;
        }
        else if (actionCommand.equals("Reset")) {
            x = x0; y = y0;
        }
        repaint();
    }
    ...
    public static void main(String[] args) {
        DrawFrame frame = new DrawFrame("MovingRectangleWithReset");
        MovingRectangleWithReset mr = new MovingRectangleWithReset();
        mr.setPreferredSize(new Dimension(300, 200));
        mr.setBackground(Color.white);
        frame.getContentPane().add(mr, BorderLayout.CENTER);
        JButton move = new JButton("Move");
        move.addActionListener(mr);
        JButton reset = new JButton("Reset");
        reset.addActionListener(mr);
        JPanel buttonPanel = new JPanel();
        buttonPanel.add(move);
        buttonPanel.add(reset);
        frame.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
        frame.pack();
        frame.show();
    }
}
```

Notice that the `actionPerformed` method checks which button was pressed by asking the `ActionEvent` passed to it for the `String` that labels the button. It then takes the appropriate action.

Also, `JPanels` have, by default, a `FlowLayout` object to manage the way in which components are placed in them. This layout manager tries to arrange the components in a row from left to right as they are added.

Exercise 1: Illustrate the steps of the proof of Proposition I of Book I of *The Elements* as follows. Begin with an illustration consisting only of two points. Include buttons **Next**, **Previous** and **Reset**. When the **Next** button is pressed, the next step of the proof is displayed until the end is reached. Use the `setEnabled(boolean b)` method of `JButton` to disable the **Previous** button initially and the **Next** button at the conclusion of the proof.

Exercise 2: Create a program that graphs the function $f(x) = e^{-x} \sin(x)$ and a grid in the viewing window $-1 \leq x \leq 3$ and $-2 \leq y \leq 2$. Add a button that, when pressed, causes the point $(1, f(1))$ to be at the center of the viewing window with the scale on the horizontal and vertical axes halved. Be sure to include a **Reset** button. You may wish to allow the user to zoom in only so far using the button's `setEnabled(boolean b)` method.

Exercise 3: Create a program that demonstrates the behavior of the complex function $f(z) = z^2$. There should be two `FigurePanel`s: the `FigurePanel` on the left should be thought of as the domain of the function with a 1×1 grid graphed while the `FigurePanel` on the right should be thought of as the co-domain of the function in which the image of the grid is drawn. Add a button that, when pressed, causes both panels to zoom in around the point $z = 1 + i$ and $f(z) = 2i$. The scales of both panels should be halved and the spacing of the grid halved as well. (How does this demonstrate that $f'(1 + i) = 2 + 2i$?)

4: Handling mouse actions

A button is useful for allowing the viewer to change the illustration in very specific ways. However, there are times that we would like to accommodate a wider range of changes than those allowed only by a button. For instance, we may wish to allow the viewer to rearrange the position of some points in the plane. In this case, we need to be able to respond to events triggered when the mouse is clicked, dragged and released.

In the same way that a `JButton` fires an `ActionEvent` when pressed, a `JPanel` fires a `MouseEvent` when certain mouse actions are taken over it. In fact, there are so many possible mouse actions that there are two possible listeners—`MouseListener` and `MouseMotionListener`.

Here is a simple program that demonstrates the possibilities. In practice, we will use only a limited number of these methods.

