Events Handling

Introduction

- The Swing components related events are handled using the Events Delegation model.

  The same apply for AWT components.

- The Events Delegation model is a variation of the Observer design pattern.

The Observer Design Pattern

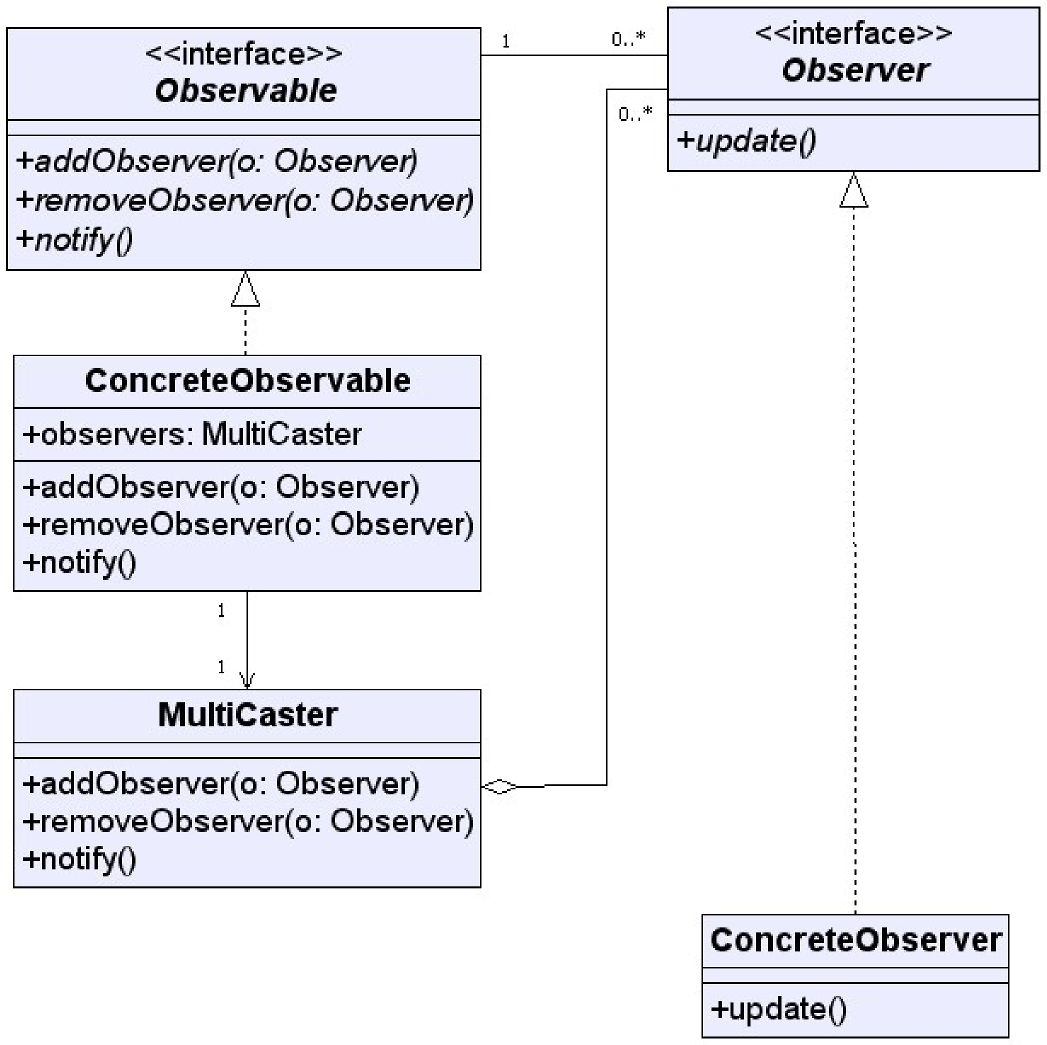
- The Observer design pattern answers the following question:

  How to allow one (or more objects) dependent on another object to be dynamically notified when the state of the other object is changed... ?

