

Components - JavaBeans

[Download as Power Point file for saving or printing.](#)

Components - Overview

- Component encapsulates a set of functions.
- Component-based software uses multiple, independent components.
- Loosely-coupled components require communication mechanism.
- JavaBeans define a component communication protocol.
- Will examine three communication mechanisms.
 1. Roll our own – to understand underlying approach
 2. Bound properties – a simple but limited JavaBean approach
 3. JavaBean protocol – a general, unrestricted JavaBean approach

Component (Class) Communication

- Component is a module that encapsulates a set of functions.
- Component-based software uses multiple, independent components.
- Loosely-coupled components require communication mechanism.
- Event should trigger communication notifying registered components.
- Notifying component must know method name in registered components.
- Implementing Java interface requires method definitions in component.
- Components register with event component to be notified of event.
- Event component calls the interface method of all registered components when event occurs.

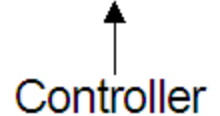
Basic Controller that registers and notifies ControllerListeners.

```
public class Controller {
    private Vector <ControllerListener>controllerListeners = new Vector <ControllerListener> ();

    public void addControllerListener(ControllerListener cl) {
        controllerListeners.addElement(cl);
    }
    private void notifyControllerListener( String message) {
        for (int i = 0; i < controllerListeners.size(); i++)
            controllerListeners.elementAt(i).controllerEvent( message );
    }
}
```

Application

Model



Controller

```
public interface ControllerListener {
    void controllerEvent( String message );
}
```

Basic Model. Notified of controllerEvent

```
class Model implements ControllerListener {
    public void controllerEvent( String message ) {
        System.out.println( message );
    }
}
```

Application using Controller and Model.

```
public class Application {  
    public static void main( String args[] ) {  
  
        Controller controller=new Controller();  
  
        Model model=new Model();  
  
        controller.addControllerListener( model );  
  
        controller.notifyControllerListener( "Hello World");  
  
    }  
}
```

```

public class Controller {
    private Vector <ControllerListener>controllerListeners =
        new Vector <ControllerListener> ();
    public void addControllerListener(ControllerListener cl) {
        controllerListeners.addElement(cl);
    }
    private void notifyControllerListener( String message) {
        for (int i = 0; i < controllerListeners.size(); i++)
            controllerListeners.elementAt(i).controllerEvent( message );
    }
}

```

```

public interface ControllerListener {
    void controllerEvent( String message );
}

```

```

class Model implements ControllerListener {
    public void controllerEvent( String message ) {
        System.out.println( message );
    }
}

```

Application

Model

Controller



```

Controller controller=new Controller();
Model model=new Model();
controller.addControllerListener( model );
controller.notifyControllerListener( "Hello World");

```

Application: Calculator

Calculator

View

Model

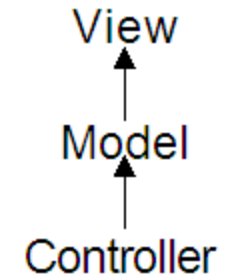
Controller



- Calculator control is through input buttons.
- Pressing 4, should update calculator model by 4.
- Should also update the view by 4.
- How to communicate between calculator controller and calculator model?
- How to communicate between calculator model and calculator view?
- Are calculator controller, model and view independent?
- Should controller *know* about view and model or vice-versa?
- When building a component, don't know future use.
- Standard communication between objects is key.

Model-View-Controller

Calculator



- Model-View-Controller (MVC) is architecture/pattern.
 - Controller: input buttons.
 - Model: calculation state.
 - View: display.
- Communication by notifying event listeners, indicated by arrows here.
- Model listens for Controller event, View listens for Model events.
- Controller notifies Model, Model notifies View.
- Controller, Model and View independent, no a priori knowledge of others.
- Standard communication between objects, really just method calls.