```
public class MouseEventDisplay extends JPanel implements
MouseListener, MouseMotionListener {
    int x = 0, y = 0;
    String text = "";
    public MouseEventDisplay() {
        setBackground(Color.white);
        setFont(new Font("sanserif", Font.BOLD, 12));
        addMouseListener(this);
        addMouseMotionListener(this);
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawString(text, 10, 15);
    }
    public void mouseClicked(MouseEvent me) {
        setUp("clicked", me.getX(), me.getY());
    }
    public void mouseEntered(MouseEvent me) {
        setUp("entered", me.getX(), me.getY());
    }
    public void mouseExited(MouseEvent me) {
        setUp("exited", me.getX(), me.getY());
    }
    public void mousePressed(MouseEvent me) {
        setUp("pressed", me.getX(), me.getY());
    }
    public void mouseReleased(MouseEvent me) {
        setUp("released", me.getX(), me.getY());
    }
    public void mouseDragged(MouseEvent me) {
        setUp("dragged", me.getX(), me.getY());
    }
    public void mouseMoved(MouseEvent me) {
        setUp("moved", me.getX(), me.getY());
    }
    public void setUp(String s, int x, int y) {
        this.x = x; this.y = y; text = "Mouse " + s + " at " + x + ", " + y;
        repaint();
    }
    public static void main(String[] args) {
        DrawFrame frame = new DrawFrame("MouseEventDisplay");
        MouseEventDisplay display = new MouseEventDisplay();
        display.setPreferredSize(new Dimension(300, 300));
        frame.getContentPane().add(display, BorderLayout.CENTER);
        frame.pack();
        frame.show();
    }
}
```

In spite of the proliferation of methods, this is still a relatively simple program. We declare our `JPanel` to implement `java.awt.event.MouseListener` and `java.awt.event.MouseMotionListener`. We then declare, in the constructor, that this `JPanel` will listen to mouse events that occur over it. When a mouse event occurs, we will simply print out which occurs and where it happens.

The only two methods required for `MouseMotionListener` are `mouseDragged` and `mouseMoved`. The others are needed for `MouseListener`.

The large number of methods for `MouseListener` make it somewhat inconvenient to implement. Fortunately, Java provides us with a way out by defining a class called `MouseAdapter`, an implementation of `MouseListener` in which all the methods are empty. This allows us to extend `MouseAdapter` and override just the methods we are interested in. In the same vein, there is a class `MouseMotionAdapter` we may extend.

In our next example, we'll draw two points and a line between them. When the mouse is clicked in a point and then dragged, the point is moved and line modified accordingly. The code for the `JPanel` is given below. It relies on a class `MoveablePoint` that will be described later.

```
public class MovingPoints extends JPanel {
    MoveablePoint[] p;
    MoveablePoint moving;
    Line2D.Float line;
    public MovingPoints() {
        setBackground(Color.white);
        p = new MoveablePoint[2];
        p[0] = new MoveablePoint(100, 100);
        p[1] = new MoveablePoint(200, 200);
        line = new Line2D.Float(p[0], p[1]);
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                for (int i = 0; i < 2; i++) {
                    if (p[i].hit(me.getX(), me.getY())) {
                        moving = p[i]; movePoint(me.getX(), me.getY());
                        return;
                    }
                }
            }
            public void mouseReleased(MouseEvent me) {
                movePoint(me.getX(), me.getY());
                moving = null;
            }
        });
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent me) {
                movePoint(me.getX(), me.getY());
            }
        });
    }
    void movePoint(int x, int y) {
        if (moving == null) return;
        moving.setLocation(x, y);
        line.setLine(p[0].x, p[0].y, p[1].x, p[1].y);
        repaint();
    }
    public void paintComponent(Graphics gfx) {
        super.paintComponent(gfx);
        Graphics2D g = (Graphics2D) gfx;
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        g.setPaint(Color.blue);    g.draw(line);
        Shape s0 = p[0].getShape(); Shape s1 = p[1].getShape();
        g.setColor(Color.red);    g.fill(s0); g.fill(s1);
        g.setColor(Color.black);  g.draw(s0); g.draw(s1);
    }
}
```

You should notice here that the panel listens for three types of mouse events: pressing the mouse, dragging it (with the mouse pressed) and releasing it. In particular, when the mouse is pressed, we determine whether it is pressed over one of our points and if so, set the moving field equal to that point. When dragged, we move the point and update the line. Similarly, when the mouse is released, we move the point, update the line and set moving back to null to indicate that no point is currently selected.

Here is the class `MoveablePoint`