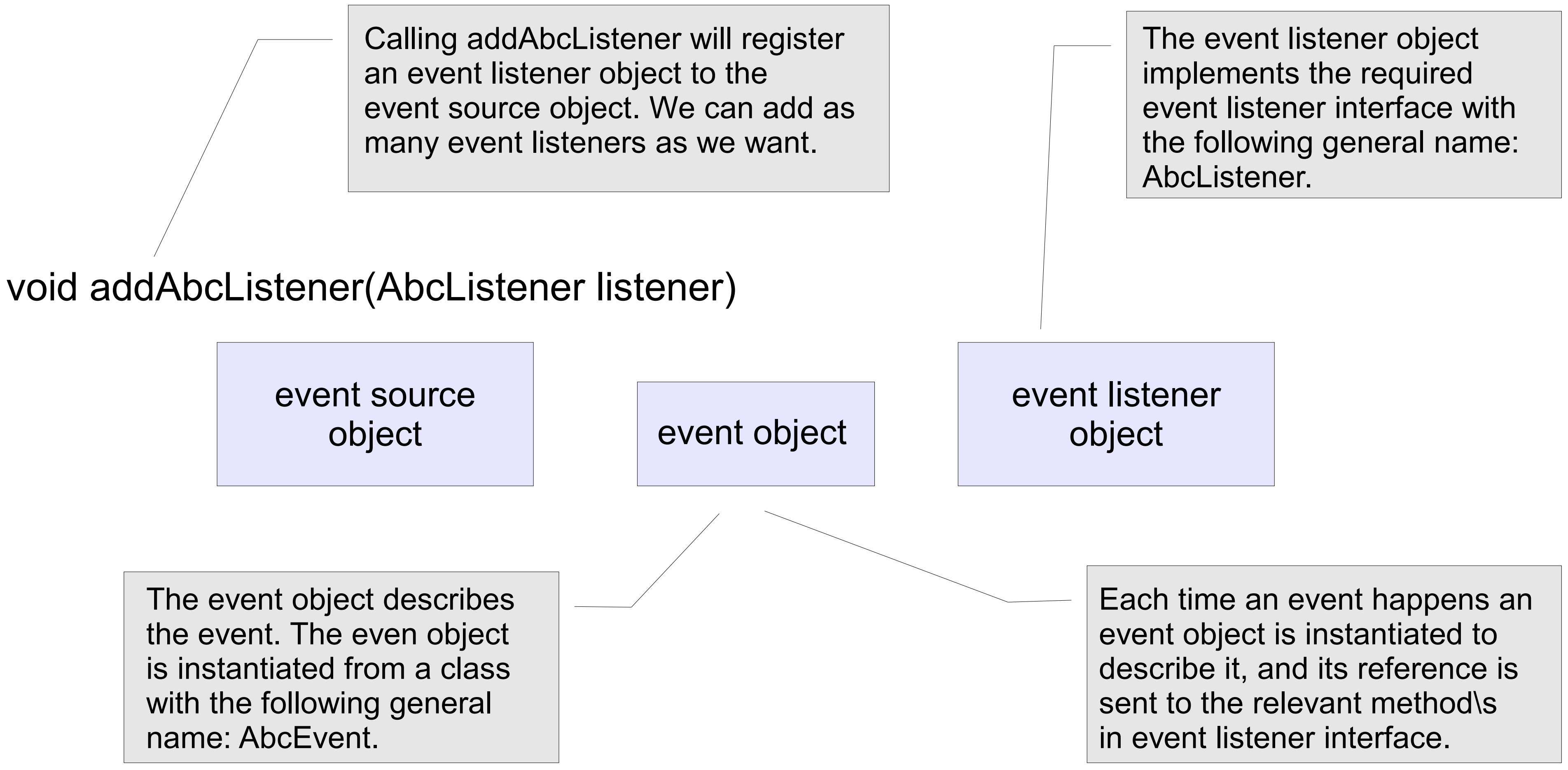
- The Observer design pattern solution is:

  Declare the Observer and the Observable interfaces. The first will be implemented by the classes from which we will instantiate the objects that should be notified of a change. The second will be implemented by the object that when its state changes the notification should be sent. Observer objects shall be registered as observers by calling the addObserver method on the Observable object.

The Observer Design Pattern



The Events Delegation Model



Calling addAbcListener will register an event listener object to the event source object. We can add as many event listeners as we want.

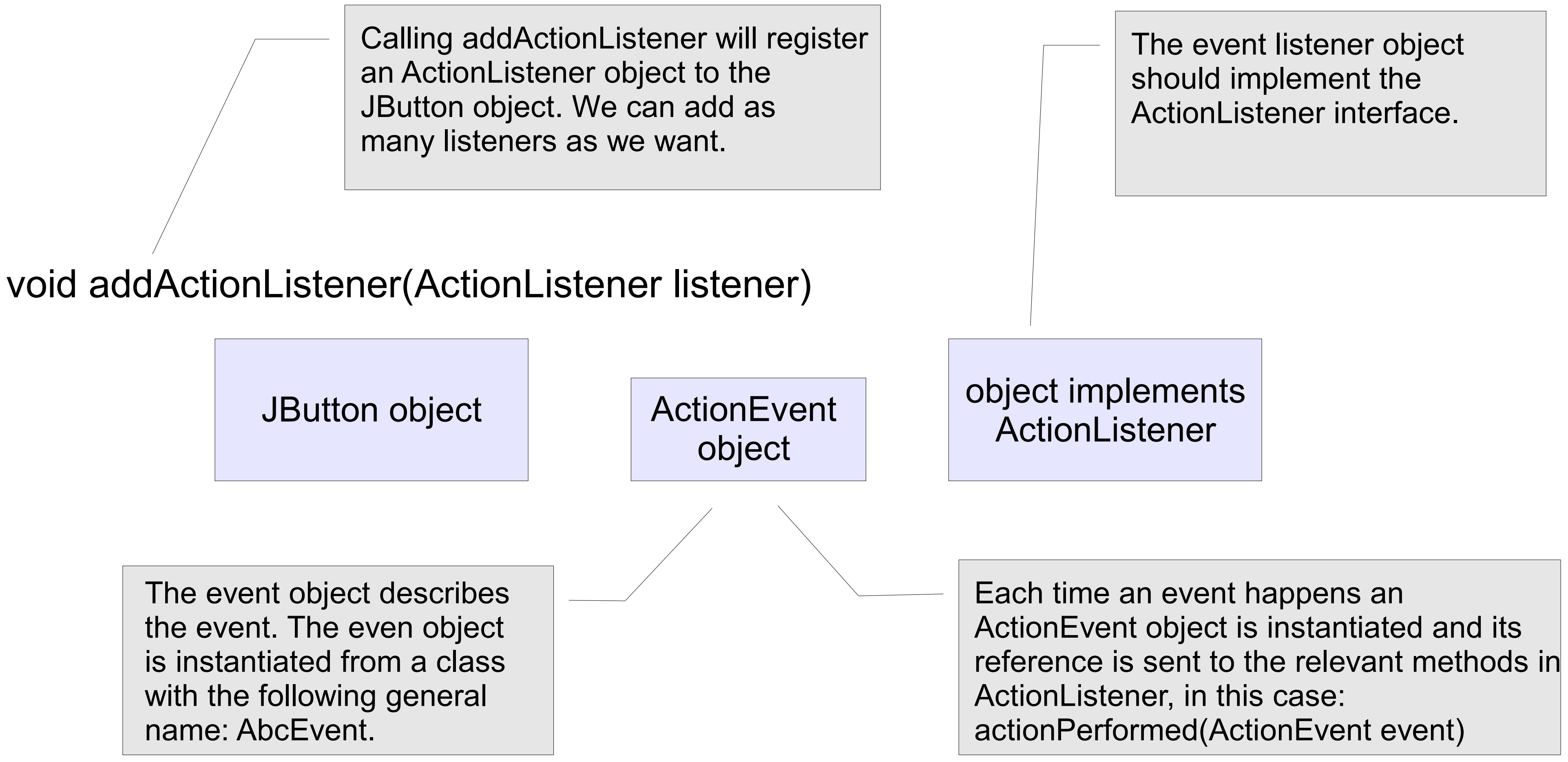
The event listener object implements the required event listener interface with the following general name: AbcListener.