Calculator: Part of Controller that creates the visual keypad.

```
public class Controller extends Panel {
    private double accumulator=0.0, operand;
    private char operation;

    public Controller()
    {
        setLayout(new GridLayout(4,5));
        for (int i=0; i<=9; i++) makeButton((char)(i+48));
        makeButton('+');          makeButton('-');          makeButton('*');
        makeButton('/');          makeButton('=');          makeButton('R');
    }
    private void makeButton(char c) {
        final char label = c;
        Button button = new Button(label+"");
        add(button);
        button.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    notifyControllerListener( label );
                }
            }
        );
    }
}
```

Part of Controller that registers and notifies ControllerListeners.

```
public interface ControllerListener {  
    void controllerEvent(char c);  
}
```

```
public class Controller extends Panel {  
    private Vector <ControllerListener>controllerListeners = new Vector <ControllerListener> ();  
  
    public void addControllerListener(ControllerListener cl) {  
        controllerListeners.addElement(cl);  
    }  
  
    public void removeControllerListener(ControllerListener cl) {  
        controllerListeners.removeElement(cl);  
    }  
  
    private void notifyControllerListener(char c) {  
        for (int i = 0; i < controllerListeners.size(); i++)  
            controllerListeners.elementAt(i).controllerEvent(c);  
    }  
}
```

```

class Model implements ControllerListener {
    private double accumulator=0.0, operand;
    private char operation;
    public void controllerEvent(char c) {
        switch (c) {
            case '=' : equal(); break;
            case '+' : case '-' : case '*' : case '/' :
                operation = c;
                operand = accumulator;
                accumulator = 0.0;
                break;
            case 'R' : operation = c;
                accumulator = 0;
                operand = 0;
                break;
            default : accumulator = accumulator * 10+(int)c-48;
        }
        for (int i = 0; i < modelListeners.size(); i++)
            modelListeners.elementAt(i).modelEvent(accumulator+""");
    }
}

```

```

public interface ControllerListener {
    void controllerEvent(char c);
}

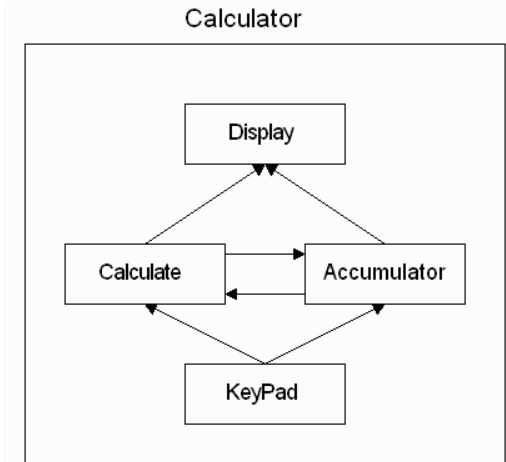
```

```

public interface ModelListener {
    void modelEvent(String s);
}

```

Problem: Build a calculator



- When button 4 pressed, should display 4.
- Should also update the accumulator with 4.
- How to communicate between button and display?
- How to communicate between button and accumulator?
- Are calculator button, accumulator and display independent?
- Should button *know* about display and accumulator or vice-versa?
- What other objects need to be informed when a button is pressed?
- When we build the button, don't know what objects will need the button.
- Standard communication between objects is key.

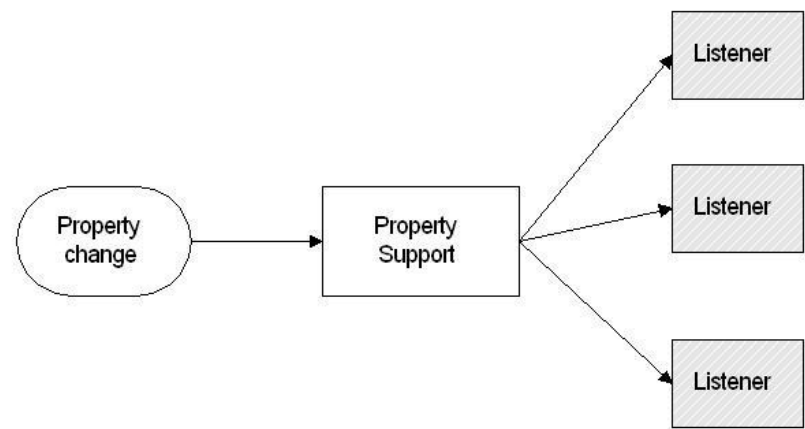
JavaBeans Overview

- Components are independent classes communicating via protocols.
- Example: calculator button and display independent. Display must be updated when button clicked. Communicate via protocol.
- JavaBeans define protocol for abstracting communication between objects.
- Protocol defines when/how/what data is passed between objects.
- A JavaBean is a class that conforms to the JavaBean protocol.
- Allows connecting objects via a predefined, standard interface.
- User (programmer) only defines data *source* and *destination* objects.
- JavaBean objects handle passing data to other JavaBean objects.

