18 JavaBeans and Bean Events

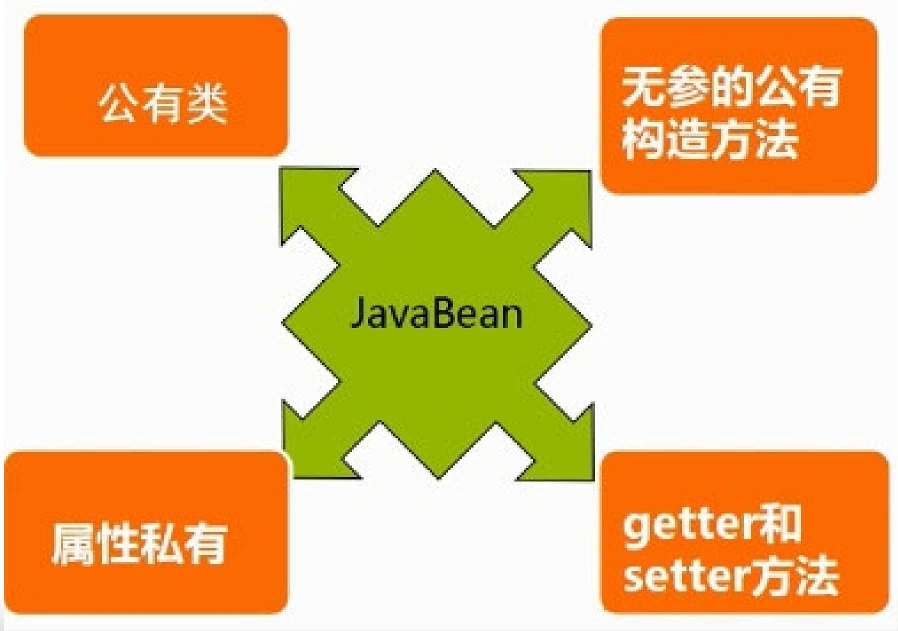
Chapter 32@8e

What is JavaBean?

A JavaBeans component is **a serializable public class with a public no-arg constructor**.

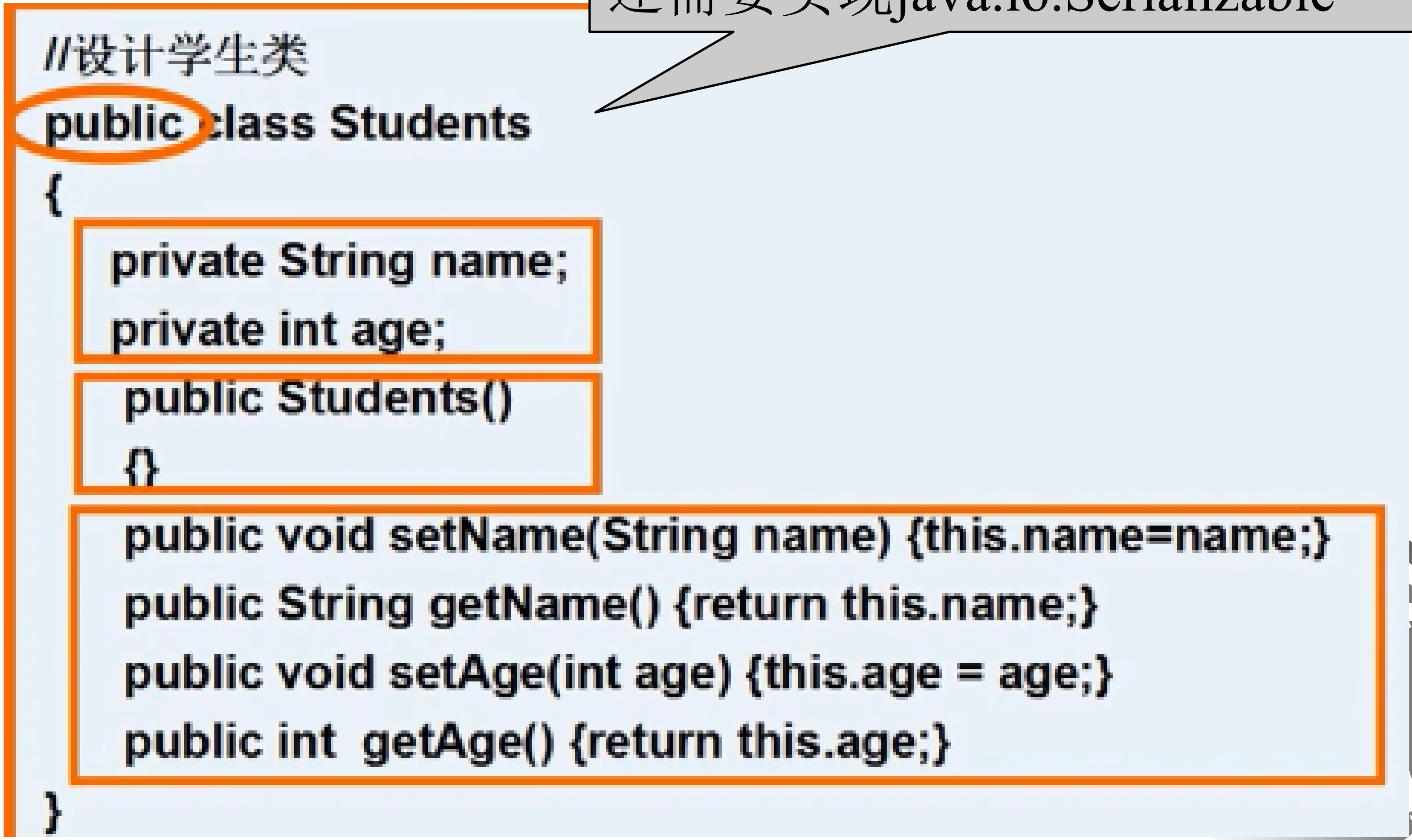
Javabeans就是一个普通的java类，但符合某种特定规范。使用Javabeans的好处是解决代码重复编写，功能区分明确，提高了代码的可维护性。

遵循某种设计原则



What is JavaBean?

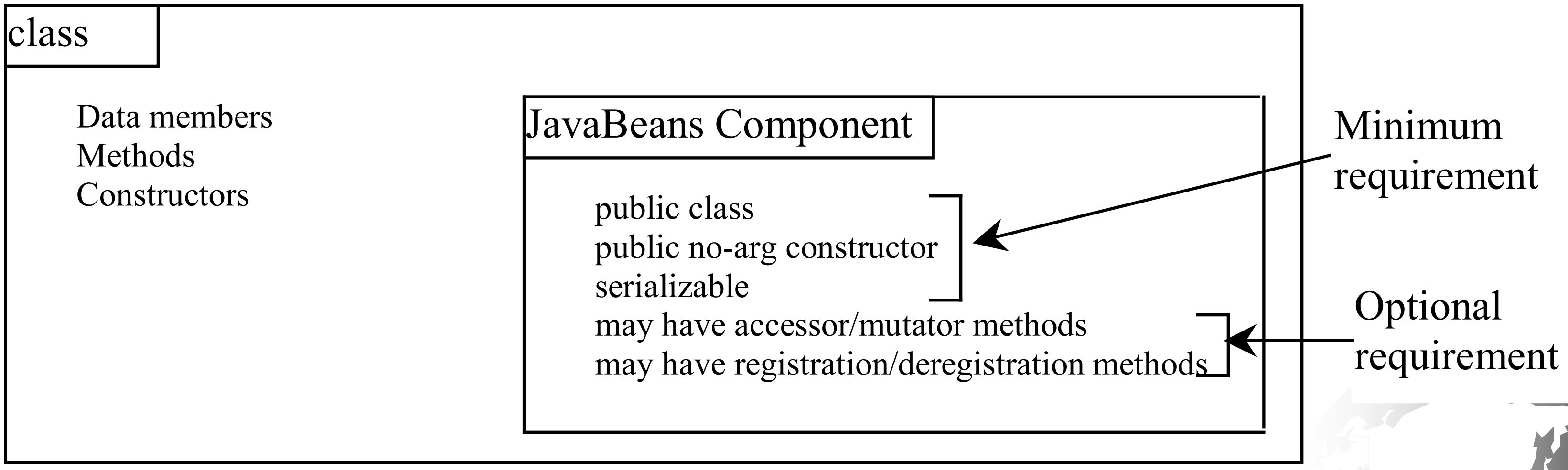
还需要实现java.io.Serializable



```
//设计学生类
public class Students
{
    private String name;
    private int age;
     public Students()
     {}
     public void setName(String name) {this.name=name;}
     public String getName() {return this.name;}
     public void setAge(int age) {this.age = age;}
     public int  getAge() {return this.age;}
}
```

What is JavaBean?

Every GUI class is a JavaBeans component, because (1) it is a public class; (2) it has a public no-arg constructor; (3) It is an extension of java.awt.Component, which implements java.io.Serializable.



class

Data members
Methods
Constructors

JavaBeans Component

public class
public no-arg constructor
serializable
may have accessor/mutator methods
may have registration/deregistration methods

Minimum requirement

Optional requirement

Why JavaBeans?

The JavaBeans technology was developed to enable the programmers to rapidly build applications by **assembling objects and test them during design time**, thus **making reuse of the software more productive**.

JavaBeans is a software component architecture that extends the power of the Java language by enabling well-formed objects to be **manipulated visually at design time** in a pure Java builder tool, such as JBuilder and NetBeans.

JavaBeans Properties and Naming Patterns

- The get method is named

  get<PropertyName>(),

  which takes no parameters and returns an object of the type identical to the property type.

- For a property of boolean type, the get method should be named

  is<PropertyName>(),

  which returns a boolean value.

- The set method should be named

  set<PropertyName>(newValue),

  which takes a single parameter identical to the property type and returns void.

Properties and Data Fields

Properties describe the state of the bean. Naturally, data fields are used to store properties. However, a bean property is not necessarily a data field. For example, in the MessagePanel class in Example 12.5 in Chapter 13, you may create a new property named messageLength that represents the number of the characters in message. The get method for the property may be defined as follows:

```
public int getMessageLength() {
  return message.length();
}
```

NOTE: A property may be read-only with a get method but no set method, or write-only with a set method but no get method.

Bean Events

A bean may communicate with other beans. The **Java event delegation model** provides the foundation for beans to send, receive, and handle events.

When something happens to a bean, such as a mouse click on a javax.swing.JButton bean, an event object is created to encapsulate information pertaining to the event. The bean passes the event object to the interested beans for the event to be processed.

Events are typically **generated** by Java GUI components, such as javax.swing.JButton, but **are not limited** to GUI components. This section introduces the development of custom events and the beans that can generate events.

