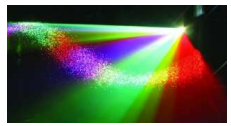
Chapter 2
Event-Driven Programming

The Design and Implementation of Multimedia Software

David Bernstein

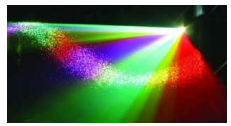
Jones and Bartlett Publishers

www.jbpub.com



About this Chapter

- Good designs are adequate, rugged, easy to repair/enhance, easy to understand/document, and have components that are easy to re-use.
- This book is predicated on the belief that object-oriented techniques help lead to designs with these properties.
- However, for multimedia software, object-oriented techniques on their own are often not enough.
- Hence, this chapter considers how an event-driven design can help ensure that multimedia software has these desirable properties.



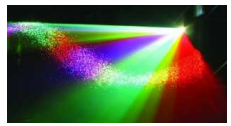
Motivation

- Most introductory programming and design courses focus on software products that follow a 'step-by-step' process.

- Most multimedia software products cannot be described/conceptualized in this way.

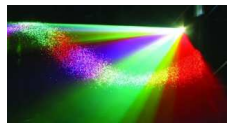
  The software needs to respond to various user actions (e.g., mouse clicks, key presses) that might occur in any order and at any time.

  The software needs to do multiple things 'at the same time' (e.g., present visual and auditory content, present multiple tracks of auditory content).



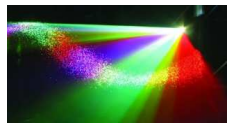
What's Next?

We need to consider event-driven design and programming.



Focus of Event-Driven Designs

- The events that can occur (e.g., mouse clicks, timing signals, key presses);
- The classes that can generate events of different kinds (often called *event generators*); and
- The classes that need to respond to events of different kinds (often called *event receivers*).



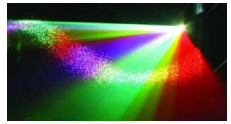
The Event Queue

- The event queue is responsible for ensuring that everything happens in the right order.

- The event queue is a central repository for events.

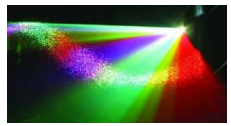
  Event generators add events to the back of the queue (a process known as *posting*).

  Event receivers are sent events as they are removed from the front of the queue (a process that is known as *firing* or *dispatching*).



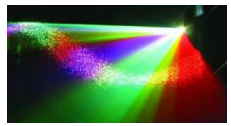
What's Next?

We need to consider the event queue and dispatch thread in Java.



The Java `EventQueue`

- Events are managed by a single `EventQueue` object.
- The `EventQueue` object fires events using the `dispatchEvent()` method.

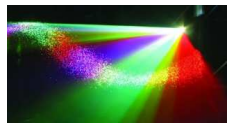


The Event Dispatch Thread

- Events are dispatched in a single *event dispatch thread*.
- There are two closely related ways to execute statements in the event dispatch thread (both involve static methods in the `SwingUtilities` class that are passed a `Runnable` object).

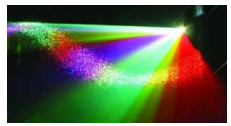
  The `invokeAndWait()` method blocks until all pending events have been processed.

  The `invokeLater()` method adds the call to the `Runnable` object's `run()` method to the end of the event queue and returns immediately.



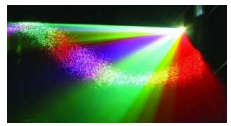
What's Next? GUIs

- They are a nice way to experiment with event-based programming
- We will need to use them to present visual content



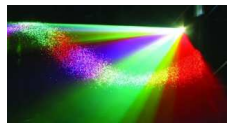
JLabel

- Displays a `String`, an `Icon` or both.
- The alignment of a `JLabel` object's content can be controlled with the `setHorizontalAlignment()` and `setVerticalAlignment()` methods, both of which must be passed an `int` value (defined in `SwingConstants`).



