9.10 Processing Timer Events

In this section we will study timer events and show how they allow you to implement simple animations.

The `Timer` class in the `javax.swing` package generates a sequence of action events, spaced apart at even time intervals. (You can think of a timer as an invisible button that is automatically clicked.) This is useful whenever you want to have an object updated in regular intervals. For example, in an animation, you may want to update a scene ten times per second and redisplay the image, to give the illusion of movement.

A timer generates timer events at fixed intervals.

When you use a timer, you specify the frequency of the events and an object of a class that implements the `ActionListener` interface. Place whatever action you want to occur inside the `actionPerformed` method. Finally, start the timer.

```
class MyListener implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Action that is executed at each timer event
   }
}

MyListener listener = new MyListener();
Timer t = new Timer(interval, listener);
t.start();
```

Then the timer calls the `actionPerformed` method of the `listener` object every `interval` milliseconds.

Our sample program will display a moving rectangle. We first supply a `RectangleComponent` class with a `moveBy` method that moves the rectangle by a given amount.

ch09/timer/RectangleComponent.java

```java
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

/**
   This component displays a rectangle that can be moved.
*/
public class RectangleComponent extends JComponent
{
   private static final int BOX_X = 100;
   private static final int BOX_Y = 100;
   private static final int BOX_WIDTH = 20;
   private static final int BOX_HEIGHT = 30;

   private Rectangle box;

   public RectangleComponent()
   {
      // The rectangle that the paint method draws
      box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
   }

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

      g2.draw(box);
   }

   /**
      Moves the rectangle by a given amount.
      @param x the amount to move in the x-direction
      @param y the amount to move in the y-direction
   */
   public void moveBy(int dx, int dy)
   {
      box.translate(dx, dy);
      repaint();
   }
}
```

The `repaint` method causes a component to repaint itself. Call `repaint` whenever you modify the shapes that the `paintComponent` method draws.

Note the call to `repaint` in the `moveBy` method. This call is necessary to ensure that the component is repainted after the state of the rectangle object has been changed. Keep in mind that the component object does not contain the pixels that show the drawing. The component merely contains a `Rectangle` object, which itself contains four coordinate values. Calling `translate` updates the rectangle coordinate values. The call to `repaint` forces a call to the `paintComponent` method. The `paintComponent` method redraws the component, causing the rectangle to appear at the updated location.

The `actionPerformed` method of the timer listener simply calls `component.moveBy(1, 1)`. This moves the rectangle one pixel down and to the right. Since the `actionPerformed` method is called many times per second, the rectangle appears to move smoothly across the frame.

ch09/timer/RectangleMover.java

```java
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JFrame;
import javax.swing.Timer;

/**
   This program moves the rectangle.
*/
public class RectangleMover
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;

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

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setTitle("An animated rectangle");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      final RectangleComponent component = new RectangleComponent();
      frame.add(component);

      frame.setVisible(true);

      class TimerListener implements ActionListener
      {
         public void actionPerformed(ActionEvent event)
         {
            component.moveBy(1, 1);
         }
      }

      ActionListener listener = new TimerListener();

      final int DELAY = 100; // Milliseconds between timer ticks
      Timer t = new Timer(DELAY, listener);
      t.start();
   }
}
```

SELF CHECK

21. Why does a timer require a listener object?
22. What would happen if you omitted the call to `repaint` in the `moveBy` method?

Common Error 9.6

Forgetting to Repaint

You have to be careful when your event handlers change the data in a painted component. When you make a change to the data, the component is not automatically painted with the new data. You must call the `repaint` method of the component, either in the event handler or in the component's mutator methods. Your component's `paintComponent` method will then be invoked with an appropriate `Graphics` object. Note that you should not call the `paintComponent` method directly.

This is a concern only for your own painted components. When you make a change to a standard Swing component such as a `JLabel`, the component is automatically repainted.

9.11 Mouse Events

You use a mouse listener to capture mouse events.

If you write programs that show drawings, and you want users to manipulate the drawings with a mouse, then you need to process mouse events. Mouse events are more complex than button clicks or timer ticks.

A mouse listener must implement the `MouseListener` interface, which contains the following five methods:

```
public interface MouseListener
{
   void mousePressed(MouseEvent event);
       // Called when a mouse button has been pressed on a component
   void mouseReleased(MouseEvent event);
       // Called when a mouse button has been released on a component
   void mouseClicked(MouseEvent event);
       // Called when the mouse has been clicked on a component
   void mouseEntered(MouseEvent event);
       // Called when the mouse enters a component
   void mouseExited(MouseEvent event);
       // Called when the mouse exits a component
}
```

The `mousePressed` and `mouseReleased` methods are called whenever a mouse button is pressed or released. If a button is pressed and released in quick succession, and the mouse has not moved, then the `mouseClicked` method is called as well. The `mouseEntered` and `mouseExited` methods can be used to paint a user-interface component in a special way whenever the mouse is pointing inside it.