The Event Delegation Model
事件委托模型

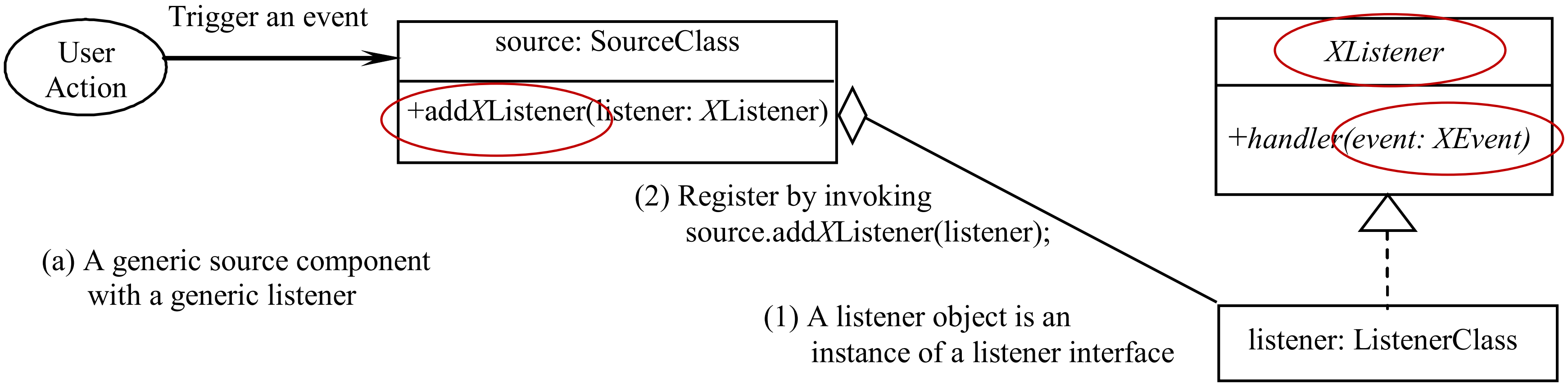
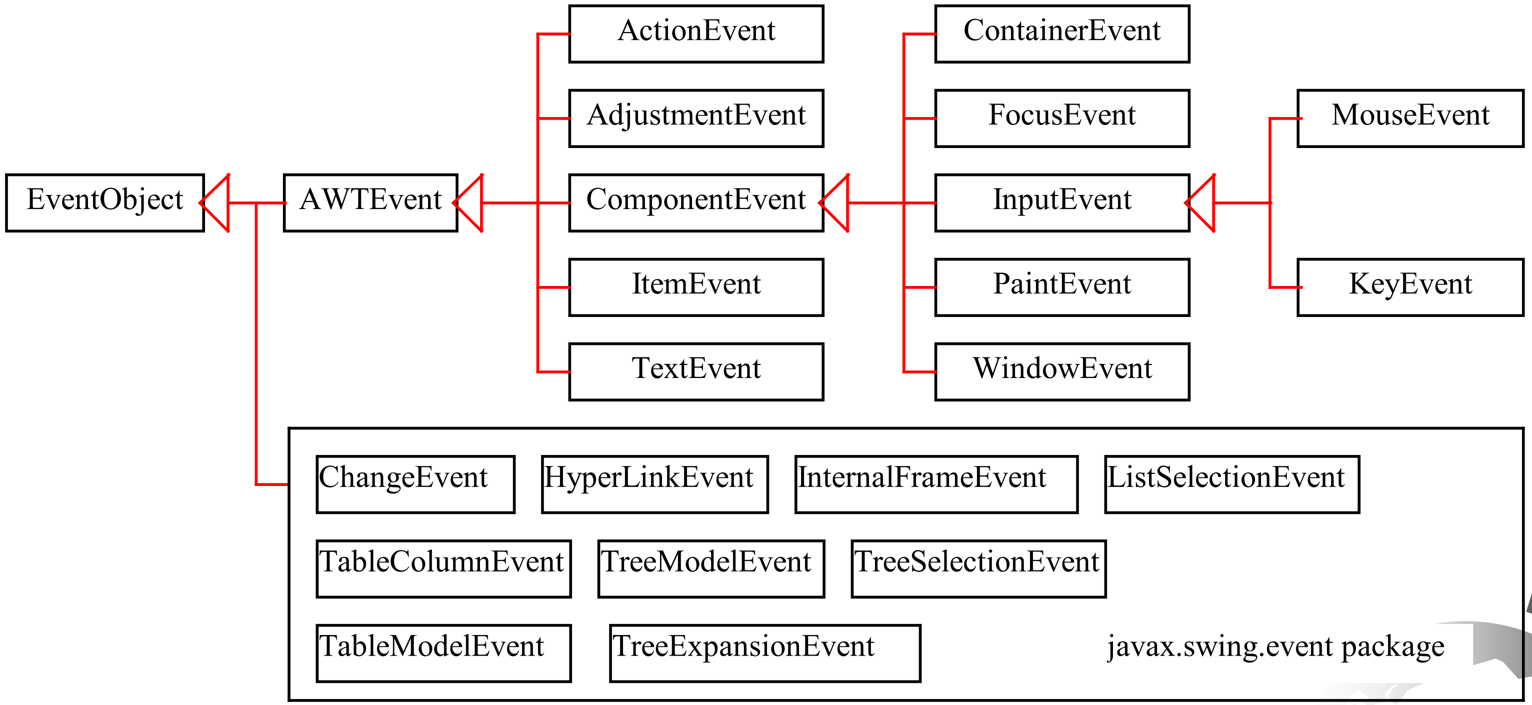

Trigger an event
User Action
source: SourceClass
+addXListener(listener: XListener)
XListener
+handler(event: XEvent)
(2) Register by invoking
source.addXListener(listener);
(a) A generic source component
with a generic listener
(1) A listener object is an
instance of a listener interface
listener: ListenerClass

Figure 15.3

Predefined Event Pairs
(Event Classes and Listener Interface)

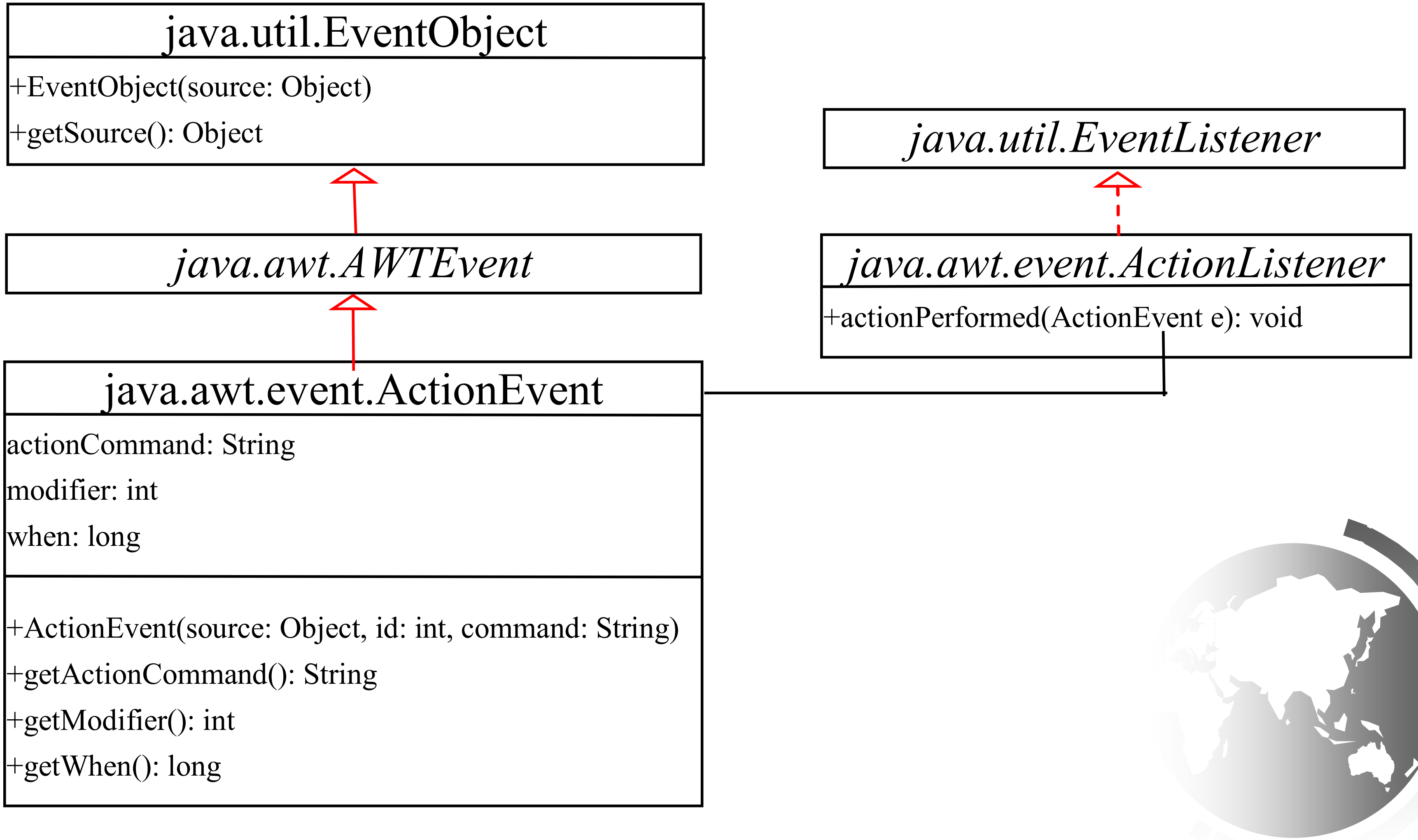


Examples:

ActionEvent/ActionListener

AdjustmentEvent/AdjustmentListener

Examples of Event Pairs



java.util.EventObject

+EventObject(source: Object)

+getSource(): Object

java.awt.AWTEvent

java.awt.event.ActionEvent

actionCommand: String

modifier: int

when: long

+ActionEvent(source: Object, id: int, command: String)