void addAbcListener(AbcListener listener)

The event object describes the event. The even object is instantiated from a class with the following general name: AbcEvent.

Each time an event happens an event object is instantiated to describe it, and its reference is sent to the relevant method\s in event listener interface.

The Events Delegation Model



Calling addActionListener will register an ActionListener object to the JButton object. We can add as many listeners as we want.

The event listener object should implement the ActionListener interface.

void addActionListener(ActionListener listener)

The event object describes the event. The even object is instantiated from a class with the following general name: AbcEvent.

Each time an event happens an ActionEvent object is instantiated and its reference is sent to the relevant methods in ActionListener, in this case: actionPerformed(ActionEvent event)

The Events Delegation Model

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

public class SimpleActionEventDemo
{
    private JFrame frame;
    private JButton btOne,btTwo;
    private JTextField textField;
    private int counter;
    private ActionListener buttonListener;
```

The Events Delegation Model

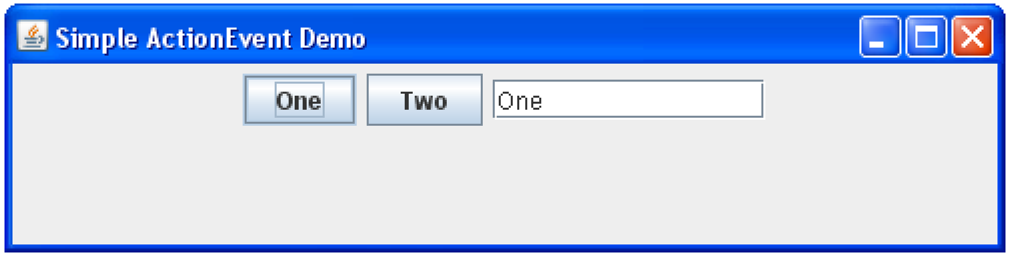
```
public SimpleActionEventDemo()
{
    frame = new JFrame("Simple ActionEvent Demo");
    btOne = new JButton("One");
    btTwo = new JButton("Two");
    textField = new JTextField(12);
    buttonListener = new MyButtonListener();
    btOne.addActionListener(buttonListener);
    btTwo.addActionListener(buttonListener);
    frame.setLayout(new FlowLayout());
    frame.add(btOne);
    frame.add(btTwo);
    frame.add(textField);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

The Events Delegation Model

```
public void go()
{
    frame.setSize(500,400);
    frame.setVisible(true);
}
class MyButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = null;
        Object source = e.getSource();
        if(source==btOne)
        {
            text = "One";
        }
        else if(source==btTwo)
        {
            text = "Two";
        }
        textField.setText(text);
    }
}
```

The Events Delegation Model

```
    public static void main(String args[])
    {
        SimpleActionEventDemo demo = new SimpleActionEventDemo();
        demo.go();
    }
}
```



The Action Command

- The `ActionEvent` object describes the event that took place when the user pressed the button.
- Identifying on which button the user pressed can be done either by calling `getEventSource()` or by calling the `getActionCommand()` on the ActionEvent object.
- The getActionCommand() method returns the string associated as the event source command.

  The ActionCommand associated with a button is its label (by default).

The Action Command

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

public class AnotherActionEventDemo
{
    private JFrame frame;
    private JButton btOne,btTwo;
    private JTextField textField;
    private int counter;
    private ActionListener buttonListener;

    public AnotherActionEventDemo()
    {
        frame = new JFrame("Another ActionEvent Demo");
        btOne = new JButton("One");
        btTwo = new JButton("Two");
        textField = new JTextField(12);
```

The Action Command

```
    buttonListener = new MyButtonListener();
    btOne.addActionListener(buttonListener);
    btTwo.addActionListener(buttonListener);
    frame.setLayout(new FlowLayout());
    frame.add(btOne);
    frame.add(btTwo);
    frame.add(textField);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

public void go()
{
    frame.setSize(500,400);
    frame.setVisible(true);
}
```

The Action Command