The most commonly used method is `mousePressed`. Users generally expect that their actions are processed as soon as the mouse button is pressed.

You add a mouse listener to a component by calling the `addMouseListener` method:

```
public class MyMouseListener implements MouseListener
{
   // Implements five methods
}

MouseListener listener = new MyMouseListener();
component.addMouseListener(listener);
```

In our sample program, a user clicks on a component containing a rectangle. Whenever the mouse button is pressed, the rectangle is moved to the mouse location. We first enhance the `RectangleComponent` class and add a `moveTo` method to move the rectangle to a new position.

ch09/mouse/RectangleComponent.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Rectangle;
import javax.swing.JComponent;

```

```
/**
   This component displays a rectangle that can be moved.
*/
public class RectangleComponent extends JComponent
{
   private static final int BOX_X = 100;
   private static final int BOX_Y = 100;
   private static final int BOX_WIDTH = 20;
   private static final int BOX_HEIGHT = 30;

   private Rectangle box;

   public RectangleComponent()
   {
      // The rectangle that the paint method draws
      box = new Rectangle(BOX_X, BOX_Y, BOX_WIDTH, BOX_HEIGHT);
   }

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

      g2.draw(box);
   }

   /**
      Moves the rectangle to the given location.
      @param x the x-position of the new location
      @param y the y-position of the new location
   */
   public void moveTo(int x, int y)
   {
      box.setLocation(x, y);
      repaint();
   }
}
```

Note the call to `repaint` in the `moveTo` method. As explained in the preceding section, this call causes the component to repaint itself and show the rectangle in the new position.

Now, add a mouse listener to the component. Whenever the mouse is pressed, the listener moves the rectangle to the mouse location.

```
class MousePressListener implements MouseListener
{
   public void mousePressed(MouseEvent event)
   {
      int x = event.getX();
      int y = event.getY();
      component.moveTo(x, y);
   }

   // Do-nothing methods
   public void mouseReleased(MouseEvent event) {}
   public void mouseClicked(MouseEvent event) {}
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
}
```

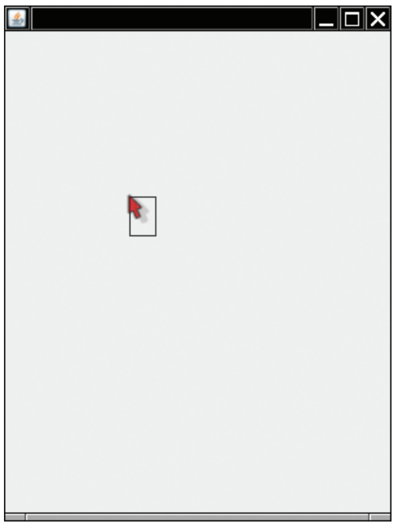
Figure 8 Clicking the Mouse Moves the Rectangle

It often happens that a particular listener specifies actions only for one or two of the listener methods. Nevertheless, all five methods of the interface must be implemented. The unused methods are simply implemented as do-nothing methods.

Go ahead and run the `RectangleComponentViewer` program. Whenever you click the mouse inside the frame, the top-left corner of the rectangle moves to the mouse pointer (see Figure 8).

ch09/mouse/RectangleComponentViewer.java

```java
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;

/**
   This program displays a RectangleComponent.
*/
public class RectangleComponentViewer
{
   private static final int FRAME_WIDTH = 300;
   private static final int FRAME_HEIGHT = 400;

   public static void main(String[] args)
   {
      final RectangleComponent component = new RectangleComponent();

      // Add mouse press listener

      class MousePressListener implements MouseListener
      {
         public void mousePressed(MouseEvent event)
         {
            int x = event.getX();
            int y = event.getY();
            component.moveTo(x, y);
         }

         // Do-nothing methods
         public void mouseReleased(MouseEvent event) {}
         public void mouseClicked(MouseEvent event) {}
         public void mouseEntered(MouseEvent event) {}
         public void mouseExited(MouseEvent event) {}
      }

      MouseListener listener = new MousePressListener();
      component.addMouseListener(listener);

      JFrame frame = new JFrame();
      frame.add(component);