+getActionCommand(): String

+getModifier(): int

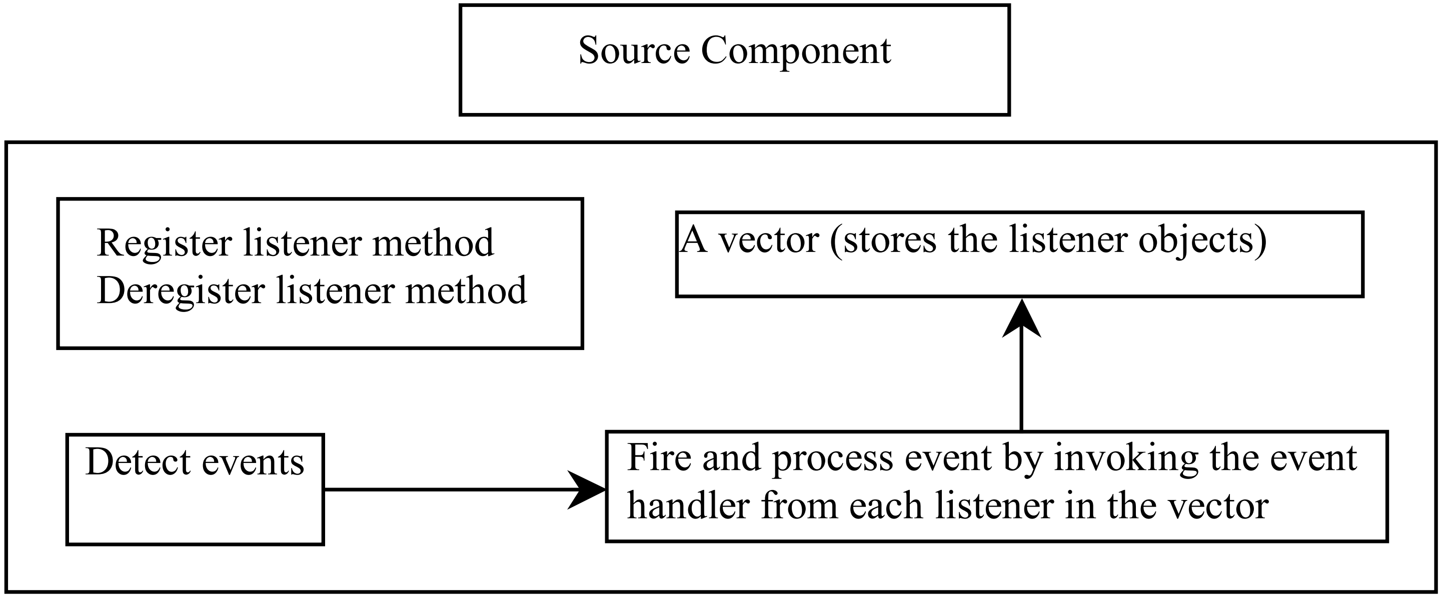
+getWhen(): long

java.util.EventListener

java.awt.event.ActionListener

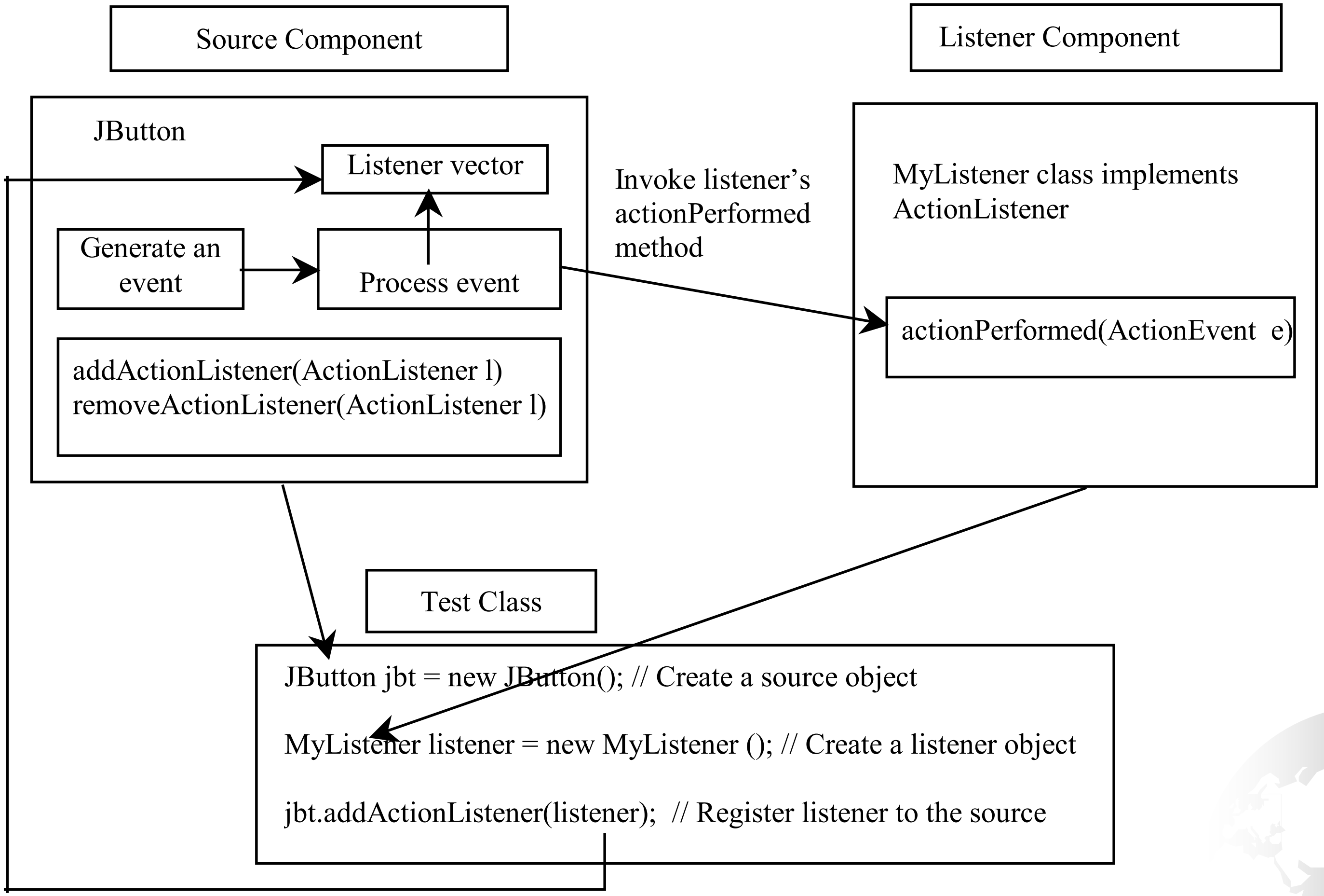
+actionPerformed(ActionEvent e): void

Source Components


Source Component
Register listener method
Deregister listener method
A vector (stores the listener objects)
Detect events
Fire and process event by invoking the event handler from each listener in the vector

The source component detects events and processes the events by invoking the event listeners' handler.

Listener Components



Source Component

JButton

Listener vector

Generate an event

Process event

addActionListener(ActionListener l)
removeActionListener(ActionListener l)

Invoke listener's actionPerformed method

Listener Component

MyListener class implements ActionListener

actionPerformed(ActionEvent e)

Test Class

```
JButton jbt = new JButton(); // Create a source object

MyListener listener = new MyListener (); // Create a listener object

jbt.addActionListener(listener);  // Register listener to the source
```

The listener is registered with the source, and the source invokes the listener's handler to process the event.

Creating Custom Source Components

Unicast Registration Methods:

A source component must have the appropriate registration and deregistration methods for adding and removing listeners. Events can be unicasted (only one object is notified of the event) or multicasted (each object in a list of listeners is notified of the event). The naming pattern for adding a unicast listener is

```
public void add<Event>Listener(<Event>Listener l)
  throws TooManyListenersException;
```

Creating Custom Source Components

Multicast Registration Methods:

The naming pattern for adding a multicast listener is the same, except that it does not throw the TooManyListenersException.

```
public void add<Event>Listener(<Event>Listener l)
```

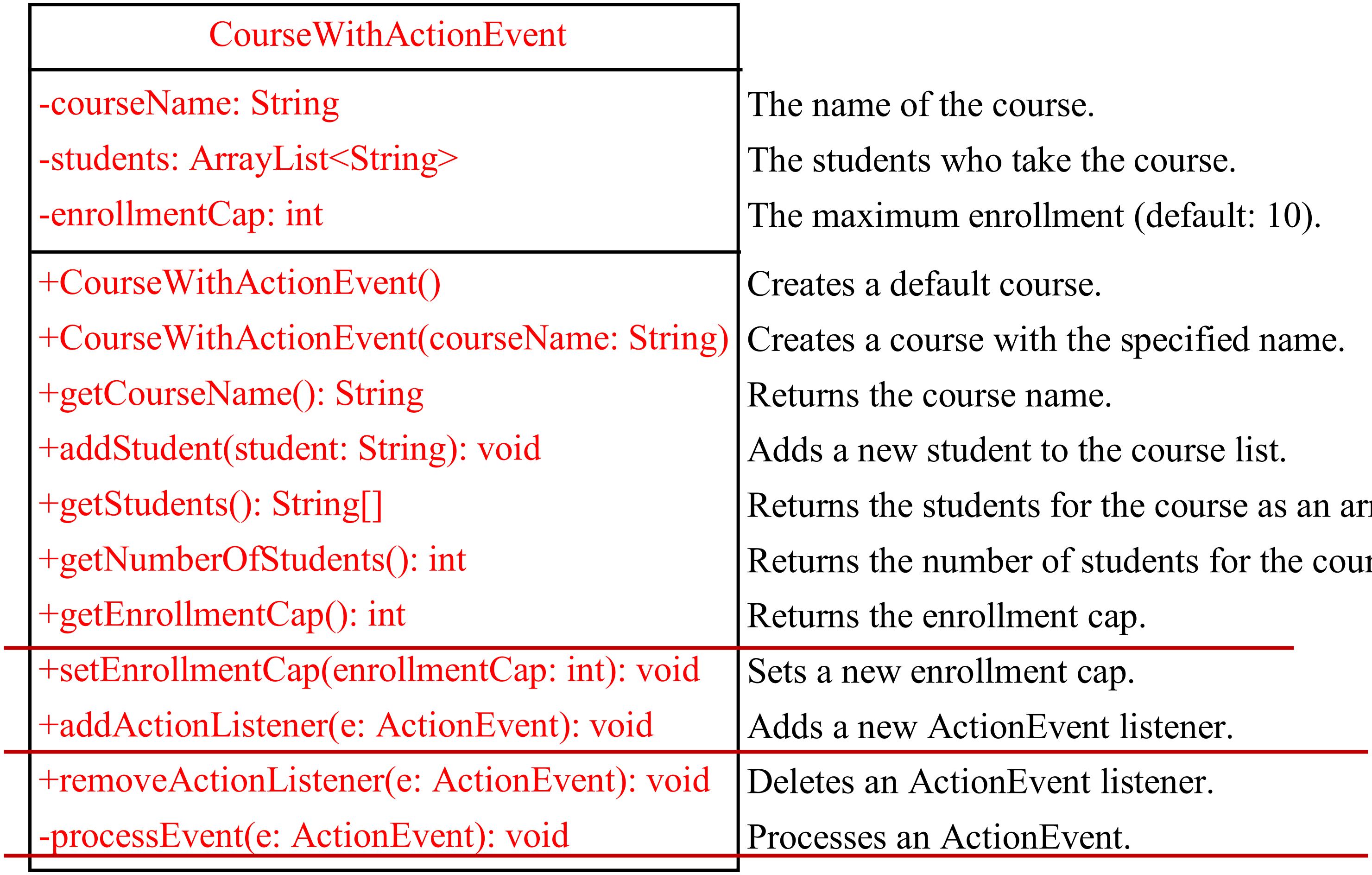
The naming pattern for removing a listener (either unicast or multicast) is:

```
public void remove<Event>Listener(<Event>Listener l)
```

A source component contains the code that creates an event object and passes it to the listening components by calling a method in the listener's event listener interface.

You may use a standard Java event class like ActionEvent to create event objects **or** may define your own event classes if necessary.