```
    class MyButtonListener implements ActionListener
    {
        public void actionPerformed(ActionEvent e)
        {
            String text = null;
            String command = e.getActionCommand();
            if(command.equals("One"))
            {
                text = "One";
            }
            else if(command.equals("Two"))
            {
                text = "Two";
            }
            textField.setText(text);
        }
    }
    public static void main(String args[])
    {
        AnotherActionEventDemo demo = new AnotherActionEventDemo();
        demo.go();
    }
}
```

Multi Threaded Events Handling

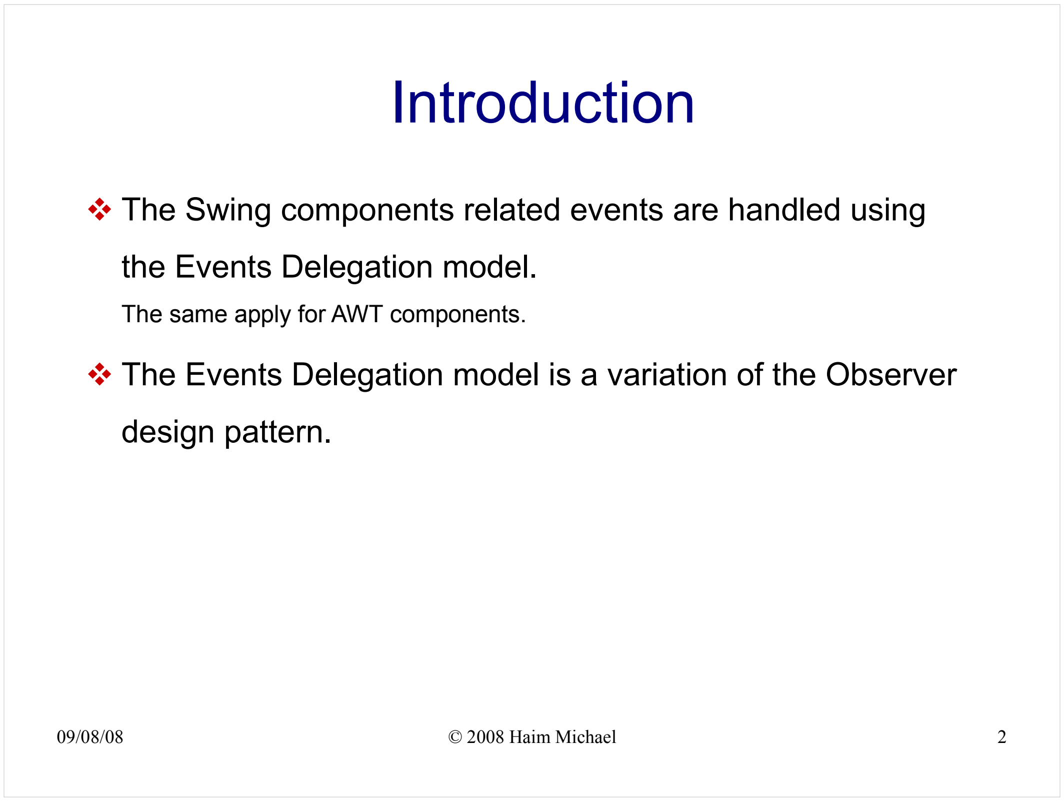
- All Swing components are not thread safe.

  This was done in order to increase their efficiency and decrease the code complexity.

- Given this design we must ensure that every access to a Swing component must be done from a single one thread... from the event dispatcher thread.

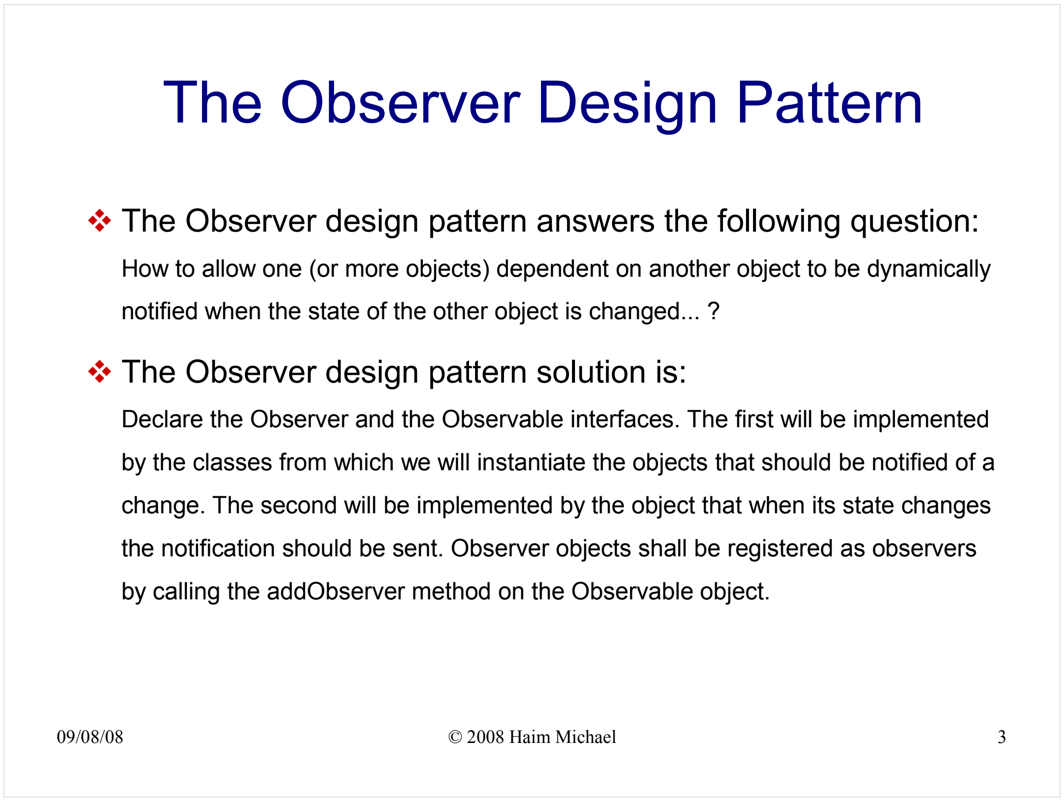
  Calling the EventQueue.isDispatchThread() or the SwingUtilities.isEventDispatchThread() methods can assist us to verify the current thread.

Events Handling



Introduction

- The Swing components related events are handled using the Events Delegation model.
  The same apply for AWT components.
- The Events Delegation model is a variation of the Observer design pattern.