JavaBean Protocol

- **Property** – Attribute (instance variable) of object.
- **Naming conventions** for property accessor methods (*get* to read and *set* to write property value).
- **Notification** to property users when property value changes (*bound* property).
- **Event notification** by event source to list of event handlers.
- Ability by a property user to **veto** a property change (*constrained* property).
- **Customization** - allows Graphical User Interface Development environments to display and modify properties.
- **Persistence** of component by writing and reading properties to permanent storage.
- **Introspection** by a component into the methods and properties of another (i.e. examine component for specific naming patterns (e.g. *get* and *set*)).
- **Reflection**, essentially the reverse of introspection, where a component allows or assists the examination of properties and methods by following naming conventions (e.g. *get* and *set*) .

Bound Properties



- Property report change to *listeners*.
- Property owner uses *PropertyChangeSupport* object to notify listeners.
- Change is reported to listeners by calling the listener's *propertyChange* method.
- Property change updated simultaneously to all listeners.
- Beans with bound properties can also generate an *event* when a property changes, involving a public method in a bean when the property changes.
- A bound property requires a method for registering listeners to be notified that the bound property changed (e.g. `addDigitListener`).
- To notify listeners, the method *firePropertyChange* is called with three parameters, a string defining the name of the bound property, the bound property *old* value and *new* value. If old ¹ new value the property listener is notified of the change.
- While simple to use, *PropertyChangeListeners* are limited.
- For example, it gets a little messy when listening to more than one property change at a time, requiring the listener to know and examine the source of the property.

The basics: Events and Listeners



Ouch

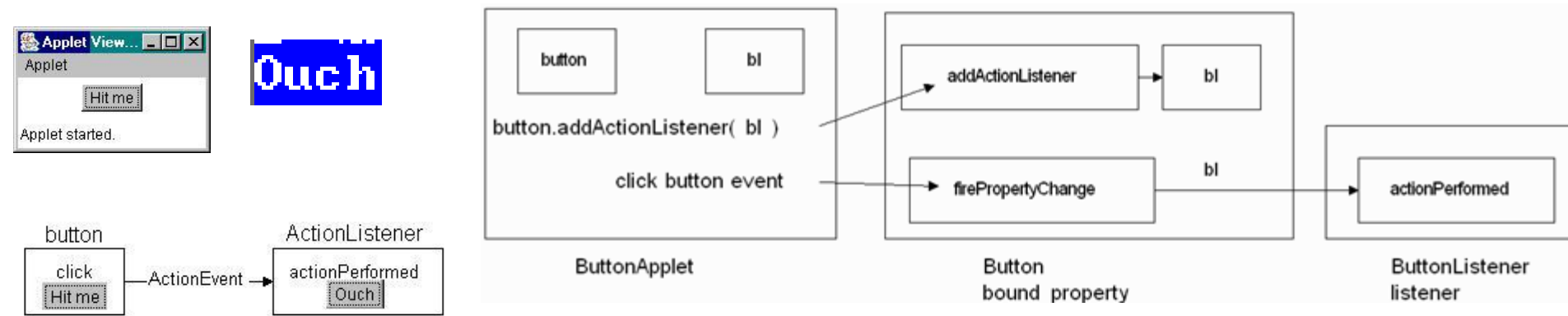
```
import java.applet.*;
import java.awt.*;
public class ButtonApplet extends Applet {

    public void init() {
        Button button = new Button("Hit me");
        add(button);
    }

    public boolean action (Event e, Object o) {
        if (o.equals("Hit me"))
            System.out.println("Ouch");
        return true;
    }
}
```