Example: Creating a Source Component

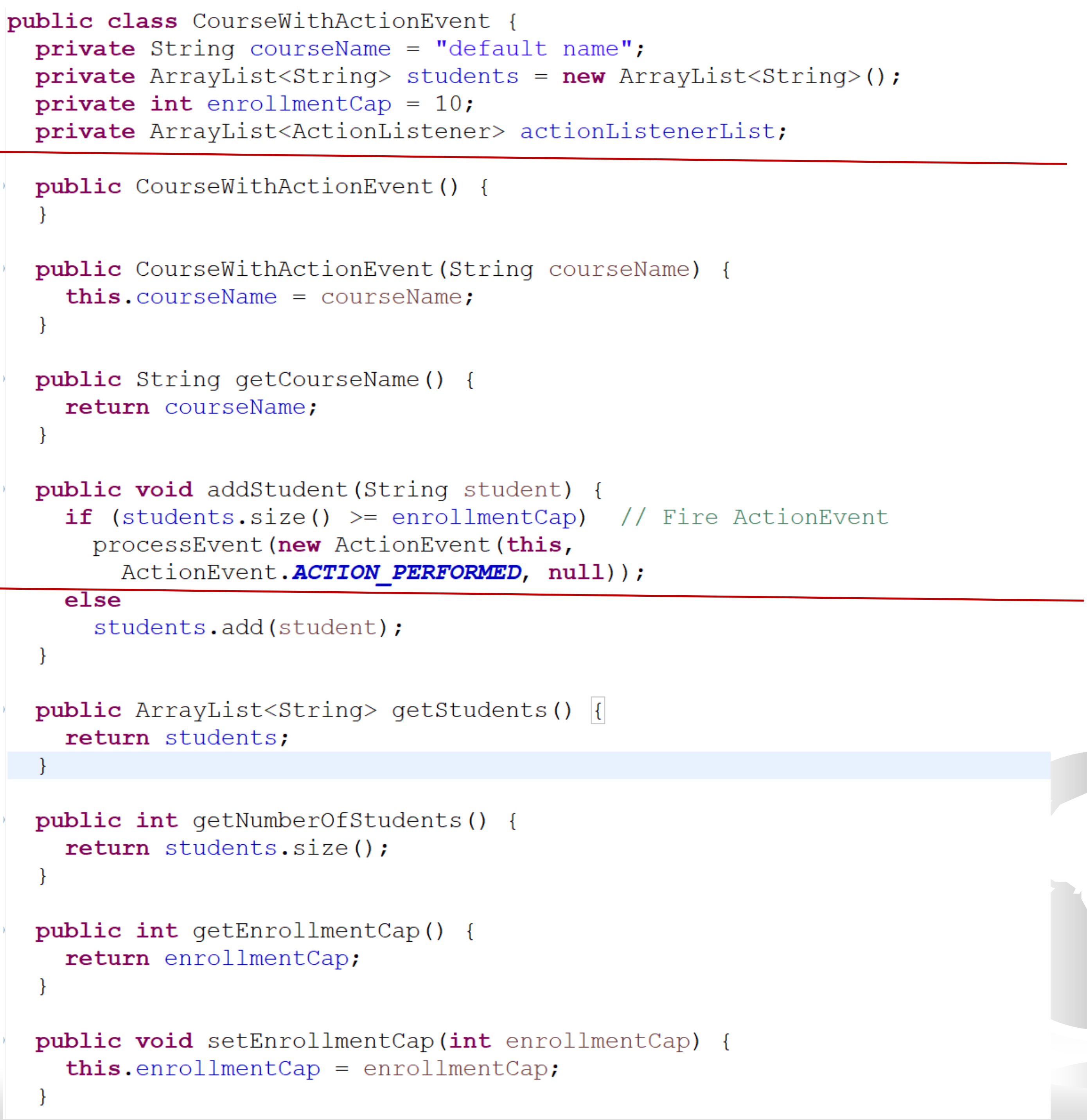


CourseWithActionEvent	
-courseName: String	The name of the course.
-students: ArrayList<String>	The students who take the course.
-enrollmentCap: int	The maximum enrollment (default: 10).
+CourseWithActionEvent()	Creates a default course.
+CourseWithActionEvent(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course list.
+getStudents(): String[]	Returns the students for the course as an array.
+getNumberOfStudents(): int	Returns the number of students for the course.
+getEnrollmentCap(): int	Returns the enrollment cap.
+setEnrollmentCap(enrollmentCap: int): void	Sets a new enrollment cap.
+addActionListener(e: ActionEvent): void	Adds a new ActionEvent listener.
+removeActionListener(e: ActionEvent): void	Deletes an ActionEvent listener.
-processEvent(e: ActionEvent): void	Processes an ActionEvent.

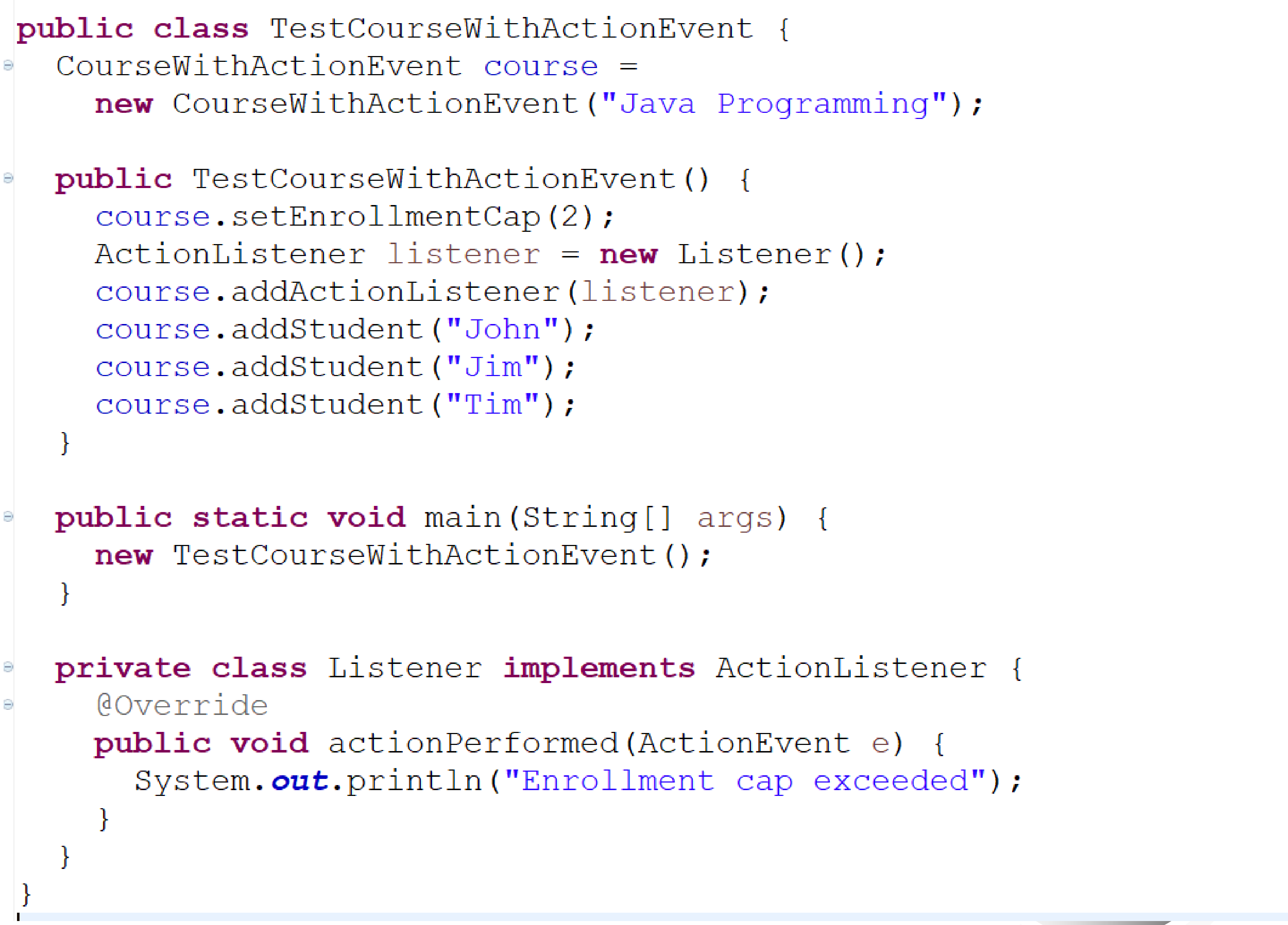
CourseWithActionEvent

TestCourseWithActionEvent

TestCourseWithActionEvent

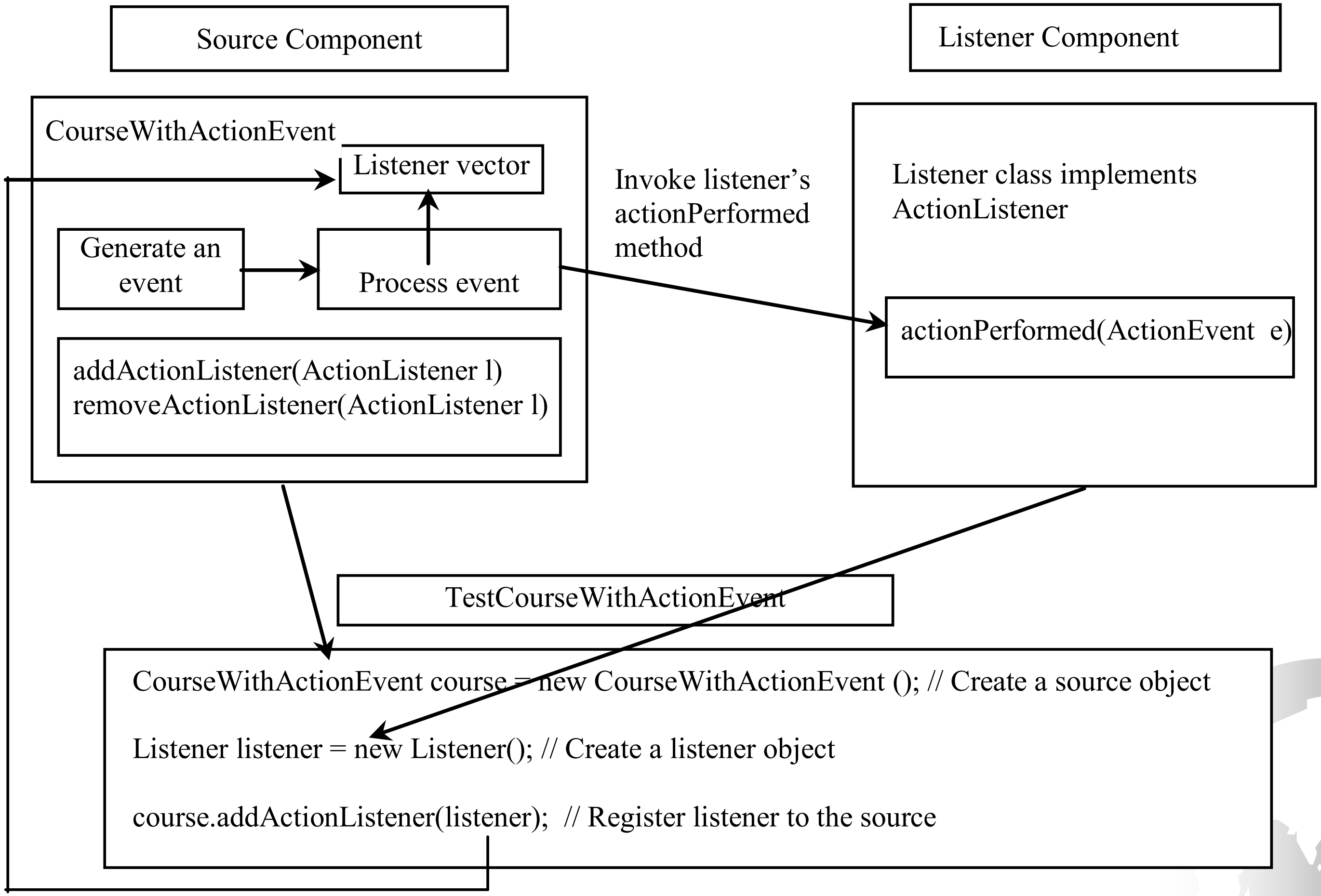


```java
public class CourseWithActionEvent {
  private String courseName = "default name";
  private ArrayList<String> students = new ArrayList<String>();
  private int enrollmentCap = 10;
  private ArrayList<ActionListener> actionListenerList;

  public CourseWithActionEvent() {
  }

  public CourseWithActionEvent(String courseName) {
    this.courseName = courseName;
  }

  public String getCourseName() {
    return courseName;
  }

  public void addStudent(String student) {
    if (students.size() >= enrollmentCap)  // Fire ActionEvent
      processEvent(new ActionEvent(this,
        ActionEvent.ACTION_PERFORMED, null));
    else
      students.add(student);
  }

  public ArrayList<String> getStudents() {
    return students;
  }

  public int getNumberOfStudents() {
    return students.size();
  }

  public int getEnrollmentCap() {
    return enrollmentCap;
  }

  public void setEnrollmentCap(int enrollmentCap) {
    this.enrollmentCap = enrollmentCap;
  }
  /** Register an action event listener */
  public synchronized void addActionListener
      (ActionListener listener) {
    if (actionListenerList == null) {
      actionListenerList = new ArrayList<ActionListener>(2);
    }

    if (!actionListenerList.contains(listener)) {
      actionListenerList.add(listener);
    }
  }

  /** Remove an action event listener */
  public synchronized void removeActionListener
      (ActionListener listener) {
    if (actionListenerList !=
        null && actionListenerList.contains(listener)) {
      actionListenerList.remove(listener);
    }
  }

  /** Fire ActionEvent */
  private void processEvent(ActionEvent e) {
    ArrayList<ActionListener> list;

    synchronized (this) {
      if (actionListenerList == null) return;
      list = (ArrayList<ActionListener>)actionListenerList.clone();
    }

    for (int i = 0; i < list.size(); i++) {
      ActionListener listener = (ActionListener)list.get(i);
      listener.actionPerformed(e);
    }
  }
}
```



```java
public class TestCourseWithActionEvent {
  CourseWithActionEvent course =
    new CourseWithActionEvent("Java Programming");

  public TestCourseWithActionEvent() {
    course.setEnrollmentCap(2);
    ActionListener listener = new Listener();
    course.addActionListener(listener);
    course.addStudent("John");
    course.addStudent("Jim");
    course.addStudent("Tim");
  }

  public static void main(String[] args) {
    new TestCourseWithActionEvent();
  }

  private class Listener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
      System.out.println("Enrollment cap exceeded");
    }
  }
}
```