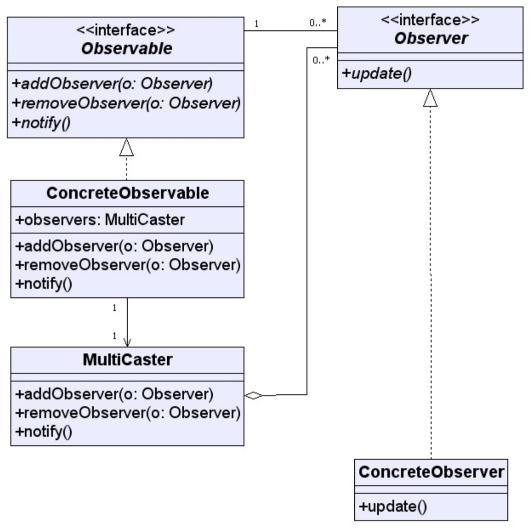
The Observer Design Pattern

- The Observer design pattern answers the following question:
  How to allow one (or more objects) dependent on another object to be dynamically notified when the state of the other object is changed... ?
- The Observer design pattern solution is:
  Declare the Observer and the Observable interfaces. The first will be implemented by the classes from which we will instantiate the objects that should be notified of a change. The second will be implemented by the object that when its state changes the notification should be sent. Observer objects shall be registered as observers by calling the addObserver method on the Observable object.

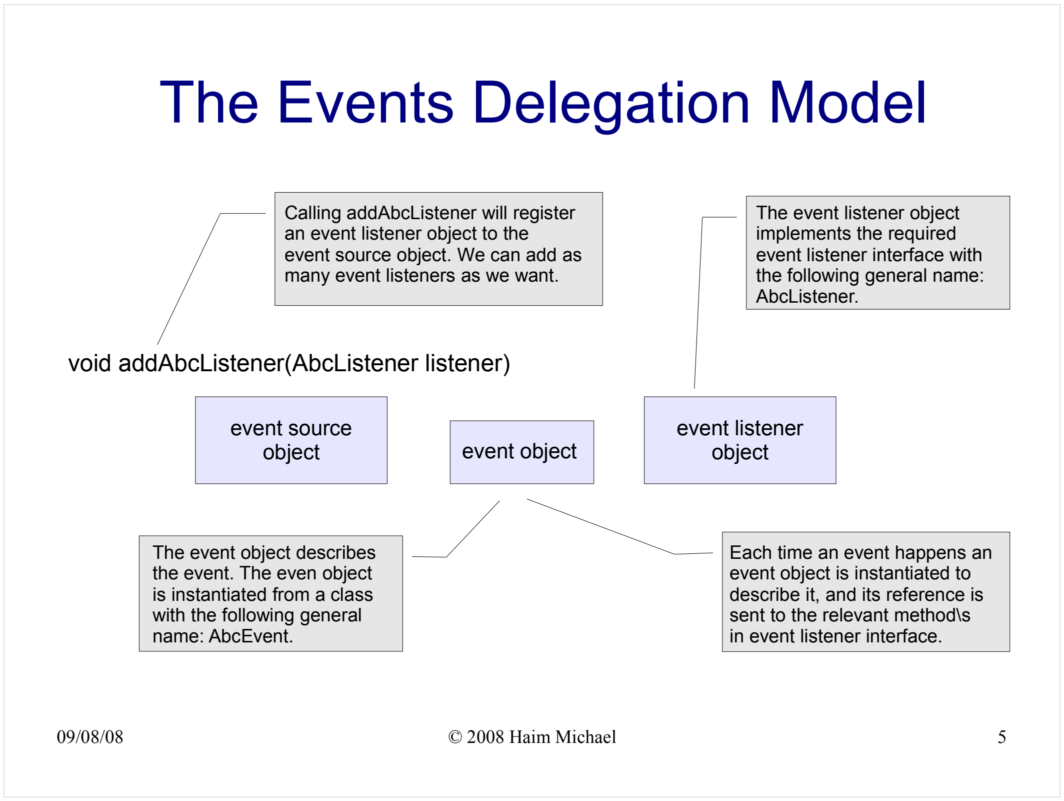
09/08/08 © 2008 Haim Michael 3

In the Events Delegation model the listener objects wait for events to happen (instead for a state to change).

The Observer Design Pattern



09/08/08 © 2008 Haim Michael 4



The Events Delegation Model

Calling addAbcListener will register an event listener object to the event source object. We can add as many event listeners as we want.

The event listener object implements the required event listener interface with the following general name: AbcListener.

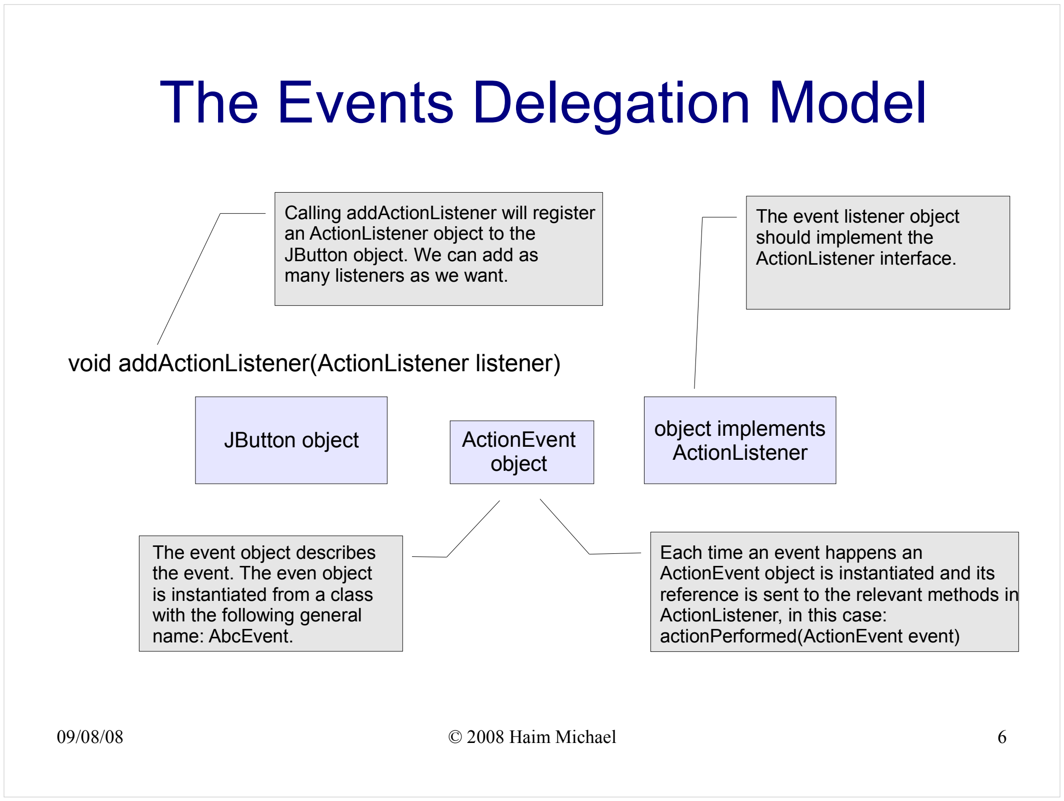
void addAbcListener(AbcListener listener)