- No Beans
- Applet with a button
- Event when clicked, executes *action*
- To handle event, we must figure out what happened (i.e. the *if* statement)
- Not following an abstract protocol
- Hardwired button event in *action*

Events and Listeners: a protocol



```
public class ButtonApplet extends Applet {  
    public void init() {  
        Button button = new Button("Hit me");  
        buttonListener bl = new ButtonListener( );  
        button.addActionListener(bl);  
        add(button);  
    }  
}  
  
class ButtonListener implements ActionListener {  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Ouch"); }  
}
```

- Event and listener
- Define ButtonListener which must implement *actionPerformed*
- Event when button clicked
- Listener notified by Button execution of *actionPerformed*

Example - connect a keypad to a display

Example

- Keypad - Has a property, a single *digit*. When digit (property) changes, any listeners (i.e. Display) of that property are notified.
- Display - Notified when a property to which it is listening changes.

Keypad connected to Display in the main class by calling:

- `keypad.addDigitListener(display);`

In the following example:

1. *main* defines a Keypad and Display object.
2. *main* registers a listener (Display) with property source (KeyPad).
3. *main* calls KeyPad *setDigit()* to set the property.
4. KeyPad *setDigit()* calls *firePropertyChange()* to notify all listeners that a property value (digit) has changed.
5. Display *propertyChange()* is called when the property changes, receiving the new property value, which is printed.

```

public class Example {
    public static void main(String a[]) {
        Keypad keypad=new Keypad();
        Display display=new Display();
        keypad.addDigitListener(display );
        keypad.setDigit("0");
        keypad.setDigit("1");
        keypad.setDigit("2");
    }
}

```

```

import java.beans.*;
class Display implements
        PropertyChangeListener {
    public void propertyChange(
        PropertyChangeEvent e) {
        System.out.println( "Display "+
            e.getNewValue() );
    }
}

```

```

import java.beans.*;
class Keypad {
    private PropertyChangeSupport pcDigit = new PropertyChangeSupport(this);
    private String digit=null;
    public void addDigitListener(PropertyChangeListener listener) {
        pcDigit.addPropertyChangeListener(listener);
    }
    public void setDigit(String newDigit) {
        String oldDigit = this.digit;
        this.digit=newDigit;
        pcDigit.firePropertyChange("digit", oldDigit, newDigit);
    }
}

```

Execution

Display 0
 Display 1
 Display 2

Discussion of Example, Keypad and Display classes

Example

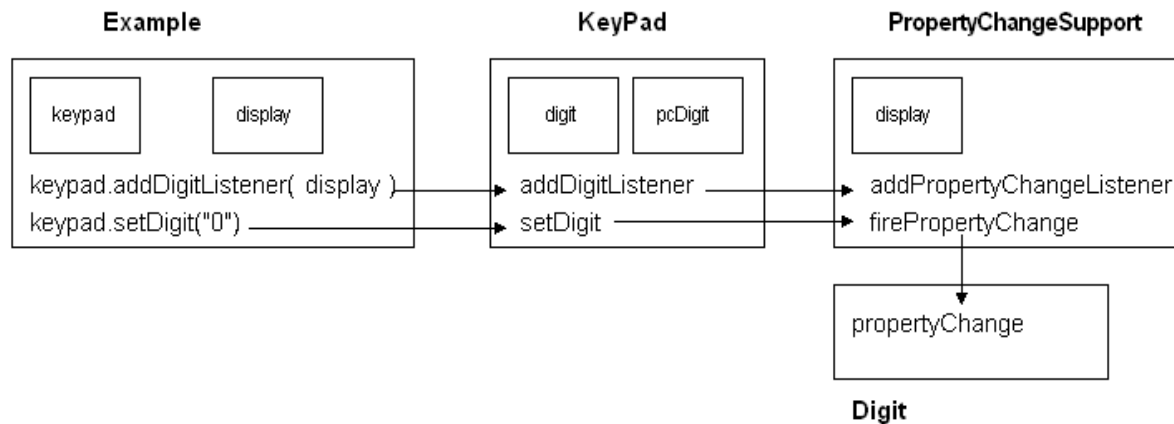
1. `Keypad keypad=new Keypad();`
2. `Display display=new Display();`
3. **`keypad.addDigitListener(display);`** *Registers display object as keypad listener*
4. `keypad.setDigit("0");` *Sets the 'digit' property of a Keypad*

