

Events in JavaBeans

An Event Set consists of:

- **An Event Listener Interface:**

This interface extends *java.util.EventListener* interface, and defines one or more methods that must be implemented by a class that wishes to receive this event.

- **An Event Object**

An **event object** is passed from the event source to the listener. It extends *java.util.EventObject* or one of its subclasses.

- **The Event Registration Methods**

These are **add-** or **remove-listener** methods, that are implemented by the event generating component. The names of these methods must conform to the JavaBeans naming patterns.

Predefined Event Sets

The *java.awt* package provides several predefined event sets, for example, the *key* event set consists of:

The *KeyListener* interface, with methods *keyPressed()*, *KeyReleased()*, and *keyTyped()*

The *KeyEvent* event object class.

A component that generates *key* events must define the *addKeyListener(KeyListener kl)* and *removeKeyListener(KeyListener kl)* registration methods.

Steps to implement multi-cast events in a JavaBean

- Select a **name** *<eventName>*, for the event set, e.g. *Timer*
- Define an **event object** class, *<eventName>Event*, derived from *java.util.EventObject*. e.g. *TimerEvent*.

Define any fields needed to hold **state** information for the event.

Include a **constructor** that takes at least one argument of type **object**, calls the constructor of its parent event object first, and then initializes any fields. e.g.

```

public class TimerEvent extends EventObject {

    private int tickCount;

    public TimerEvent(Object source) {
        super(source);
        tickCount = 0;
    }

    public int getTickCount() {
        return tickCount;
    }

    // ...
}

```

- Define the **event listener interface**, *<eventName>Listener*, i.e., *TimerListener*, extending *java.util.EventListener* interface. Define one or more operations in this interface. These operations should be passed *<eventName>Event* in the argument list, along with any other arguments that your design dictates. These operations will be implemented by methods of the event listening classes (event handling methods). Typically, these event listening classes are *local anonymous inner classes*.

In our example, the *TimerListener* interface could be something like:

```

public interface TimerListener extends EventListener {
    public void timerTick(TimerEvent event);
    public void timerStart(TimerEvent event);
    public void timerStop(TimerEvent event);
}

```

- Implement **event registration methods** in the event source component, these registration methods must follow the JavaBeans naming patterns.

Multicast add-listener registration method:

```

public void add<eventName >Listener (<eventName>Listener aListener);

```

e.g. *public synchronized void addTimerListener(TimerListener aListener);*

keyword, *synchronized*, is used to protect from race and deadlock problems in a multithreaded environment.

Remove -listener registration method:

```

public void remove<eventName >Listener (<eventName>Listener aListener);

```

e.g. *public synchronized void removeTimerListener(TimerListener aListener);*

keyword, *synchronized*, is used to protect from race and deadlock problems in a multithreaded environment.

- **Sending an event to listeners**

It is the responsibility of the event emitting class to determine when an event has occurred. When a particular event occurs, the event emitting class must :

1. Create an **event object**, by passing its “**this**” reference to the constructor of the event object. Set the state of the event object if needed.
2. Loop through all the objects that have registered themselves as being interested in listening to this particular event, and invoke one or more operations defined in the event listening interface. It passes the event object as an argument to these operations. For example, component, *Clock*, that emits *Timer* events might look something like:

```
public class Clock {  
  
    private Vector listenerList = new Vector();  
  
    public synchronized void addTimerListener(TimerListener tl) {  
        // add tl to listenerList  
    }  
    public synchronized void removeTimerListener (TimerListener  
tl){  
        // remove tl from listenerList  
    }  
  
    public processTimerTick() {  
        TimerEvent ev = new TimerEvent(this);  
        // set state of event object, ev, if needed  
        For(int i = 0; i < listenerList.size(); ++i) {  
            ((TimerListener)listenerList.elementAt(i)).timerTick(ev);  
        }  
    }  
  
    public void run() {  
        //  
        // at each tick invoke this.processTimerTick()  
        //  
    }  
  
}
```