event source object
event object
event listener object

The event object describes the event. The even object is instantiated from a class with the following general name: AbcEvent.

Each time an event happens an event object is instantiated to describe it, and its reference is sent to the relevant method\s in event listener interface.

09/08/08 © 2008 Haim Michael 5



The Events Delegation Model

Calling addActionListener will register an ActionListener object to the JButton object. We can add as many listeners as we want.

The event listener object should implement the ActionListener interface.

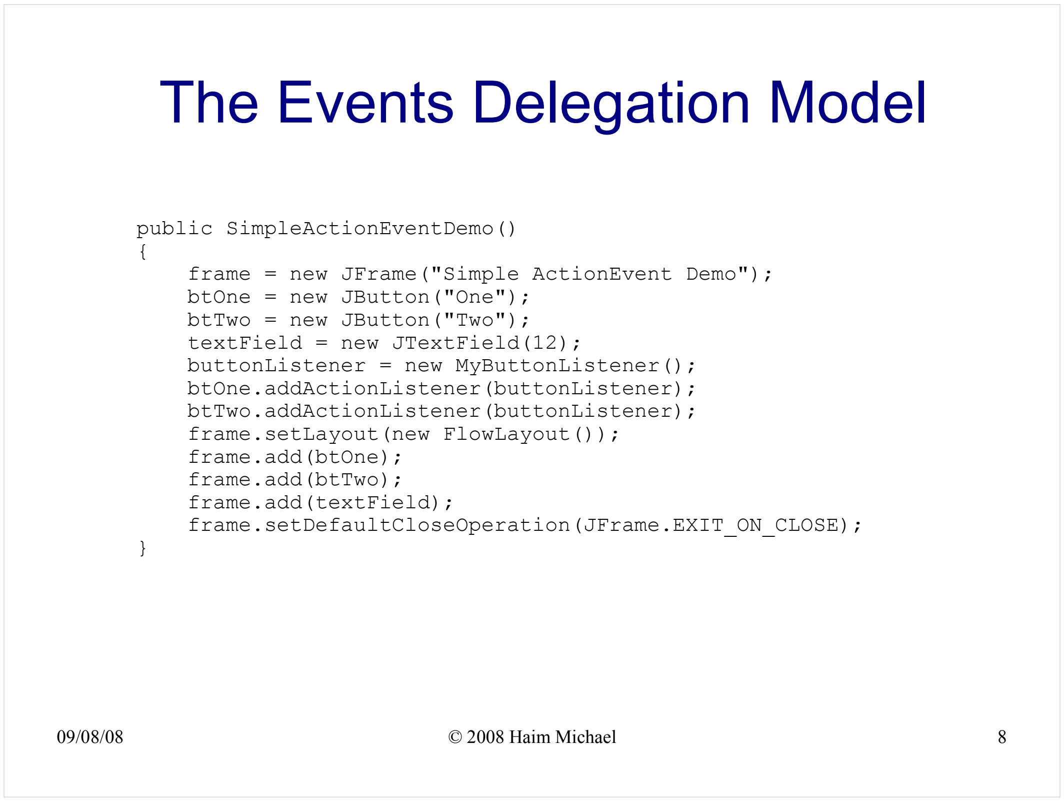
void addActionListener(ActionListener listener)

The event object describes the event. The even object is instantiated from a class with the following general name: AbcEvent.

Each time an event happens an ActionEvent object is instantiated and its reference is sent to the relevant methods in ActionListener, in this case: actionPerformed(ActionEvent event)

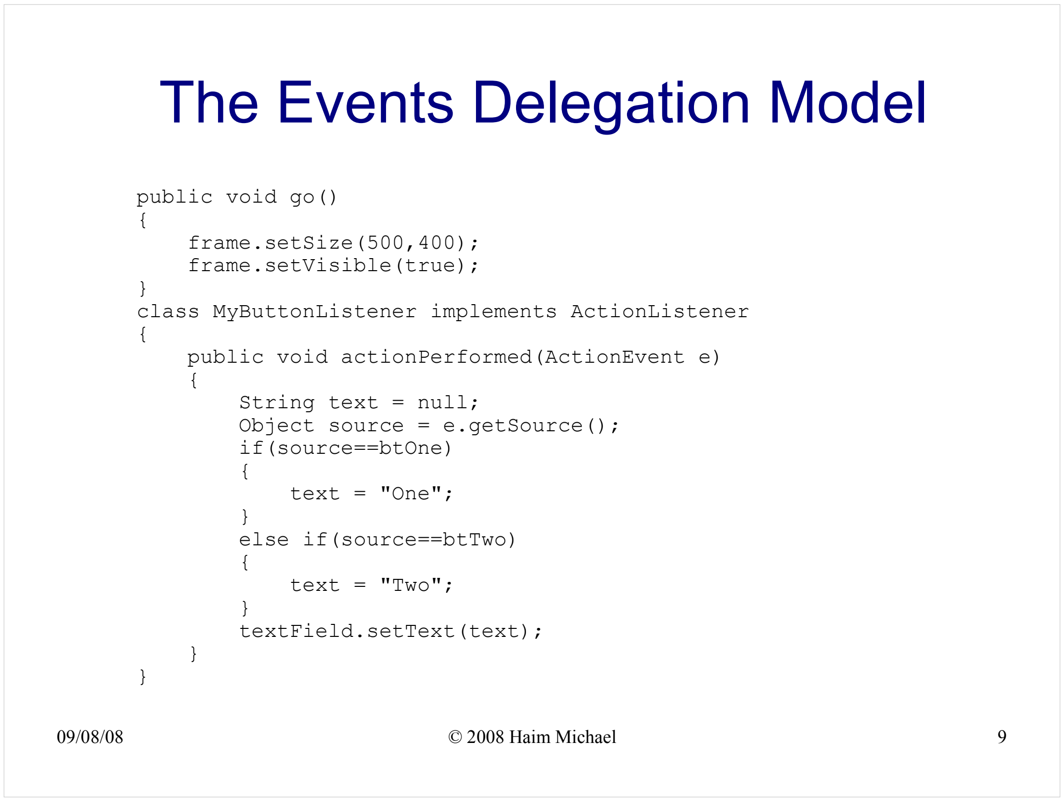
The Events Delegation Model

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

public class SimpleActionEventDemo
{
    private JFrame frame;
    private JButton btOne,btTwo;
    private JTextField textField;
    private int counter;
    private ActionListener buttonListener;
```