Keypad

1. `private PropertyChangeSupport pcDigit = new PropertyChangeSupport(this);` *JavaBean*
2. `private String digit=null;` *Keypad 'digit' property*
3. `public void addDigitListener(PropertyChangeListener l) { pcDigit.addPropertyChangeListener(l); }` *Called to register property listeners*
4. `public void setDigit(String newDigit) {` *Mutator for 'digit' property*
5. `String oldDigit = this.digit;`
6. `this.digit=newDigit;`
7. **`pcDigit.firePropertyChange("digit", oldDigit, newDigit);`** *Notify listener of 'digit' change*

Display

1. `class Display implements PropertyChangeListener {` *Must define propertyChange method*
2. `public void propertyChange(PropertyChangeEvent e) {` *Called by firePropertyChange*
3. `System.out.println("Display "+ e.getNewValue());` *e is the 'digit' property object, a String*



Example

1. `Keypad keypad=new Keypad();`
2. `Display display=new Display();`
3. **`keypad.addDigitListener(display);`**
4. `keypad.setDigit("0");`

Keypad

1. `private PropertyChangeSupport pcDigit = new PropertyChangeSupport(this);` *JavaBean*
2. `private String digit=null;` *Keypad 'digit' property*
3. `public void addDigitListener(PropertyChangeListener l) { pcDigit.addPropertyChangeListener(l); }`
Called to register property listeners
4. `public void setDigit(String newDigit) {` *Mutator for 'digit' property*
5. `String oldDigit = this.digit;`
6. `this.digit=newDigit;`
7. `pcDigit.firePropertyChange("digit", oldDigit, newDigit);` *Notify listener of 'digit' change*

Display

1. `class Display implements PropertyChangeListener {` *Must define propertyChange method*
2. `public void propertyChange(PropertyChangeEvent e) {` *Called by firePropertyChange*
3. `System.out.println("Display "+ e.getNewValue());` *e is the 'digit' property object, a String*

Java Listener Naming Conventions Example

Interface `MouseListener`

All Superinterfaces:

[EventListener](#)

All Known Subinterfaces:

[MouseListener](#)

All Known Implementing Classes:

[AWTEventMulticaster](#), [MouseDragGestureRecognizer](#), [MouseAdapter](#), [DefaultCaret](#), [BasicButtonListener](#)

Method Summary	
void	mouseClicked (MouseEvent e) Invoked when the mouse has been clicked on a component.
void	mouseEntered (MouseEvent e) Invoked when the mouse enters a component.
void	mouseExited (MouseEvent e) Invoked when the mouse exits a component.
void	mousePressed (MouseEvent e) Invoked when a mouse button has been pressed on a component.
void	mouseReleased (MouseEvent e) Invoked when a mouse button has been released on a component.

Java Listener Naming Conventions

Exercise 1 - Event listeners and adapters follow a naming convention where:

`void addxxxListener(xxxListener object)` defines the method to register a listener with a component.

An interface is defined as:

`xxxListener`

and event methods and classes are defined as:

`void xxxevent(xxxEvent e)`

For example, the mouse is:

`void addMouseListener(MouseListener object)`

`void mouseClicked(MouseEvent e)`

`void mouseExited(MouseEvent e)`

1. Give the code pattern to implement a `MouseListener` named *ouchListener* that overrides the *mouseClicked* method to print "Ouch" on standard output using `System.out.print("Ouch");`.
2. Give the code to construct and register an *ouchListener* object.

Exercise 1 - Java Listener Naming Conventions

Event listeners and adapters follow a naming convention where:

```
void addxxxListener( xxxListener object)
```

defines the method to register a listener with a component.