Interaction Between Source and Listener Components



Source Component

CourseWithActionEvent

Listener vector

Generate an event

Process event

addActionListener(ActionListener l)
removeActionListener(ActionListener l)

Invoke listener's actionPerformed method

Listener Component

Listener class implements ActionListener

actionPerformed(ActionEvent e)

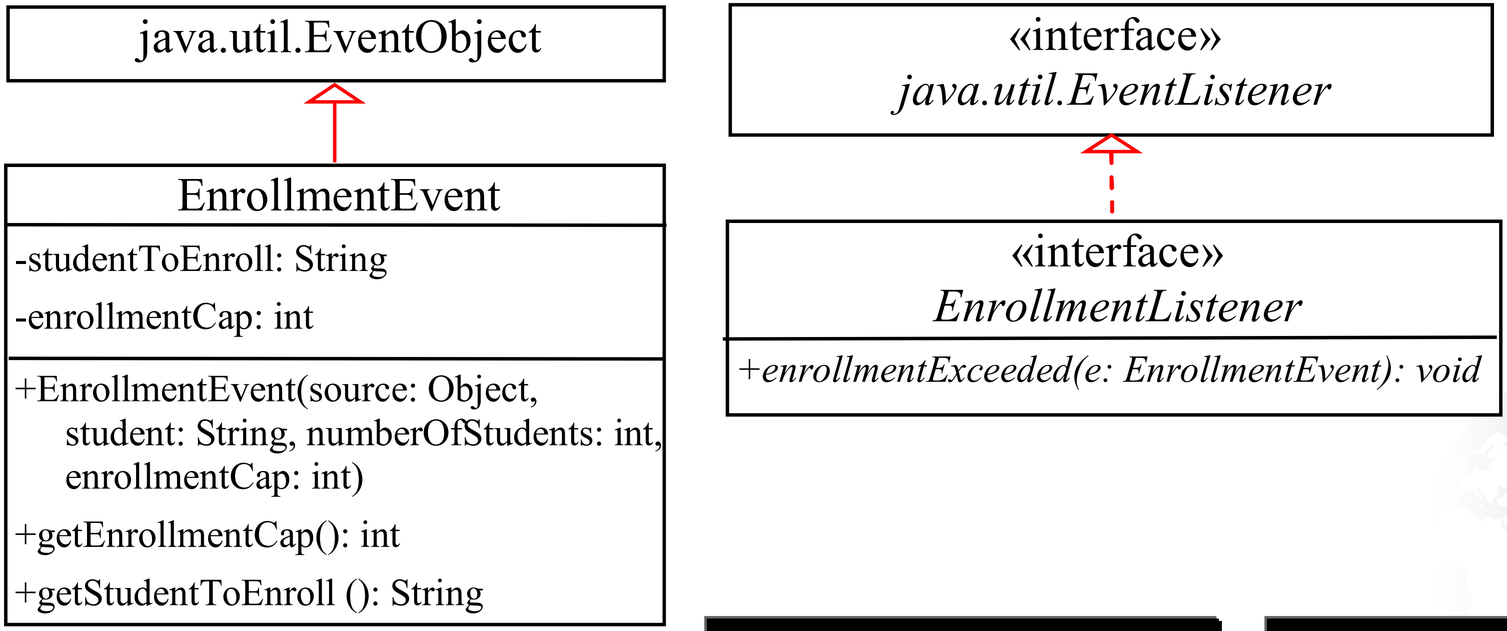
TestCourseWithActionEvent

```
CourseWithActionEvent course = new CourseWithActionEvent (); // Create a source object

Listener listener = new Listener(); // Create a listener object

course.addActionListener(listener);  // Register listener to the source
```

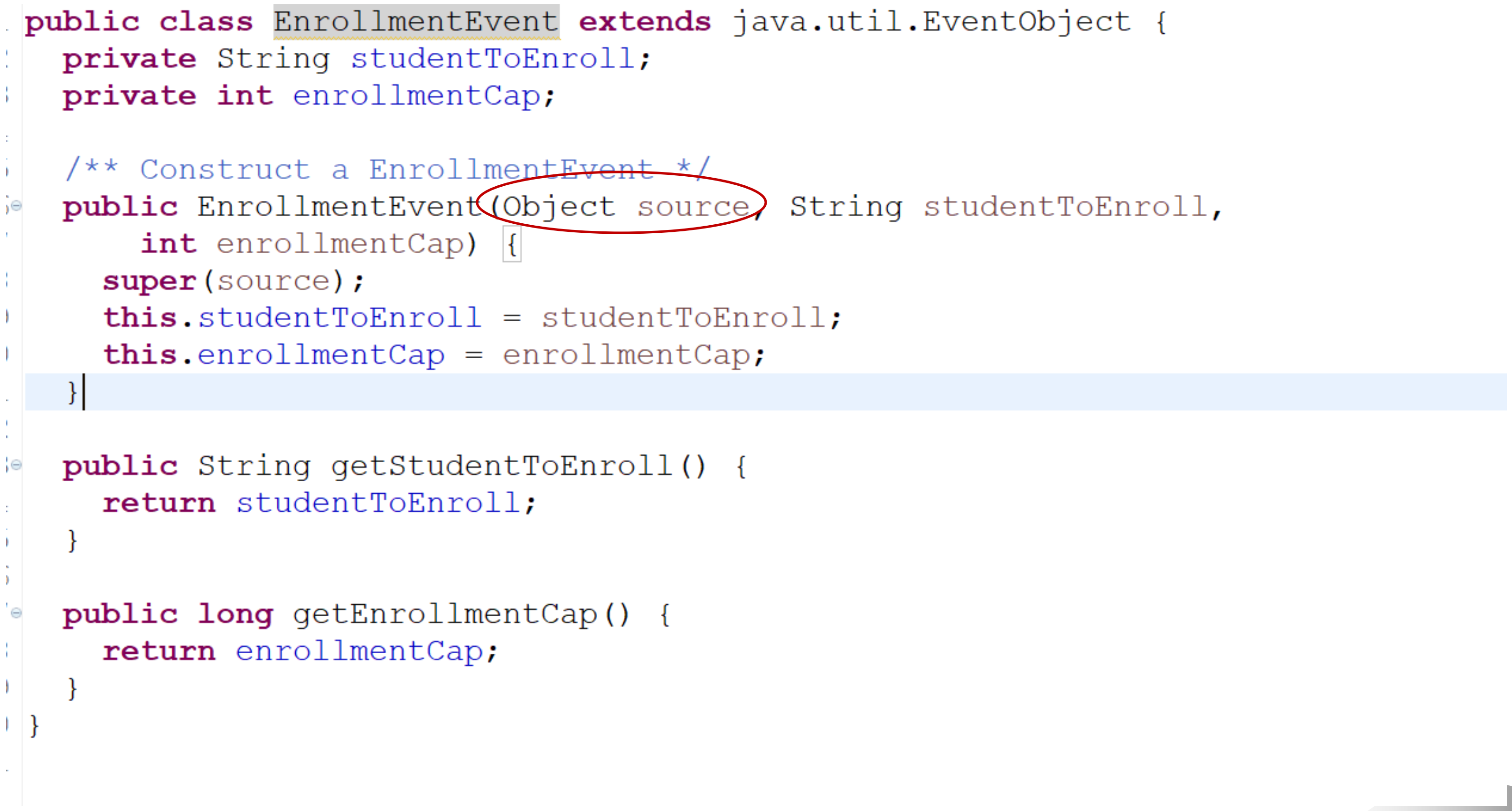
The listener is registered with the source course, and the source invokes the listener's handler actionPerformed to process the event.

Example: Creating a Custom Event Set

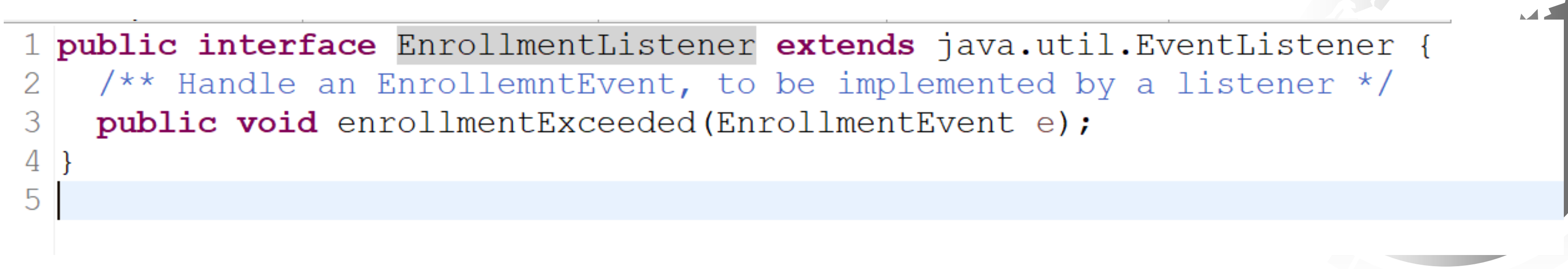
Problem: This example creates a **custom** event named EnrollmentEvent for describing enrollment events, and its corresponding listener interface EnrollmentListener for defining an enrollmentExceeded handler.


java.util.EventObject
EnrollmentEvent
-studentToEnroll: String
-enrollmentCap: int
+EnrollmentEvent(source: Object, student: String, numberOfStudents: int, enrollmentCap: int)
+getEnrollmentCap(): int
+getStudentToEnroll (): String
«interface»
java.util.EventListener
«interface»
EnrollmentListener
+enrollmentExceeded(e: EnrollmentEvent): void