The Events Delegation Model

```
public SimpleActionEventDemo()
{
    frame = new JFrame("Simple ActionEvent Demo");
    btOne = new JButton("One");
    btTwo = new JButton("Two");
    textField = new JTextField(12);
    buttonListener = new MyButtonListener();
    btOne.addActionListener(buttonListener);
    btTwo.addActionListener(buttonListener);
    frame.setLayout(new FlowLayout());
    frame.add(btOne);
    frame.add(btTwo);
    frame.add(textField);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```



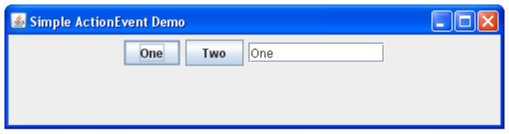
The Events Delegation Model

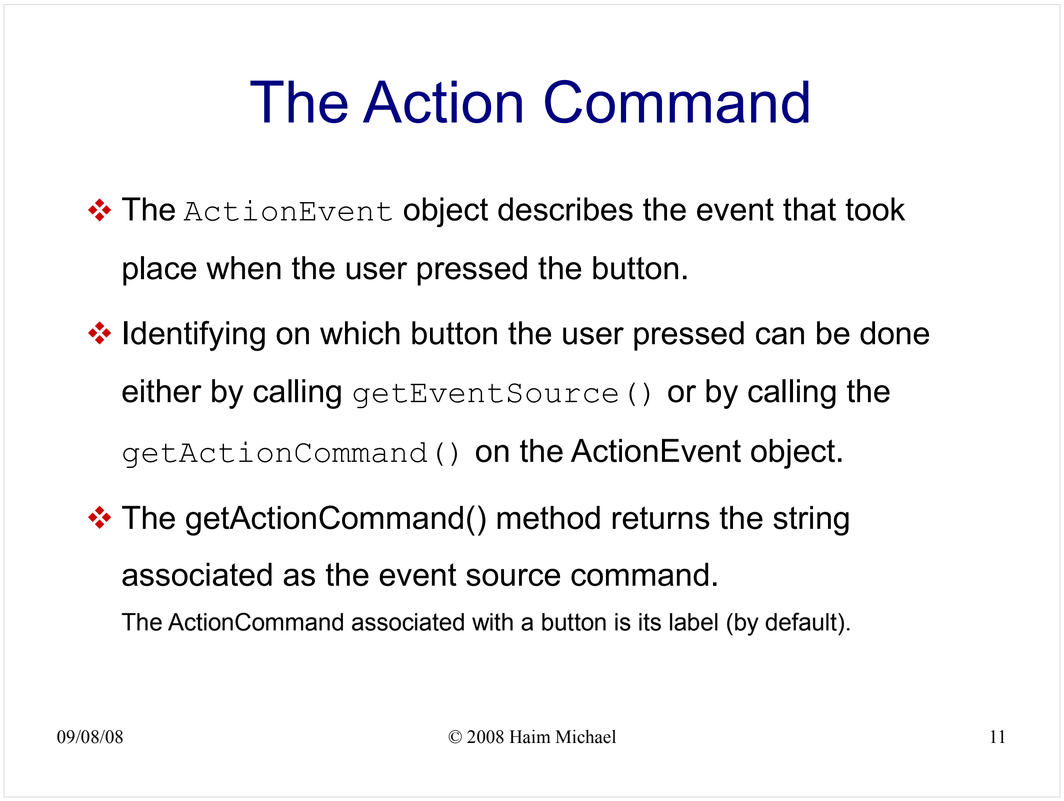
```
public void go()
{
    frame.setSize(500,400);
    frame.setVisible(true);
}
class MyButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = null;
        Object source = e.getSource();
        if(source==btOne)
        {
            text = "One";
        }
        else if(source==btTwo)
        {
            text = "Two";
        }
        textField.setText(text);
    }
}
```

09/08/08 © 2008 Haim Michael 9

The Events Delegation Model