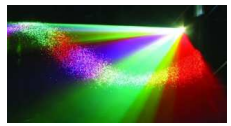
JLabel (cont.)

```
label = new JLabel(s, SwingConstants.CENTER);
```



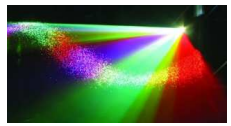
JButton

- A `JButton` is a GUI component that behaves like a key on the keyboard.
- Like a `JLabel`, it can contain a `String`, an `Icon`, or both.



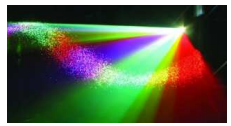
JButton (cont.)

```
button = new JButton(CHANGE);
```



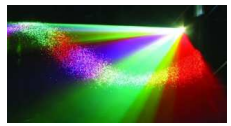
Other Components

- JCheckBox
- JList
- JSlider
- JSpinner
- JTextArea
- JTextField



Types of Containers

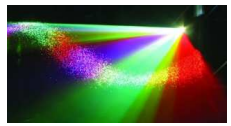
- *Top-Level Containers*
- Ordinary Containers



Top-Level Containers

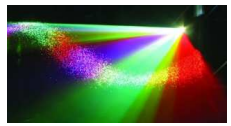
Constructing a JFrame

```
window = new JFrame();
window.setSize(600,400);
```



Top-Level Containers (cont.)

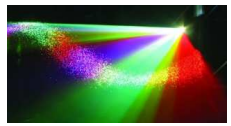
- Though top-level containers are containers they should not be used directly in that way.
- Instead, one should get the *root pane* (which is, itself, a container) from the top-level container and use it.
- To that end, all top-level containers implement the `RootPaneContainer` interface which includes a `getRootPane()` method that returns a `JRootPane`.
- In fact, one should use the *content pane* (that is inside of the root pane) which can be obtained using the `getContentPane()` method of the `JRootPane` (or the `getContentPane()` method of the `RootPaneContainer`)



Ordinary Containers

Getting the Content Pane (which is a `JPanel`)

```
contentPane = (JPanel)window.getContentPane();
```



Layout Basics

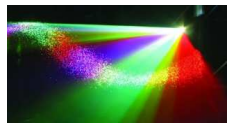
- Definition:

  The process of positioning and sizing the components in a container.

- Approaches:

  Use a `LayoutManager`

  Use *absolute layout* (sometimes called *null layout*).



Absolute Layout

Getting Started

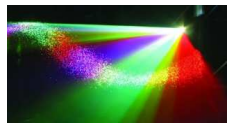
```
contentPane.setLayout(null);
```

Setting-up the JLabel

```
label.setBounds(50,50,500,100);
contentPane.add(label);
```

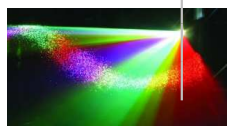
Setting-up the JButton

```
button.setBounds(450,300,100,50);
contentPane.add(button);
```



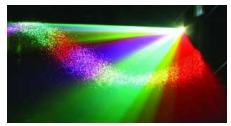
BadRandomMessageExample

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

public class BadRandomMessageApplication
{
    // The pseudo-random number generator
    private static Random         rng = new Random();

    // The messages
    private static final String[] MESSAGES = {
       "What a great book.","Bring on the exercises.",
       "Author, author!","I hope it never ends."};

    public static void main(String[] args) throws Exception
    {
       JFrame                    window;
       JLabel                    label;
       JPanel                    contentPane;
       String                    s;

       // Select a message at random
       s = createRandomMessage();

       // Construct the "window"
       window = new JFrame();
       window.setSize(600,400);
       window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



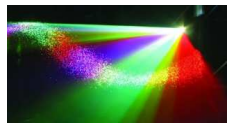
BadRandomMessageExample (cont.)

```
        // Get the container for all content
        contentPane = (JPanel)window.getContentPane();
        contentPane.setLayout(null);

        // Add a component to the container
        label = new JLabel(s, SwingConstants.CENTER);
        label.setBounds(50,50,500,100);
        contentPane.add(label);

        // Make the "window" visible
        window.setVisible(true);
    }

    private static String createRandomMessage()
    {
        return  MESSAGES[rng.nextInt(MESSAGES.length)];
    }
}
```



A Problem

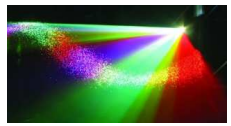
- The `main()` method manipulates elements of the GUI outside of the event dispatch thread (i.e., in the main thread).
- One can correct this problem using the `invokeAndWait()` method in the `SwingUtilities` class.