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setVisible(true);
   }
}
```

SELF CHECK

23. Why was the `moveBy` method in the `RectangleComponent` replaced with a `moveTo` method?

24. Why must the `MousePressListener` class supply five methods?

Special Topic 9.3

Event Adapters

In the preceding section you saw how to install a mouse listener into a mouse event source and how the listener methods are called when an event occurs. Usually, a program is not interested in all listener notifications. For example, a program may only be interested in mouse clicks and may not care that these mouse clicks are composed of “mouse pressed” and “mouse released” events. Of course, the program could supply a listener that implements all those methods in which it has no interest as “do-nothing” methods, for example:

```
class MouseClickListener implements MouseListener
{
   public void mouseClicked(MouseEvent event)
   {
      Mouse click action
   }

   // Four do-nothing methods
   public void mouseEntered(MouseEvent event) {}
   public void mouseExited(MouseEvent event) {}
   public void mousePressed(MouseEvent event) {}
   public void mouseReleased(MouseEvent event) {}
}
```

This is boring. For that reason, some friendly soul has created a `MouseAdapter` class that implements the `MouseListener` interface such that all methods do nothing. You can *extend* that class, inheriting the do-nothing methods and overriding the methods that you care about, like this:

```
class MouseClickListener extends MouseAdapter
{
   public void mouseClicked(MouseEvent event)
   {
```

```
            Mouse click action
        }
    }
```

See Chapter 10 for more information on the process of extending classes.

Random Fact 9.2

Programming Languages

Many hundreds of programming languages exist today, which is actually quite surprising. The idea behind a high-level programming language is to provide a medium for programming that is independent from the instruction set of a particular processor, so that one can move programs from one computer to another without rewriting them. Moving a program from one programming language to another is a difficult process, however, and it is rarely done. Thus, it seems that there would be little use for so many programming languages.

Unlike human languages, programming languages are created with specific purposes. Some programming languages make it particularly easy to express tasks from a particular problem domain. Some languages specialize in database processing; others in "artificial intelligence" programs that try to infer new facts from a given base of knowledge; others in multimedia programming. The Pascal language was purposefully kept simple because it was designed as a teaching language. The C language was developed to be translated efficiently into fast machine code, with a minimum of housekeeping overhead. The C++ language builds on C by adding features for object-oriented programming. The Java language was designed for securely deploying programs across the Internet.

The initial version of the C language was designed around 1972. As different compiler writers added incompatible features, the language sprouted various dialects. Some programming instructions were understood by one compiler but rejected by another. Such divergence is an immense pain to a programmer who wants to move code from one computer to another, and an effort got underway to iron out the differences and come up with a standard version of C. The design process ended in 1989 with the completion of the ANSI (American National Standards Institute) Standard. In the meantime, Bjarne Stroustrup of AT&T added features of the language Simula (an object-oriented language designed for carrying out simulations) to C. The resulting language was called C++. From 1985 on, C++ grew by the addition of many features, and a standardization process was completed in 1998. C++ has been enormously popular because programmers can take their existing C code and move it to C++ with only minimal changes. In order to keep compatibility with existing code, every innovation in C++ had to work around the existing language constructs, yielding a language that is powerful but somewhat cumbersome to use.

In 1995, Java was designed by James Gosling to be conceptually simpler and more internally consistent than C++, while retaining the syntax that is familiar to millions of C and C++

James Gosling, Designer of the Java Programming Language programmers. The Java language was an immediate success, not only because it eliminated many of the cumbersome aspects of C++, but also because it included a powerful library.

However, programming language evolution has not come to an end. There are aspects of programming that Java does not handle well. Computers with multiple processors are becoming increasingly common, and it is difficult to write *concurrent* Java programs that use multiple processors correctly and efficiently. Also, as you have seen in this chapter, it can be rather cumbersome in Java to deal with small blocks of code such as the code for measuring an object. In Java, you have to make an interface with a method for that code, and then provide a class that implements the interface. In *functional* programming languages, you can manipulate functions in the same way as objects, and such tasks becomes much easier. For example, in Scala, a hybrid functional/object-oriented language, you can simply construct a `DataSet` with a function, without having to use an interface:

```
data = new DataSet((x : Rectangle) => x.getWidth() * x.getHeight())
```

Could the Java language be enhanced for concurrent and functional programming? Some attempts have been made, but they were not encouraging because they interacted with existing language features in complex ways. Some new languages (such as Scala) run on the same virtual machine as Java and can easily call existing Java code. Nobody can tell which languages will be most successful in the years to come, but most software developers should expect to work with multiple programming languages during their career.