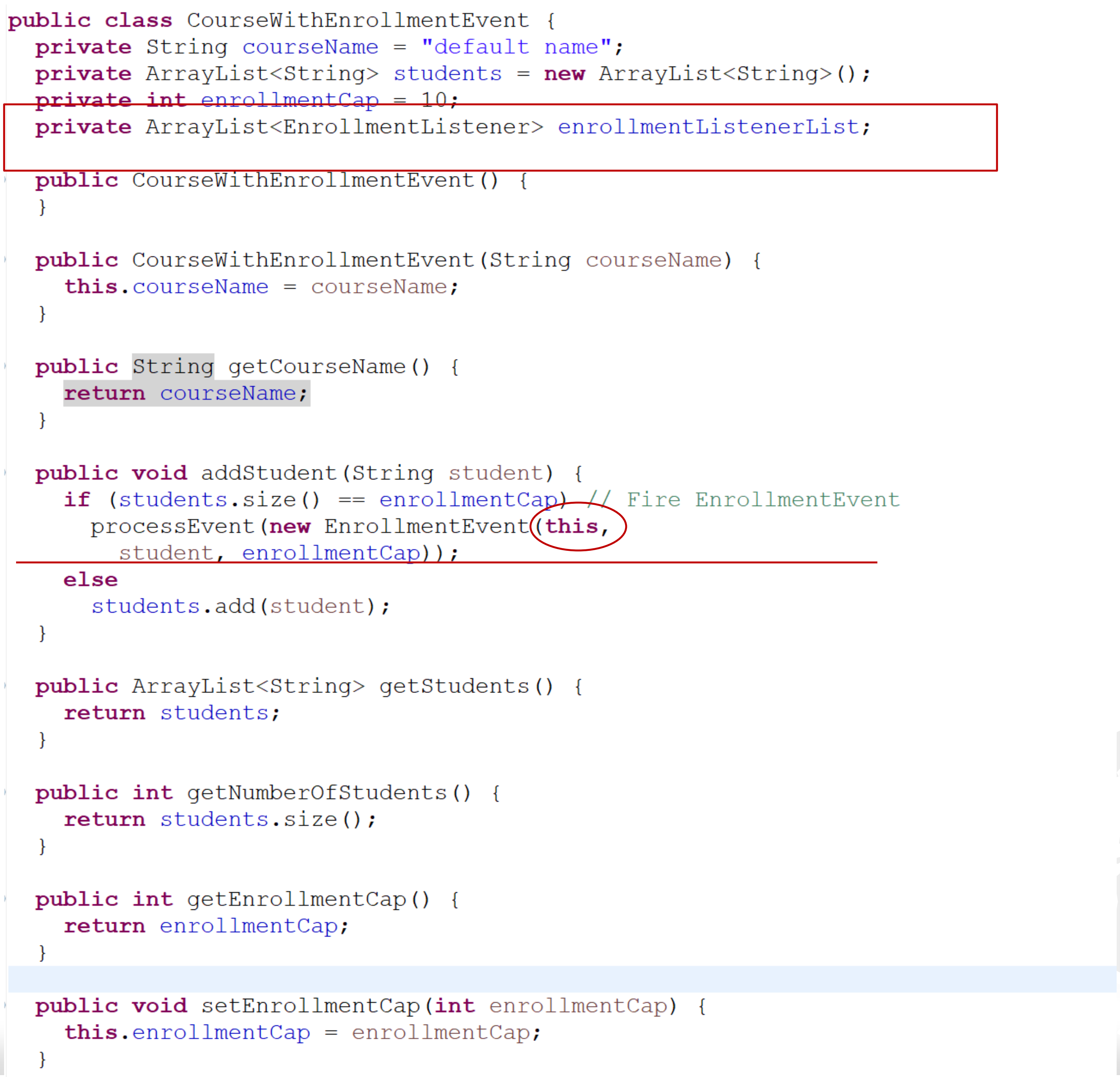
EnrollmentEvent EnrollmentListener



```java
public class EnrollmentEvent extends java.util.EventObject {
  private String studentToEnroll;
  private int enrollmentCap;

  /** Construct a EnrollmentEvent */
  public EnrollmentEvent(Object source, String studentToEnroll,
      int enrollmentCap) {
    super(source);
    this.studentToEnroll = studentToEnroll;
    this.enrollmentCap = enrollmentCap;
  }

  public String getStudentToEnroll() {
    return studentToEnroll;
  }

  public long getEnrollmentCap() {
    return enrollmentCap;
  }
}
```

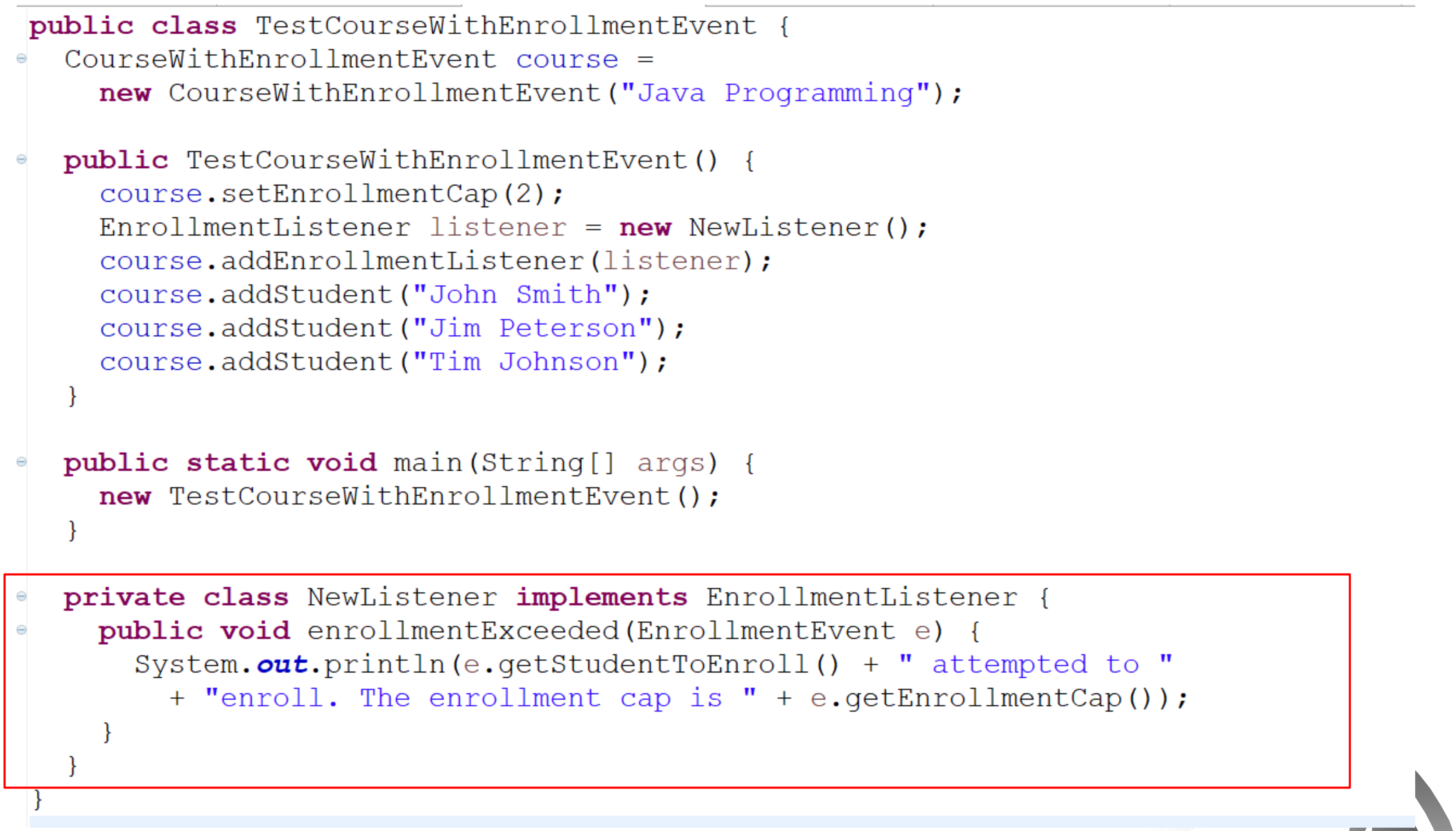


```java
public interface EnrollmentListener extends java.util.EventListener {
  /** Handle an EnrollemntEvent, to be implemented by a listener */
  public void enrollmentExceeded(EnrollmentEvent e);
}

```



```java
public class CourseWithEnrollmentEvent {
  private String courseName = "default name";
  private ArrayList<String> students = new ArrayList<String>();
  private int enrollmentCap = 10;
  private ArrayList<EnrollmentListener> enrollmentListenerList;

  public CourseWithEnrollmentEvent() {
  }

  public CourseWithEnrollmentEvent(String courseName) {
    this.courseName = courseName;
  }

  public String getCourseName() {
    return courseName;
  }

  public void addStudent(String student) {
    if (students.size() == enrollmentCap) // Fire EnrollmentEvent
      processEvent(new EnrollmentEvent(this,
        student, enrollmentCap));
    else
      students.add(student);
  }

  public ArrayList<String> getStudents() {
    return students;
  }

  public int getNumberOfStudents() {
    return students.size();
  }

  public int getEnrollmentCap() {
    return enrollmentCap;
  }

  public void setEnrollmentCap(int enrollmentCap) {
    this.enrollmentCap = enrollmentCap;
  }
```

```java
/** Register an action event listener */
public synchronized void addEnrollmentListener
    (EnrollmentListener listener) {
  if (enrollmentListenerList == null) {
    enrollmentListenerList = new ArrayList<EnrollmentListener>(2);
  }

  if (!enrollmentListenerList.contains(listener)) {
    enrollmentListenerList.add(listener);
  }
}

/** Remove an action event listener */
public synchronized void removeEnrollmentListener
    (EnrollmentListener listener) {
  if (enrollmentListenerList !=
      null && enrollmentListenerList.contains(listener)) {
    enrollmentListenerList.remove(listener);
  }
}

/** Fire EnrollmentEvent */
private void processEvent(EnrollmentEvent e) {
  ArrayList<EnrollmentListener> list;

  synchronized (this) {
    if (enrollmentListenerList == null) return;
    list = (ArrayList<EnrollmentListener>)
      enrollmentListenerList.clone();
  }

  for (int i = 0; i < list.size(); i++) {
    EnrollmentListener listener = (EnrollmentListener)list.get(i);
    listener.enrollmentExceeded(e);
  }
}
```



```
public class TestCourseWithEnrollmentEvent {
  CourseWithEnrollmentEvent course =
    new CourseWithEnrollmentEvent("Java Programming");

  public TestCourseWithEnrollmentEvent() {
    course.setEnrollmentCap(2);
    EnrollmentListener listener = new NewListener();
    course.addEnrollmentListener(listener);
    course.addStudent("John Smith");
    course.addStudent("Jim Peterson");
    course.addStudent("Tim Johnson");
  }

  public static void main(String[] args) {
    new TestCourseWithEnrollmentEvent();
  }

  private class NewListener implements EnrollmentListener {
    public void enrollmentExceeded(EnrollmentEvent e) {
      System.out.println(e.getStudentToEnroll() + " attempted to "
        + "enroll. The enrollment cap is " + e.getEnrollmentCap());
    }
  }
}
```

Creating Custom Event Pairs

You have already used event sets (e.g., ActionEvent/ActionListener) and event source components (JButton) in Java GUI programming. You can create your own event sets and source components.

A custom event class must extend java.util.EventObject or a subclass of java.util.EventObject. Additionally, it may provide constructors to create events, data members and methods to describe the event.