Fixing the Problem

Indicate that the Class is `Runnable`

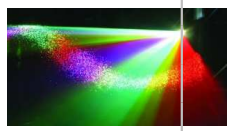
```
public class      BadRandomMessageSwingApplication
       implements Runnable
```



Fixing the Problem (cont.)

Move Code into the `run()` Method

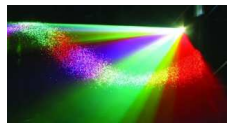
```
public void run()
{
   JFrame                    window;
   JPanel                    contentPane;
   String                    s;

   // Select a message at random
   s = createRandomMessage();

   // Construct the "window"
   window = new JFrame();
   window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   window.setSize(600,400);

   // Get the container for all content
   contentPane = (JPanel)window.getContentPane();
   contentPane.setLayout(null);

   // Add a component to the container
   label = new JLabel(s, SwingConstants.CENTER);
   label.setBounds(50,50,500,100);
   contentPane.add(label);

   // Make the "window" visible
   window.setVisible(true);
}
```



Fixing the Problem (cont.)

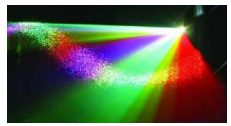
Fix the `main()` Method

```
public static void main(String[] args) throws Exception
{
   SwingUtilities.invokeAndWait(
                  new BadRandomMessageSwingApplication());
}
```



Low-Level and High-Level Events

- Using a Button (Low-Level):
  1. A `mouseEntered()` message is generated.
  2. A `mousePressed()` message is generated.
  3. A `mouseReleased()` message is generated.
  4. A `mouseClicked()` message may be generated.
- Using a Button (High-Level):
  1. The button generates a high level event.
  2. The event queue fires to the listeners/observers.



The `JButton` Class

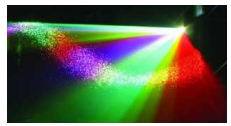
- The Event:

  `ActionEvent`

- The Observers/Listeners:

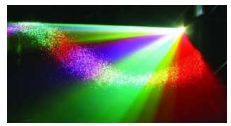
  Implement the `ActionListener` interface.

  Subscribe using the `addActionListener()` method.



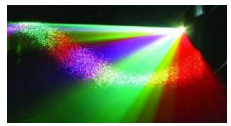
Other Events in Java

- `MouseEvent`
- `KeyEvent`
- `ItemEvent`
- `TextEvent`
- `WindowEvent`



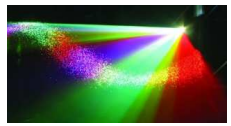
Handling Messages

```
public void actionPerformed(ActionEvent event)
{
   String      actionCommand;

   actionCommand = event.getActionCommand();
   if (actionCommand.equals(CHANGE))
   {
      label.setText(createRandomMessage());
   }
}
```



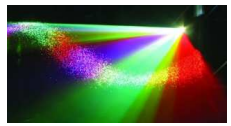
Setting-Up the JButton

```
button = new JButton(CHANGE);
button.setBounds(450,300,100,50);
contentPane.add(button);
button.addActionListener(this);
```



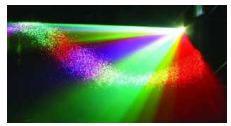
What's Next?

We need to consider timed events.