An interface is defined as:

```
xxxListener
```

and event methods and classes are defined as:

```
void xxxevent( xxxEvent e)
```

For example, the mouse is:

```
void addMouseListener( MouseListener object)
```

Method to register a Mouse listener

```
void mouseClicked( MouseEvent e)
```

Called on mouseClicked event

```
void mouseExited(MouseEvent e)
```

Called on mouseExited event

1. Give the code pattern to implement a `MouseListener` named `ouchListener` that overrides the `mouseClicked` method to print "Ouch!" on standard output using `System.out.println("Ouch!");`.

```
class ouchListener implements MouseListener {  
    public void mouseClicked(MouseEvent e) { System.out.println("Ouch"); }  
}
```

2. Give the code to construct and register an `ouchListener` object.

```
button.addMouseListener( new ouchListener() );
```

Inner Classes and Listeners

- Obviously in a GUI with many buttons, sliders, etc. that each generates multiple events, a single *action* method would be very complicated, needing to check the event against every possible event case.
- A more scalable approach is to define an individual handler or *listener* for each event as in the earlier example.
- Buttons define the single *ActionEvent* event which must be connected to the *listener* code that handles the event.
- One approach is to define a *listener* class for each type of event, in the example the *ActionEvent* listener is implemented in the *ButtonListener* class.
- Because *ButtonListener* implements *ActionListener* it must define the *actionPerformed* method which is invoked whenever the *ActionEvent* occurs.
- The *ButtonListener* object *bl* must be registered as a listener with the *button*.
- Writing a new listener *class* just to implement the event handling method (e.g. *actionPerformed*) is tedious, clutters the name space, and is error prone.
- One simplification is *inner classes* or classes that are defined in the scope of another class.

Inner classes (bottom) preferred over defining new class that is used only once (top).

```
buttonListener bl = new buttonListener();  
button.addActionListener(bl);
```

```
class buttonListener implements ActionListener  
{  
    public void actionPerformed(ActionEvent e) { System.out.print("Ouch");}  
}
```

```
button.addActionListener(new ActionListener( )  
    {  
        public void actionPerformed(ActionEvent e) { System.out.print("Ouch"); }  
    }  
);
```

```

public class ButtonApplet extends Applet {
    public void init() {
        Button button = new Button("Hit me");
        buttonListener bl = new ButtonListener( );
        button.addActionListener(bl);
        add(button);
    }
}

class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent e) { System.out.print("Ouch"); }
}

```

```

public class ButtonApplet extends Applet {
    public void init() {
        Button button = new Button("Hit me");
        button.addActionListener( new ActionListener( ) {
            public void actionPerformed(ActionEvent e) { System.out.print("Ouch"); }
        } );
        add(button);
    }
}

```

Exercise 2 - Exercise 1 defined a class to listen for *mouseClicked* events, solution given again below. Implement the corresponding inner-class.

```
button.addMouseListener( new ouchListener() );  
  
class ouchListener implements MouseListener {  
    public void mouseClicked(MouseEvent e) {  
        System.out.println("Ouch")  
    }  
}
```

Answer

```
button.addMouseListener( new MouseListener() {  
    public void mouseClicked(MouseEvent e) { System.out.println("Ouch"); }  
}  
);
```

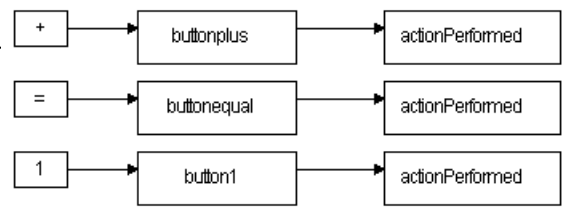
Calculator Inner Classes (Left) vs Switch Statement (Right)

```

public class Calculator extends UserInterface {
public Calculator() {
    Button button1 = new Button("1"),
           buttonequal = new Button("="),
           buttonplus = new Button("+");
button1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        setAccumulator(getAccumulator() * 10+1);
        updateDisplay(getAccumulator()+""); } } );
buttonequal.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        equal();
        updateDisplay(getAccumulator()+"");} } );
buttonplus.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        setOperation('+');
        setOperand(accumulator);
        setAccumulator(0.0);
        updateDisplay(getAccumulator()+"");} } );
}
    
```

```

public class Calculator extends UserInterface {
public Calculator() {
    addButton("1");
    addButton("+");
    addButton("=");
}
public void onClick(char c) {
    switch (c) {
        case '1': accumulator = accumulator *10+1;
                    break;
        case '=': equal();
                    break;
        case '+': setOperation('+');
                    setOperand(accumulator);
                    setAccumulator(0.0);
                    break;
        default : ;
    }
    updateDisplay(getAccumulator()+"");
}
}
    
```



Question: What's the advantage of inner classes here?