```
class MoveablePoint extends Point2D.Float {
    int r = 3;
    Shape shape;
    public MoveablePoint(int x, int y) {
        super(x, y);
        setLocation(x, y);
    }
    void setLocation(int x, int y) {
        super.setLocation(x, y);
        shape = new Ellipse2D.Float(x - r, y - r, 2*r, 2*r);
    }
    public boolean hit(int x, int y) {
        return shape.contains(x, y);
    }
    public Shape getShape() {
        return shape;
    }
}
```

The `hit` method simply returns whether the mouse is pressed inside the point while the `setLocation` method updates the coordinates of the point.

Exercise 4: On a panel whose dimensions are 300×300 , draw a circle of radius 125 pixels centered at the center of the panel and a point on the circle. Allow the viewer to move the point so that it is constrained to lie on the circle.

Exercise 5: Construct a diagram illustrating Proposition I, Book I of *The Elements* using two `MoveablePoints` to begin.

Exercise 6: Using three `MoveablePoints`, construct a figure which is always a right triangle.

Exercise 7: Using three `MoveablePoints` as the vertices of a triangle, construct the three medians and the centroid.

Exercise 8: Using three `MoveablePoints` as the vertices of a triangle, construct the circumscribing circle.

Exercise 9: Use three `MoveablePoints` as the control points of a quadratic curve. Add lines connecting the endpoints of the curve to the third control point.

Exercise 10: Construct the convex hull of ten `MoveablePoints`.

5: Updating the figure package

The ability to construct this kind of dynamic illustration is so useful we would like to incorporate it into our figure package. The idea is the same: we need to detect when the viewer requests that an item be moved and then respond in an appropriate way.

We will first define an interface called `Moveable` to be implemented by figure elements that viewers may move. To begin, we will have our `GraphicalPoint` class implement this interface. However, as we add on to the figure package, it is possible that other elements will implement this interface as well. The object that describes what should happen when a `Moveable` is moved will implement an interface called `Mover`.

Here is the `Moveable` interface:

```
package figure;
public interface Moveable {
    public boolean hit(int x, int y);
    public void setMover(Mover m);
    public Mover getMover();
}
```

First, we have a method `hit` that determines whether the viewer is requesting that this object be moved. Also, we include methods to set and retrieve the object responsible for responding to the request.

The `Mover` interface looks like this:

```
package figure;
public interface Mover {
    public void move(Moveable m, double x, double y);
}
```

The only method needed is the request to move a `Moveable` object to the point (x, y) .

Let's now look at what we should add so that `GraphicalPoint` implements the `Moveable` interface. First, we will include an instance field that can hold the `Mover` object responsible for moving the point. We will also include an instance field that holds the current `Shape` of the point as drawn on the screen.

```
Shape shape;
Mover mover;
```

Here are the methods required for `GraphicalPoint` to implement `Moveable`.

```
public boolean hit(int xp, int yp) {
    return shape.contains(xp, yp);
}
public void setMover(Mover m) {
    mover = m;
}
public Mover getMover() {
    return mover;
}
```

Up to this point, we have not changed the coordinates of a `GraphicalPoint` after instantiating it. Clearly, we need this ability now so we add

```
public void setPoint(double xp, double yp) {
    x = xp; y = yp;
}
```

Finally, we need to modify the `plot` method to store the `Shape` drawn on the screen.

```
public void plot(Graphics2D g) {
    Point2D.Double point = new Point2D.Double(x, y);
    toPixels(point);
    if (style == CIRCLE) shape =
        new Ellipse2D.Double(point.x - size, point.y - size,
                             2*size, 2*size);
    else shape =
        new Rectangle2D.Double(point.x - size, point.y - size,
                              2*size, 2*size);
    g.setPaint(color);
    g.fill(shape);
    g.setPaint(Color.black);
    g.draw(shape);
}
```

This has been fairly painless so far. Now, we need to add in some lower-level code to listen for the appropriate `MouseEvents` and to send the appropriate ones to the appropriate `Mover`. This will go in the `FigurePanel` class.

First, we will add instance fields as follows.

```
Vector moveables;
Moveable moving = null;
```

The `Vector moveables` will contain all the `Moveable` objects added to the `FigurePanel`. The `Moveable moving` will hold the `Moveable` currently being moved.

Next we add in the code to detect and respond to `MouseEvents`.

```

addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent me) {
        for (int i = 0; i < moveables.size(); i++) {
            Moveable m = (Moveable) moveables.elementAt(i);
            if (m.hit(me.getX(), me.getY())) {
                moving = m;
                moveTo(me.getX(), me.getY());
                repaint();
                break;
            }
        }
    }
    public void mouseReleased(MouseEvent me) {
        if (moving == null) return;
        moveTo(me.getX(), me.getY());
        moving = null;
        repaint();
    }
});
addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseDragged(MouseEvent me) {
        if (moving == null) return;
        moveTo(me.getX(), me.getY());
        repaint();
    }
});

```

Since we need to perform the same operation under the three different `MouseEvent`s, we add a method

```

public void moveTo(int x, int y) {
    Point2D.Double point = new Point2D.Double(x, y);
    try {
        transform.inverseTransform(point, point);
    } catch (NoninvertibleTransformException ex) {
        return;
    }
    moving.getMover().move(moving, point.x, point.y);
}

```

Notice that the `MouseEvent` gives us coordinates in pixels so we need to invert the coordinate transform and convert pixel coordinates back in the mathematical coordinates of our figure.

Lastly, we need to include methods to register and unregister `Moveable` objects. For instance, it is easy to imagine a figure containing several `GraphicalPoints` of which we wish to allow only a few to move.