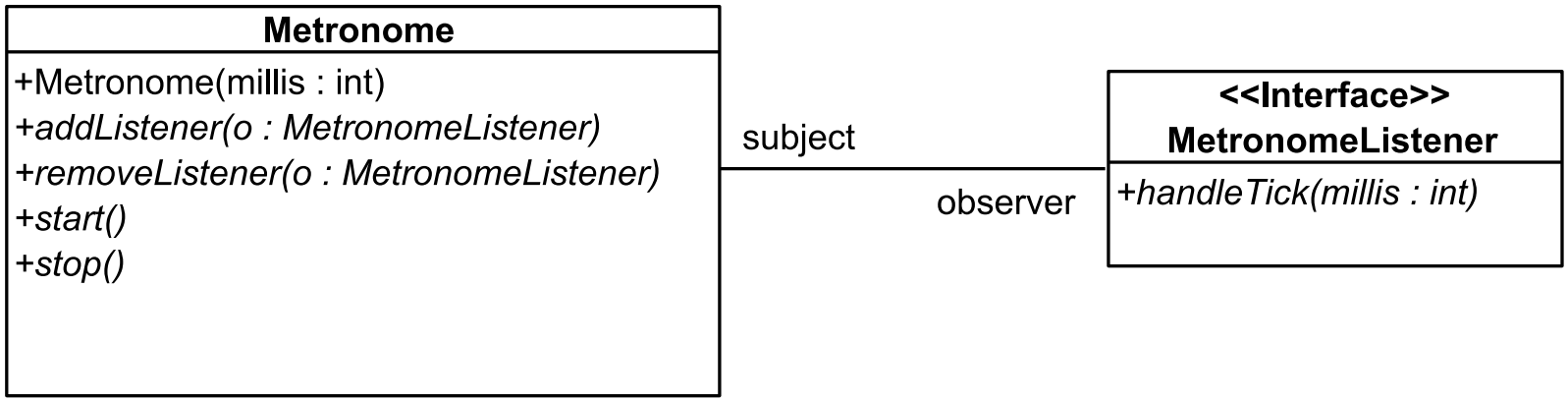


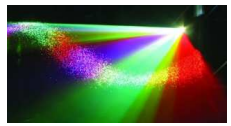
Types of 'Clock-Based' Events

- Events that occur at a particular point in time;
- Events that occur after a particular interval of time;
- Events that recur after a particular interval of time (called *fixed-delay* execution); and
- Events that recur at a particular rate (called *fixed-rate* execution).



Design of a Metronome



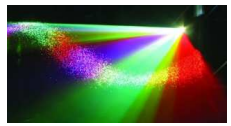


Alternative Ways to Manage Listeners

- Use a thread-safe collection.

  What are the shortcomings?

- Make a copy of the collection of listeners and use the copy for notification.

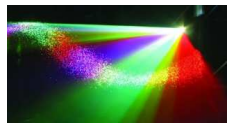


Alternative Ways to Manage Listeners

- Use a thread-safe collection.

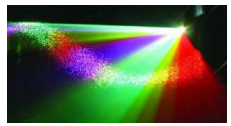
  The shortcoming is that notification process could be delayed by modifications to the collection of listeners.

- Make a copy of the collection of listeners and use the copy for notification.



Managing Listeners (cont.)

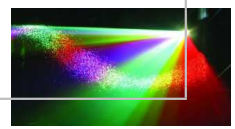
```
public synchronized void addListener(MetronomeListener ml)
{
   listeners.add(ml);
   copyListeners();
}

private void copyListeners()
{
   copy = new MetronomeListener[listeners.size()];
   listeners.toArray(copy);
}

public synchronized void removeListener(MetronomeListener ml)
{
   listeners.remove(ml);
   copyListeners();
}
```



Calling `handleTick()` in the Event Dispatch Thread

The `Runnable` to Pass to `invokeLater()`

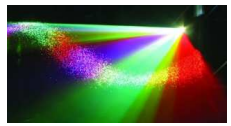
```
private class MetronomeTickDispatcher implements Runnable
{
    private MetronomeListener[]   listeners;
    private int                   time;


    public void run()
    {
       int    n;

       n = listeners.length;
       for (int i=n-1; i>=0; i--)
       {
          if (listeners[i] != null)
             listeners[i].handleTick(time);
       }
    }

    public void setup(MetronomeListener[] listeners,
                      int time)
    {
       this.listeners = listeners;
       this.time      = time;
    }
}
```



Calling `handleTick()` (cont.)