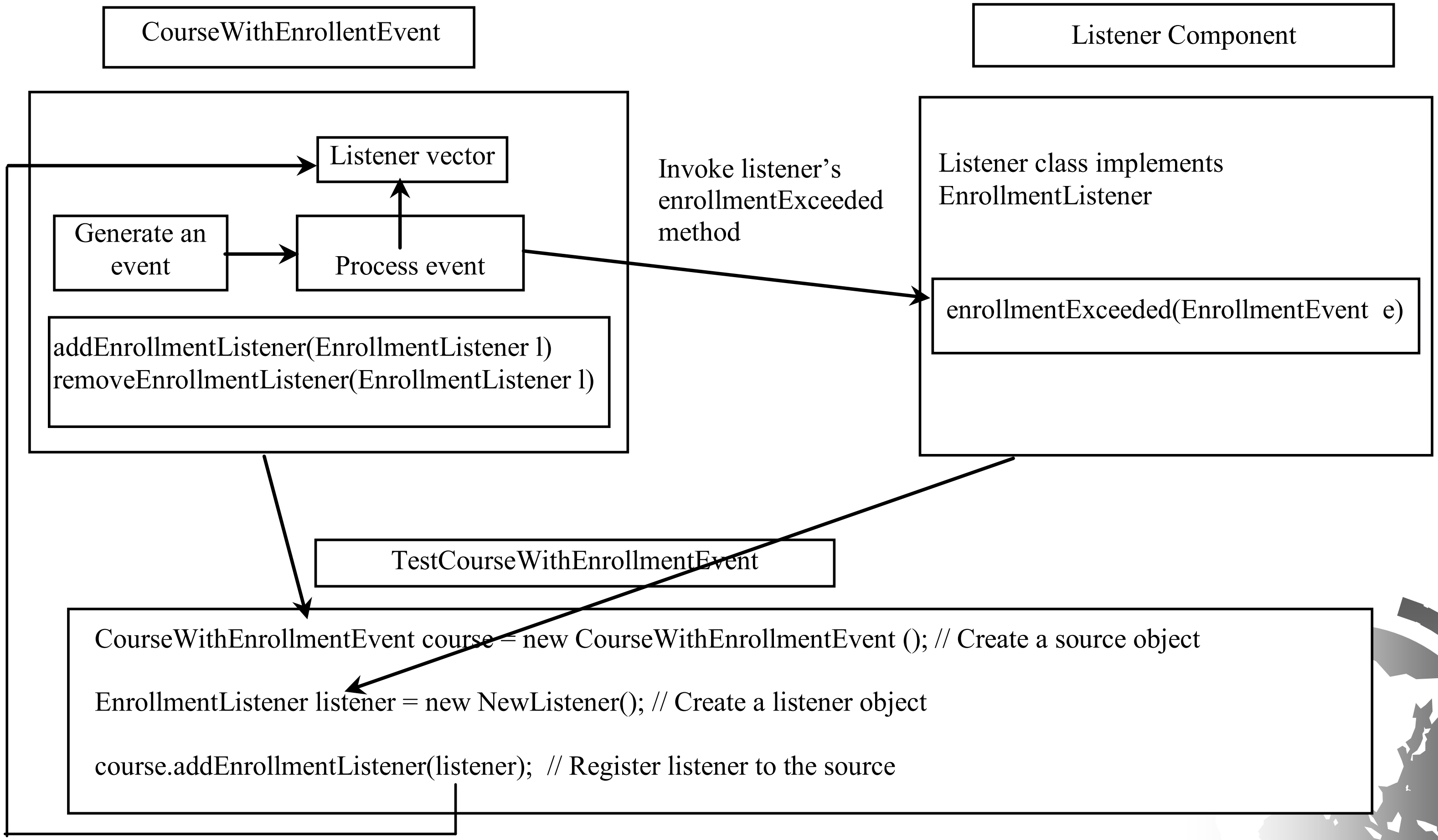
A custom event listener interface must extend java.util.EventListener or a subinterface of java.util.EventListener, and define the signature of the handlers for the event. By convention, the listener interface should be named <*Event*>Listener for the corresponding event class named <*Event*>.

Example: Source Component Using Custom Event Pairs

CourseWithEnrollmentEvent

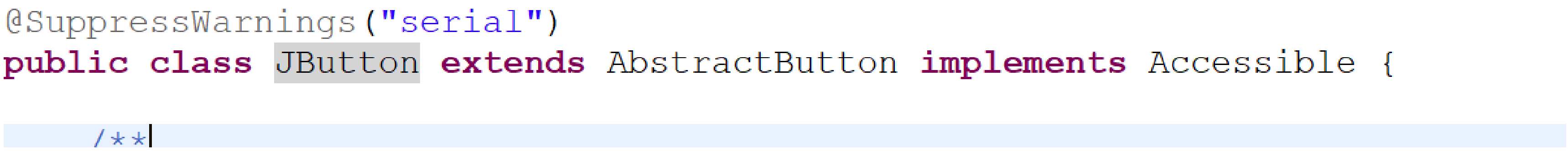
TestCourseWithEnrollmentEvent

Interaction Between Source and Listener Components

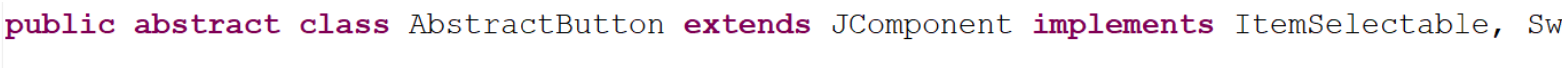

CourseWithEnrollentEvent
Listener vector
Generate an event
Process event
addEnrollmentListener(EnrollmentListener l)
removeEnrollmentListener(EnrollmentListener l)
Invoke listener's enrollmentExceeded method
Listener Component
Listener class implements EnrollmentListener
enrollmentExceeded(EnrollmentEvent e)
TestCourseWithEnrollmentEvent

```
CourseWithEnrollmentEvent course = new CourseWithEnrollmentEvent (); // Create a source object

EnrollmentListener listener = new NewListener(); // Create a listener object

course.addEnrollmentListener(listener);  // Register listener to the source
```

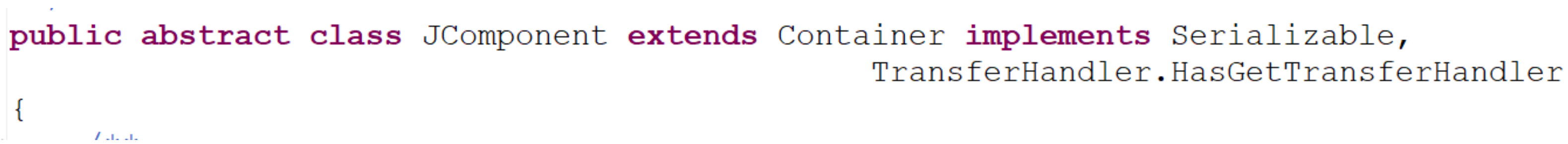
The listener is registered with the source course, and the source invokes the listener's handler enrollmentExceeded to process the event.



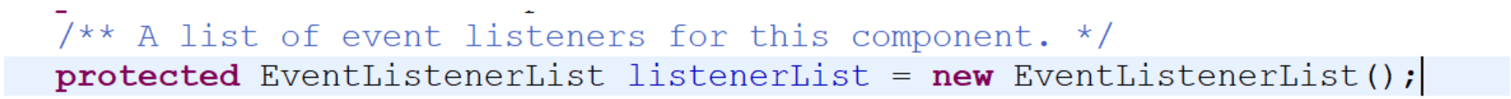
```
@SuppressWarnings("serial")
public class JButton extends AbstractButton implements Accessible {

    /**
```



```
public abstract class AbstractButton extends JComponent implements ItemSelectable, Sw
```

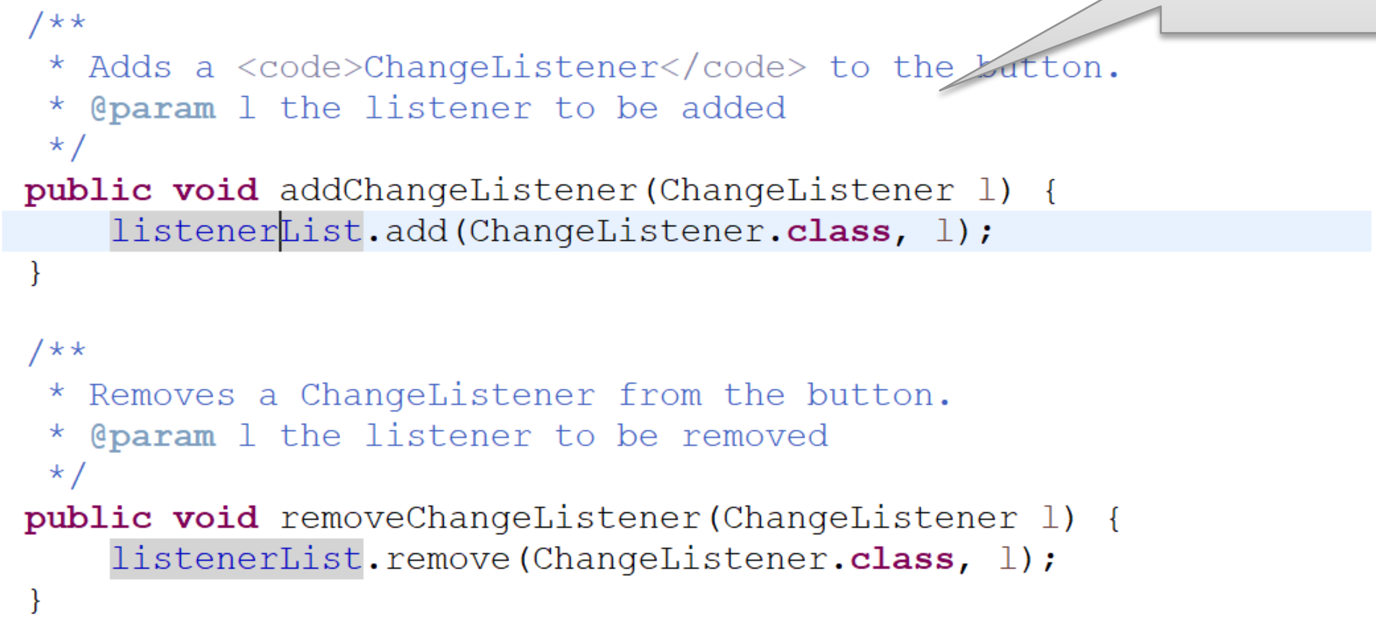


```
public abstract class JComponent extends Container implements Serializable,
                                              TransferHandler.HasGetTransferHandler
{
    /**
```