Counter Bean Example

- A JavaBean is a regular, programmer defined Java class that follows a standard protocol for naming methods that access internal attributes (properties) and for event notification.
- By following the protocol, other JavaBeans or programs can interact with a JavaBean through the standard interface.
- One key advantage is that the JavaBean is self-contained, communication is restricted through the consistently named protocol methods.
- This allows JavaBeans to implement large, independent components such as a word processor or graphing component or small components such as buttons that can communicate with other independent components.
- The component can be integrated into an application more easily than a class, often with little or no programming (when using a GUID).
- One obvious part of the naming protocol defines *mutator* methods for **setting** and *accessor* methods for **getting** the values of JavaBean class attributes.
- In the Counter bean, the *maxValue* attribute has two methods, the accessor and mutator methods of:
 1. **setMaxValue** Sets the value of attribute *maxValue*; a mutator method
 2. **getMaxValue** Gets the value of attribute *maxValue*; an accessor method

Counter Bean Example

```
public class Counter extends Panel {
    private long count=0;
    private Label label;
    private long maxValue=0;
    public void setMaxValue( long max ) { maxValue = max; }
    public long getMaxValue() { return maxValue; }
    public Counter() {
        setBackground(Color.blue);      setForeground(Color.white);
        label = new Label(""+count);    add(label);
    }
    public void increment () {
        if (count < maxValue) {
            count++;
            label.setText(count+" ");
        }
        else label.setText("!!");          // count exceeds maximum
    }
}
```



Counter Bean Example

Clicking button event:

- Executes actionPerformed
- Calls counter.increment()
- [Try it.](#)



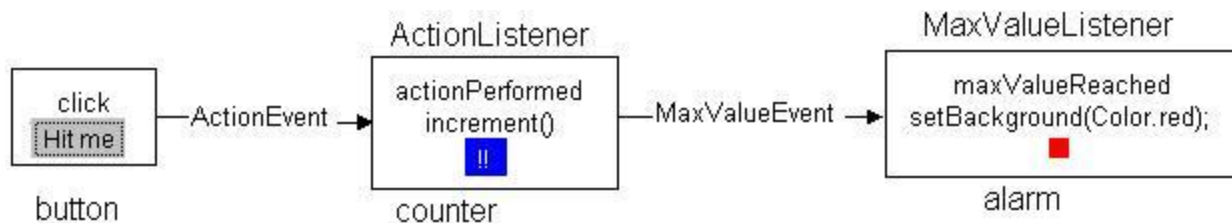
```
public class ButtonApplet2 extends Applet {  
    Button button= new Button("Hit me");  
    Counter counter = new Counter();  
    public void init( ) {  
        button.addActionListener(new ActionListener( ) {  
            public void actionPerformed(ActionEvent e) { counter.increment( ); }  
        });  
        counter.setMaxValue(4);  
        add(button);  
        add(counter);  
    }  
}
```



JavaBean Protocol Example



- Application that counts each time button clicked (blue).
- Connected to monitor of the counter value, normally green.
- When count value exceeded, monitor signals an alarm, turning red.
- Using Bean protocol can have 3 independent objects:
 - Button
 - Counter
 - Alarm
- These happen to be graphical (visible) objects



JavaBean Protocol Example

JavaBean protocol has at a minimum four parts for event xxx:

1. a registration method used by event listeners named:
void add**xxx**Listener(**xxx**Listener object)
2. An interface defining event listeners named:
xxxListener
3. Event classes defining events named:
xxxEvent
4. Event handler methods of a listener called when an event occurs as:
void **xxx***EventHandler*(**xxx**Event e)