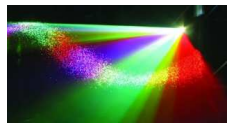
The `notifyListeners()` Method

```
protected synchronized void notifyListeners()
{
   // Setup the state of the MetronomeTickDispatcher
   dispatcher.setup(copy, time);

   // Cause the run() method of the dispatcher to be
   // called in the GUI/event-dispatch thread
   EventQueue.invokeLater(dispatcher);
}
```



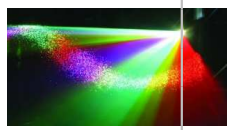
The start() Method

```
public void start()
{
   if (timerThread == null)
   {
      keepRunning = true;
      timerThread = new Thread(this);
      timerThread.start();
   }
}
```



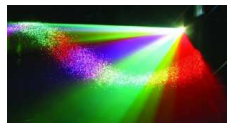
The run() Method

```
public void run()
{
   int       currentDelay;
   long      currentTick, drift;

   currentDelay = delay;
   if (adjusting) lastTick = System.currentTimeMillis();

   while (keepRunning)
   {
      try
      {
         timerThread.sleep(currentDelay);
         time += currentDelay * multiplier;

         if (adjusting) // Need to compensate for drift
         {
            currentTick = System.currentTimeMillis();
            drift = (currentTick - lastTick) - currentDelay;
            currentDelay = (int)Math.max(0, delay-drift);
            lastTick = currentTick;
         }
         notifyListeners();
      }
      catch (InterruptedException ie)
      {
         // stop() was called
      }
   }
   timerThread = null;
}
```



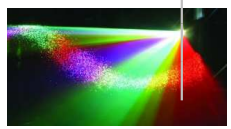
The Structure

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

import event.*;

public class      StopWatchSwingApplication
       implements ActionListener, MetronomeListener, Runnable
{
    private JLabel                    label;
    private Metronome                 metronome;

    private static final String       START = "Start";
    private static final String       STOP  = "Stop";

    public static void main(String[] args) throws Exception
    {
       SwingUtilities.invokeAndWait(
                      new StopWatchSwingApplication());
    }
}
```



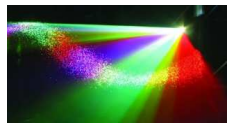
The run() Method

```
public void run()
{
   JButton                  start, stop;
   JFrame                   window;
   JPanel                   contentPane;

   window = new JFrame();
   window.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
   window.setSize(600,400);

   contentPane = (JPanel)window.getContentPane();
   contentPane.setLayout(null);

   label       = new JLabel("0");
   label.setBounds(250,100,100,100);
   contentPane.add(label);

   start = new JButton(START);
   start.setBounds(50,300,100,50);
   start.addActionListener(this);
   contentPane.add(start);


   stop  = new JButton(STOP);
   stop.setBounds(450,300,100,50);
   stop.addActionListener(this);
   contentPane.add(stop);
```



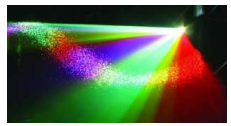
The run() Method (cont.)

```
        metronome = new Metronome(1000, true);
        metronome.addListener(this);

        window.setVisible(true);
    }
```



The `handleTick()` Method

```
public void handleTick(int millis)
{
   label.setText(""+millis/1000);
}
```



The actionPerformed() Method

```
public void actionPerformed(ActionEvent event)
{
   String   actionCommand;

   actionCommand = event.getActionCommand();
   if      (actionCommand.equals(START))
   {
      label.setText("0");
      metronome.reset();
      metronome.start();
   }
   else if (actionCommand.equals(STOP))
   {
      metronome.stop();
   }
}
```