13.3 Buttons and ActionListeners

Basic steps for using a button in a Swing program:

- Create a Button object
- Add the Button object to a container
- Create an `ActionListener` object that has an `actionPerformed` method
- Register the listener for the Button object

The following slides show an example of each step.

Use the Method `getActionCommand`

```
   public ButtonDemo( )
   {
//.........
      JButton stopButton = new JButton("Red");
      stopButton.addActionListener(this);
      contentPane.add(stopButton);

      JButton goButton = new JButton("Green");
      goButton.addActionListener(this);
      contentPane.add(goButton);
.......
   }
```

```
  public void actionPerformed(ActionEvent e)
   {
     Container contentPane = getContentPane( );

     if (e.getActionCommand( ).equals("Red"))
        contentPane.setBackground(Color.RED);
     else if (e.getActionCommand( ).equals("Green"))
        contentPane.setBackground(Color.GREEN);
     else
        System.out.println("Error in button interface.");
   }
}
```

Java Tip: Use the Method **`setActionCommand`**

- `e.getActionCommand` returns action command
  - » by default, action command is string written on button
  - » can specify a different string for action command by using `setActionCommand` method

- Example:

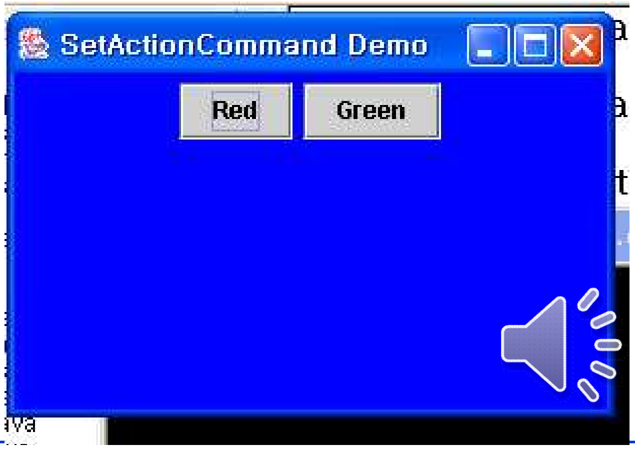
```
JButton stopButton = new JButton("Red");
stopButton.setActionCommand("Stop");
e.getActionCommand will return "Stop"
```

- Allows you to have two different buttons with the same string displayed.
- Also allows you to change what buttons say without changing the action command, and vice versa.

LISTING 13.8 A GUI with Buttons

```
import javax.swing.JButton;
import javax.swing.JFrame;
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 Simple demonstration of putting buttons in a JFrame.
*/
public class ButtonDemo extends JFrame implements ActionListener
{
    public static final int WIDTH = 300;
    public static final int HEIGHT = 200;
public ButtonDemo( )
 {
    setSize(WIDTH, HEIGHT);

    addWindowListener(new WindowDestroyer( ));
    setTitle("Button Demo");
    Container contentPane = getContentPane( );
    contentPane.setBackground(Color.BLUE);

    contentPane.setLayout(new FlowLayout( ));

    JButton stopButton = new JButton("Red");
    stopButton.addActionListener(this);
    contentPane.add(stopButton);

    JButton goButton = new JButton("Green");
    goButton.addActionListener(this);
    contentPane.add(goButton);
 }
```

```
public void actionPerformed(ActionEvent e)
   {
      Container contentPane = getContentPane( );

      if (e.getActionCommand( ).equals("Red"))
         contentPane.setBackground(Color.RED);
      else if (e.getActionCommand( ).equals("Green"))
         contentPane.setBackground(Color.GREEN);
      else
         System.out.println("Error in button interface.");
   }

   /**
    Creates and displays a window of the class ButtonDemo.
   */
   public static void main(String[] args)
   {
       ButtonDemo buttonGui = new ButtonDemo( );
       buttonGui.setVisible(true);
   }
}
```



The Model-View-Controller Pattern

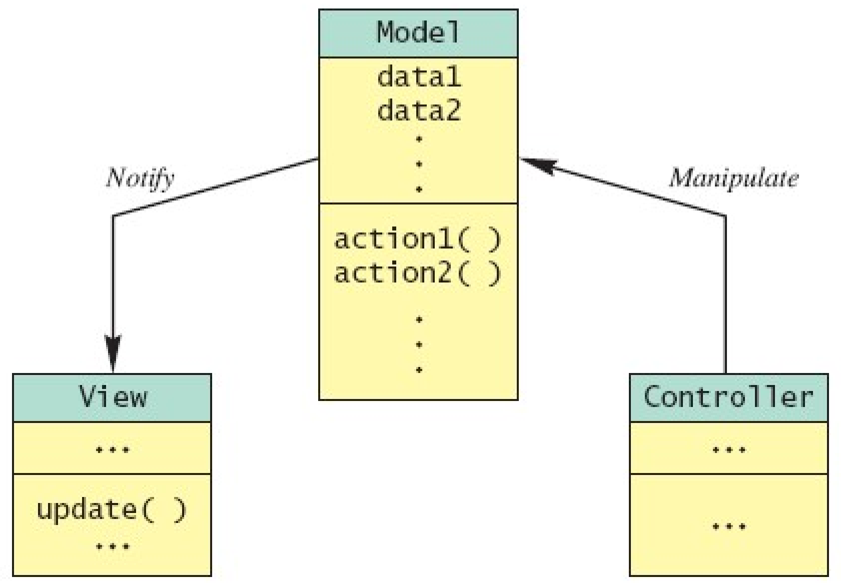
Coding look and action separately is an example of using the general ***Model-View-Controller pattern***.

- ***Model***: performs the heart of the application
- ***View***: output part of the application; displays Model's state
- ***Controller***: input part; relays user commands to Model

- The Model-View-Controller pattern is a good way to break up a difficult problem into more manageable pieces.
- It also helps make an application more modular.
- In a Swing GUI, the View and Controller might be separate classes combined into one larger class.

Model–View–Controller Pattern

- Figure 13.9 the model–view–controller pattern



- 컨트롤러
  - » 모델에 명령을 보냄으로써 모델의 상태를 변경할 수 있다.
- 모델
  - » 모델의 상태에 변화가 있을 때 컨트롤러와 뷰에 이를 통보한다. 이와 같은 통보를 통해서 뷰는 최신의 결과를 보여줄 수 있고, 컨트롤러는 모델의 변화에 따른 적용 가능한 명령을 추가·제거·수정할 수 있다.
- 뷰
  - » 사용자가 볼 결과물을 생성하기 위해 모델로부터 정보를 얻어온다.

끝