JavaBean Protocol Example

JavaBean protocol has at a minimum four parts for event **MaxValue**:

1. Counter registration method used by event listeners named:
void add**MaxValue**Listener(**MaxValue**Listener object)
2. An interface defining event listeners named:
MaxValueListener
3. Event classes defining events named:
MaxValueEvent
4. Event handler methods of a listener called when an event occurs as:
void **maxValueReached**(**MaxValue**Event e)

JavaBean Protocol Example – The Components

```
public class MaxValueEvent extends java.util.EventObject {  
    public MaxValueEvent (Object object) { super( object ); }  
}
```

```
public interface MaxValueListener extends java.util.EventListener {  
    void maxValueReached (MaxValueEvent m);  
}
```

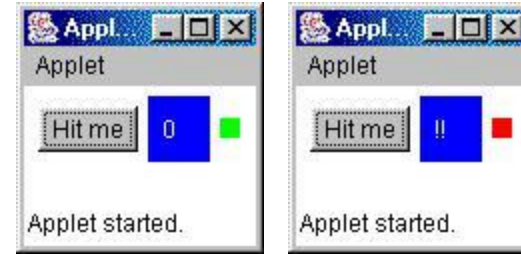
```
public class Alarm extends Panel implements MaxValueListener {  
    public Alarm () { setBackground(Color.green); }  
    public void maxValueReached (MaxValueEvent e) { setBackground(Color.red); }  
}
```

```
public class Counter extends Panel {  
    private MaxValueListener ml;  
    public synchronized void addMaxValueListener(MaxValueListener ml) { this.ml = ml; }  
    public void increment () {  
        if (count < maxValue) label.setText(++count + " ");  
        else { label.setText("!!");  
            MaxValueEvent mve = new MaxValueEvent(this);  
            ml.maxValueReached( mve );  
        }  
    }  
}
```

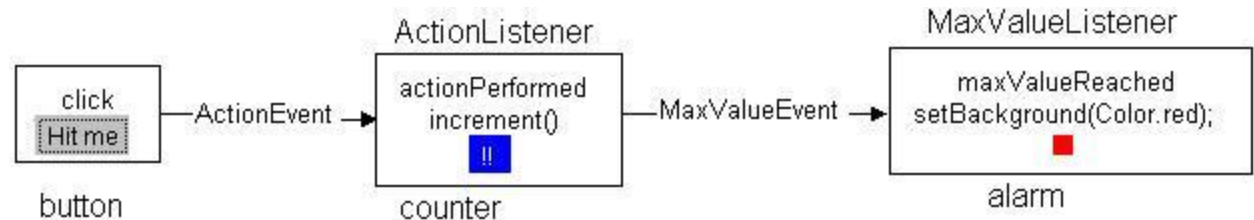
JavaBean Protocol Example – Connecting the Components

```
public class ButtonApplet extends Applet {  
    Button button = new Button("Hit me");  
    Counter counter = new Counter();  
    Alarm alarm = new Alarm();
```

```
    public void init() {  
        button.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent e) { counter.increment(); } } );  
        counter.addValueListener(alarm);  
        add(button);  
        add(counter);  
        add(alarm);  
    }  
}
```



```
// Inner-class to  
// connect button to counter  
counter.increment(); } } );  
// Connect counter to alarm
```



Using Bean protocol makes for some additional effort to follow protocol.
Creates reusable components that follow standard interfacing rules.
Require less effort and knowledge to reuse objects.

Conclusion

- JavaBean protocol is a framework for defining interfacing of independent components.
- Example use: create new spell-checker for a JavaBean word processor, graphical user-interface icon and user interaction window.
- Much more not covered:
 - **Reflection** - allows the discovery of a bean's contents, for example the names of public functions, properties, etc.
 - **Introspection** - the process which queries a bean to discover a bean's contents using reflection.
 - **BeanInfo** - A special object that assists in describing a bean properties, icon, public methods, etc. simplifying introspection.
- See [JavaBeans](#) page for complete examples and a more complete discussion.