```

public void addMoveable(Moveable m, Mover mover) {
    moveables.addElement(m);
    m.setMover(mover);
}
public void removeMoveable(Moveable m) {
    moveables.removeElement(m);
}

```

Let's see how to use these new tools with a simple example. This diagram will consist of a grid, a set of axes, two points and a line connecting them. The points may be moved with the line always drawn between the two points.

```

public class LineMover extends FigurePanel implements Mover {
    GraphicalPoint p0, p1;
    GraphicalLine line;
    public LineMover() {
        super(-3, -3, 3, 3);
        p0 = new GraphicalPoint(0, 0);
        p0.setColor(Color.red);
        p0.setSize(3);
        p1 = new GraphicalPoint(1, 1);
        p1.setColor(Color.red);
        p1.setSize(3);
        line = new GraphicalLine(p0.x, p0.y, p1.x, p1.y);
        line.setColor(Color.blue);
        Grid grid = new Grid();
        grid.setColor(Color.lightGray);
        add(grid);
        add(new Axes());
        add(line);
        add(p0); add(p1);
        addMoveable(p0, this); addMoveable(p1, this);
    }
    public void move(Moveable m, double x, double y) {
        ((GraphicalPoint) m).setPoint(x, y);
        line.setPoints(p0.x, p0.y, p1.x, p1.y);
    }
}

```

Notice that we make the FigurePanel implement Mover to consolidate our code. The only thing that is really new here are the instructions