```
        public static void main(String args[])
        {
            SimpleActionEventDemo demo = new SimpleActionEventDemo();
            demo.go();
        }
    }
```





The Action Command

- The `ActionEvent` object describes the event that took place when the user pressed the button.
- Identifying on which button the user pressed can be done either by calling `getEventSource()` or by calling the `getActionCommand()` on the ActionEvent object.
- The getActionCommand() method returns the string associated as the event source command.

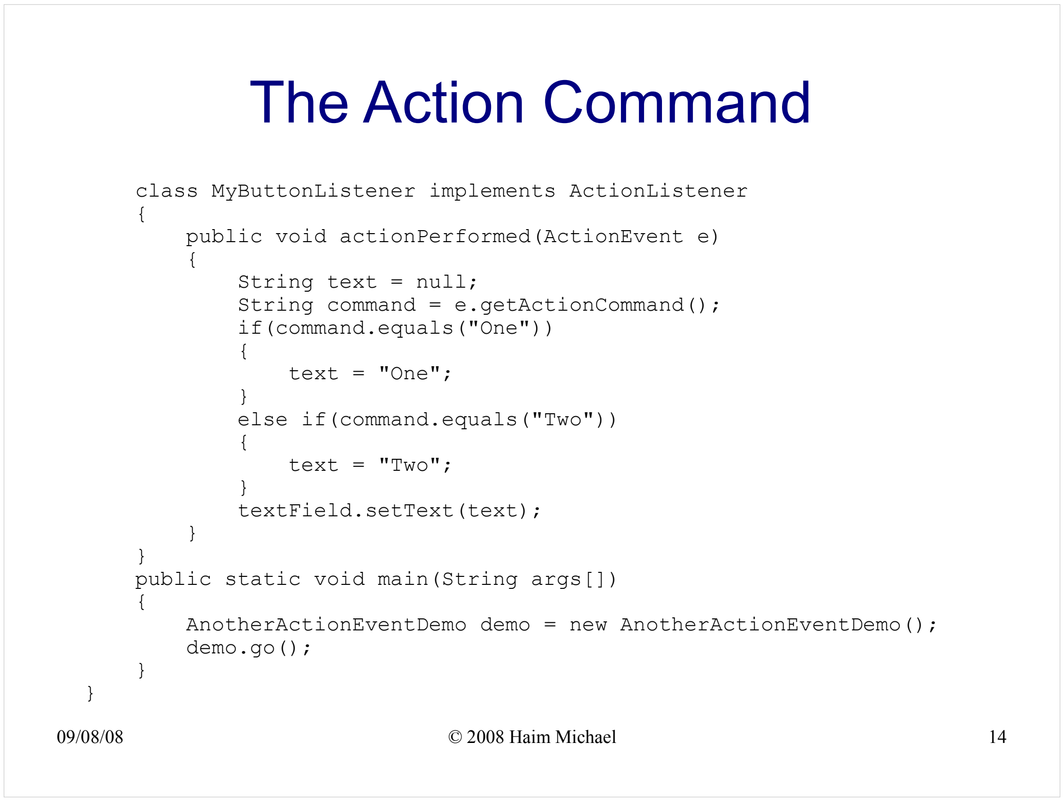
  The ActionCommand associated with a button is its label (by default).

The Action Command

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

public class AnotherActionEventDemo
{
    private JFrame frame;
    private JButton btOne,btTwo;
    private JTextField textField;
    private int counter;
    private ActionListener buttonListener;

    public AnotherActionEventDemo()
    {
        frame = new JFrame("Another ActionEvent Demo");
        btOne = new JButton("One");
        btTwo = new JButton("Two");
        textField = new JTextField(12);
```

The Action Command

```
        buttonListener = new MyButtonListener();
        btOne.addActionListener(buttonListener);
        btTwo.addActionListener(buttonListener);
        frame.setLayout(new FlowLayout());
        frame.add(btOne);
        frame.add(btTwo);
        frame.add(textField);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void go()
    {
        frame.setSize(500,400);
        frame.setVisible(true);
    }
```



The Action Command

```
        class MyButtonListener implements ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                String text = null;
                String command = e.getActionCommand();
                if(command.equals("One"))
                {
                    text = "One";
                }
                else if(command.equals("Two"))
                {
                    text = "Two";
                }
                textField.setText(text);
            }
        }
        public static void main(String args[])
        {
            AnotherActionEventDemo demo = new AnotherActionEventDemo();
            demo.go();
        }
    }
```

Multi Threaded Events Handling

- All Swing components are not thread safe.

  This was done in order to increase their efficiency and decrease the code complexity.

- Given this design we must ensure that every access to a Swing component must be done from a single one thread... from the event dispatcher thread.

  Calling the EventQueue.isDispatchThread() or the SwingUtilities.isEventDispatchThread() methods can assist us to verify the current thread.

09/08/08 © 2008 Haim Michael 15