```
addMoveable(p0, this); addMoveable(p1, this);
```

and the method

```

public void move(Moveable m, double x, double y) {
    ((GraphicalPoint) m).setPoint(x, y);
    line.setPoints(p0.x, p0.y, p1.x, p1.y);
}

```

Exercise 11: Draw the graph of the function $f(x) = x^3 - x$ in a viewing window $-2 \leq x, y \leq 2$ along with a grid and a set of axes. Now add in a point that can slide along the graph and display the tangent line at the point. You will need to think of a strategy for dealing with movements that take the point outside the viewing window.

Exercise 12: Create a figure in a viewing window $-2 \leq x, y \leq 2$ containing a moveable point and its three cube roots.

Exercise 13: Create a figure with two moveable points, \vec{v} and \vec{w} , and a set of axes. Include the lattice generated by \vec{v} and \vec{w} . You may either plot the points in the lattice or draw a skewed grid generated by the points. You may find it helpful to define a new `Plotable` object or add on to `Grid`.

Exercise 14: Illustrate Euler's formula

$$e^{ix} = \cos x + i \sin x$$

as follows. Create a figure consisting of a set of axes, a grid, the unit circle and a point that slides along the x -axis. Represent the terms in the Taylor series of e^{ix} as straight lines where the beginning of the line representing one term is at the end of the line representing the previous term. For instance,

$$e^i = 1 + i - \frac{1}{2} + \dots$$

so include a line from $(0, 0)$ to $(1, 0)$, another from $(1, 0)$ to $(1, 1)$, a third from $(1, 1)$ to $(1, 1/2)$ and so forth.

Exercise 15: A typical type of illustration depends on one parameter that we would like to allow the viewer to adjust. A convenient way to do this is through a "slider," a point that moves along an axis. Design and construct a `Slider` class to add to our figure package. There are several ways to do this so think it through carefully before beginning.

Exercise 16: Using your `Slider` class, draw the graph of $f(x) = x^3 - cx$ where c is an adjustable parameter satisfying $-1 \leq c \leq 1$.

6: Animations

So far in this chapter, we have essentially been creating figures that display a sequence of images. This is the basic idea behind creating animations. In fact, the figures we constructed with buttons were essentially simple animations in which frames are displayed one after the other in response to a button being pressed. To create animations, we need to take the button out of the figure and include a mechanism for triggering the display of the next frame.

The complication in doing this is that we usually want to give the viewer the opportunity to pause or reset the animation in the middle of it, perhaps by pressing a button. This means that our program needs to do at least two things: redisplay frames of the animation and monitor the buttons to determine if they have been pressed. In earlier versions of Java, the only way to do this was through the use of `Threads`, a feature built into the earliest releases of Java.

However, `Swing` provides us with another, simpler means of doing this using a class called `javax.swing.Timer`. (Please note that there is a similar class `java.util.Timer`. I would prefer for us to use the `Timer` in `javax.swing` since it is easier to use and designed specifically with graphical applications in mind.) The reason for this is that calls to `repaint` in `Swing` should be made from

the event-handling thread. If we create our own `Thread` to handle the display of frames of our animation, we need a means of getting the `repaint` call into event-handling thread. The `Timer` class effectively takes care of this for us. This class is really quite simple: after being started, the `Timer` fires an `ActionEvent` every time a time interval, whose length may be specified, has passed.

To use a `Timer` for animations, we put our code to display the next frame into an `ActionListener`, instantiate a `Timer`, register the `ActionListener` with the `Timer` and start the `Timer`. Here is an example that animates a point moving across the screen.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
public class SimpleAnimation extends JPanel {
    Timer timer;
    double x = 0, y = 150, dx = 2;
    double r = 3;
    int interval = 10;
    public SimpleAnimation() {
        setBackground(Color.white);
        timer = new Timer(interval, new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                x += dx;
                repaint();
            }
        });
        timer.setInitialDelay(0);
    }
    public void start() {
        if (!timer.isRunning()) timer.start();
    }
    public void stop() {
        if (timer.isRunning()) timer.stop();
    }
    public void reset() { stop(); x = 0; repaint(); }
    public void paintComponent(Graphics gfx) {
        super.paintComponent(gfx);
        Graphics2D g = (Graphics2D) gfx;
        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        Ellipse2D.Double point = new Ellipse2D.Double(x - r, y - r, 2*r, 2*r);
        g.setPaint(Color.blue);    g.fill(point);
        g.setColor(Color.black);   g.draw(point);
    }
}
```

```

static JButton control;
static SimpleAnimation animation;
public static void main(String[] args) {
    DrawFrame frame = new DrawFrame("SimpleAnimation");
    animation = new SimpleAnimation();
    animation.setPreferredSize(new Dimension(300, 300));
    frame.getContentPane().add(animation, BorderLayout.CENTER);
    control = new JButton("Start");
    control.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            String cmd = event.getActionCommand();
            if (cmd.equals("Start")) {
                animation.start();
                control.setText("Stop");
            } else if (cmd.equals("Stop")) {
                animation.stop();
                control.setText("Start");
            }
        }
    });
    JButton reset = new JButton("Reset");
    reset.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            animation.reset();
            control.setText("Start");
        }
    });
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(control);
    buttonPanel.add(reset);
    frame.getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    frame.pack();
    frame.show();
}
}

```

The code is a little longer than usual here but the idea is quite simple. First, we keep track of the coordinates of the point by declaring instance fields `x` and `y`. In the constructor, we set up the `Timer`.

```

timer = new Timer(interval, new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        x += dx;
        repaint();
    }
});
timer.setInitialDelay(0);

```

The two arguments in the `Timer`'s constructor are the interval, expressed in milliseconds, between the firing of `ActionEvents` and then an `ActionListener` to receive the events. In our case, the `ActionListener` simply increments `x` and repaints.

Next, we add three methods, `start`, `stop` and `reset` to control the `Timer` through a collection of `JButtons` added.

Making animations appear smooth almost always require some experimentation. You may wish to investigate what happens if you change the interval at which the `Timer` fires and the amount `dx` that `x` is incremented between frames.

Exercise 17: Create an animation of a ball in free fall according to $y(t) = 1/2gt^2$ where $g = 9.8$ meters per second squared.

Exercise 18: Add another `FigurePanel` next to the figure of the falling ball that contains the graph of $y(t) = 1/2gt^2$ and contains a `GraphicalPoint` that moves along the graph with the falling ball.

Exercise 19: Create an animation of a circle rolling on top of the x -axis together with a point on the circle.