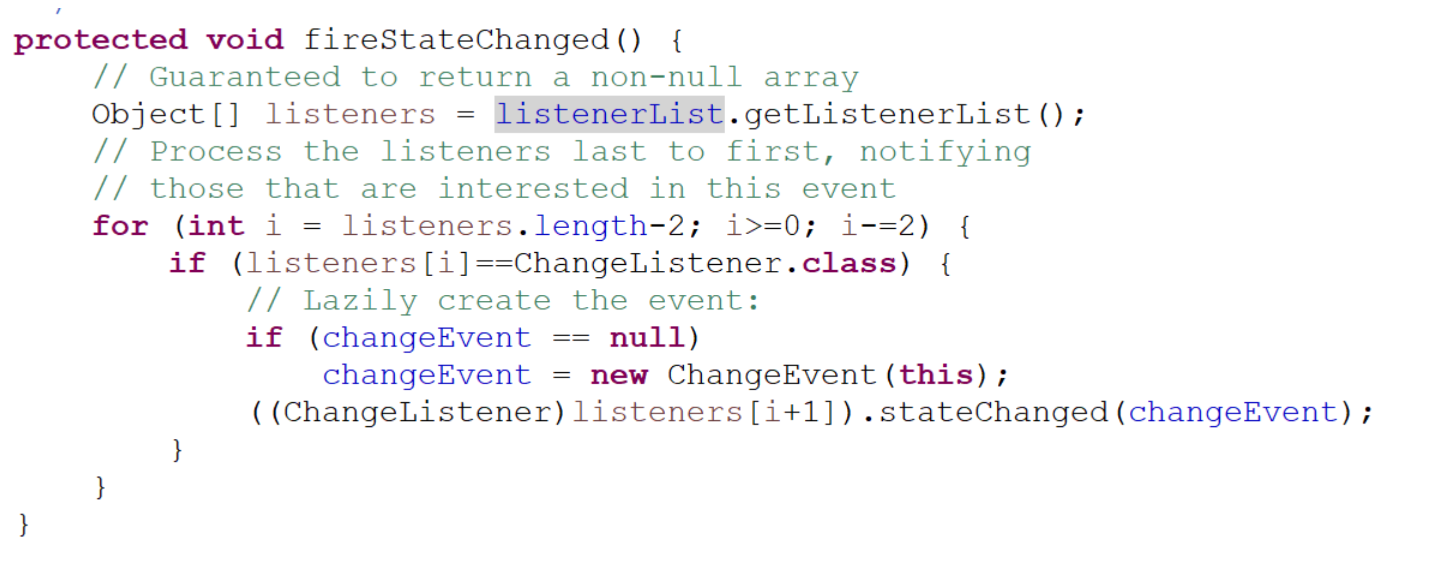


```
    /** A list of event listeners for this component. */
    protected EventListenerList listenerList = new EventListenerList();
```

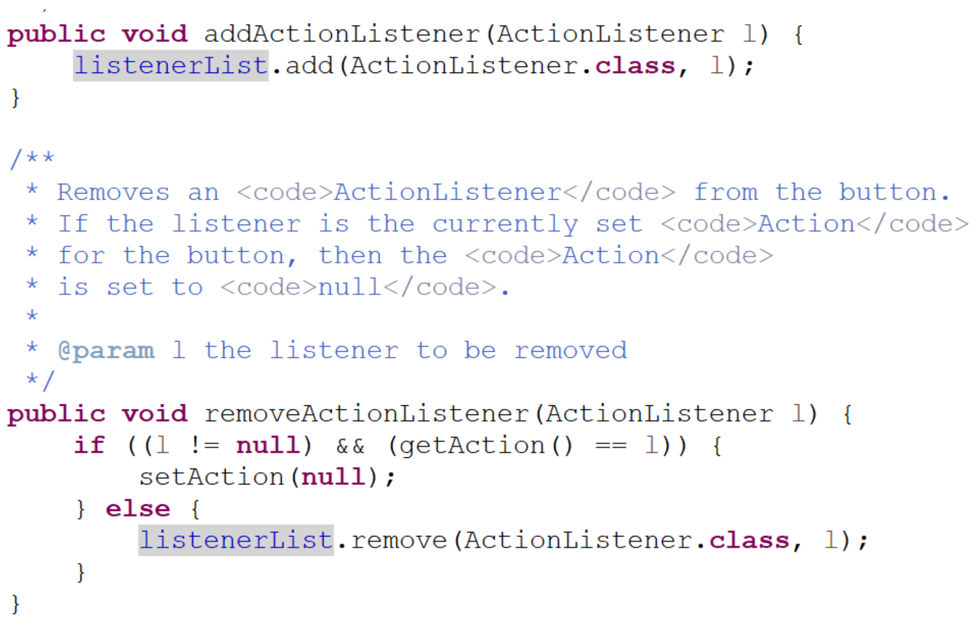
AbstractButton



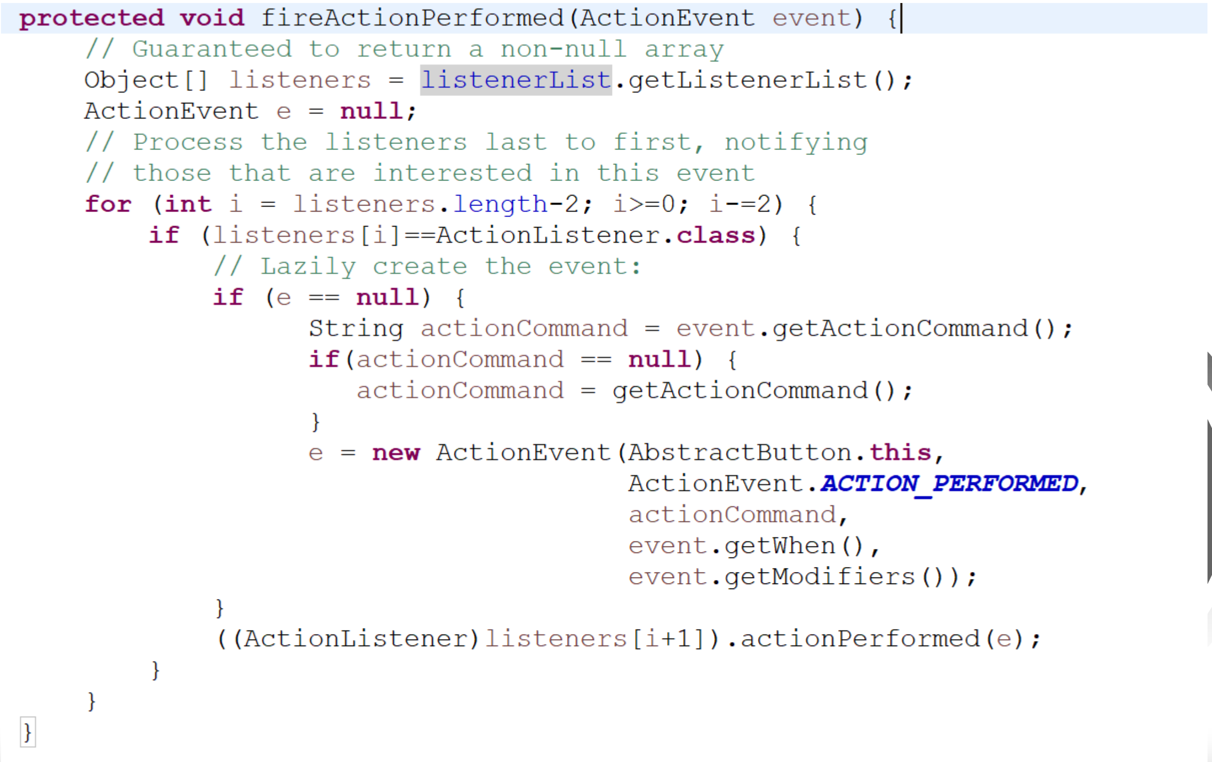
```
/**
 * Adds a <code>ChangeListener</code> to the button.
 * @param l the listener to be added
 */
public void addChangeListener(ChangeListener l) {
    listenerList.add(ChangeListener.class, l);
}

/**
 * Removes a ChangeListener from the button.
 * @param l the listener to be removed
 */
public void removeChangeListener(ChangeListener l) {
    listenerList.remove(ChangeListener.class, l);
}
```



```
protected void fireStateChanged() {
    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length-2; i>=0; i-=2) {
        if (listeners[i]==ChangeListener.class) {
            // Lazily create the event:
            if (changeEvent == null)
                changeEvent = new ChangeEvent(this);
            ((ChangeListener)listeners[i+1]).stateChanged(changeEvent);
        }
    }
}
```

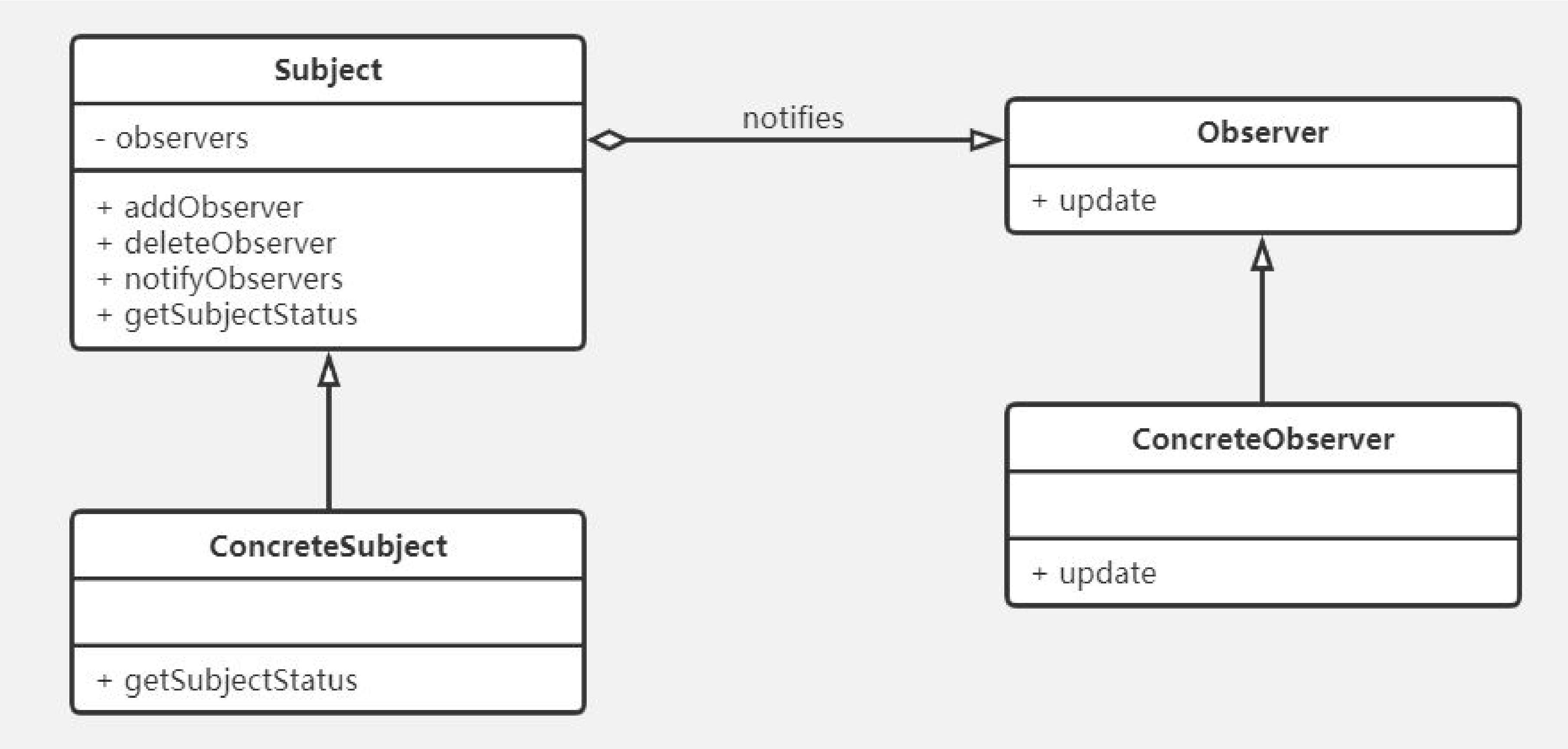


```java
public void addActionListener(ActionListener l) {
    listenerList.add(ActionListener.class, l);
}

/**
 * Removes an <code>ActionListener</code> from the button.
 * If the listener is the currently set <code>Action</code>
 * for the button, then the <code>Action</code>
 * is set to <code>null</code>.
 *
 * @param l the listener to be removed
 */
public void removeActionListener(ActionListener l) {
    if ((l != null) && (getAction() == l)) {
        setAction(null);
    } else {
        listenerList.remove(ActionListener.class, l);
    }
}
```

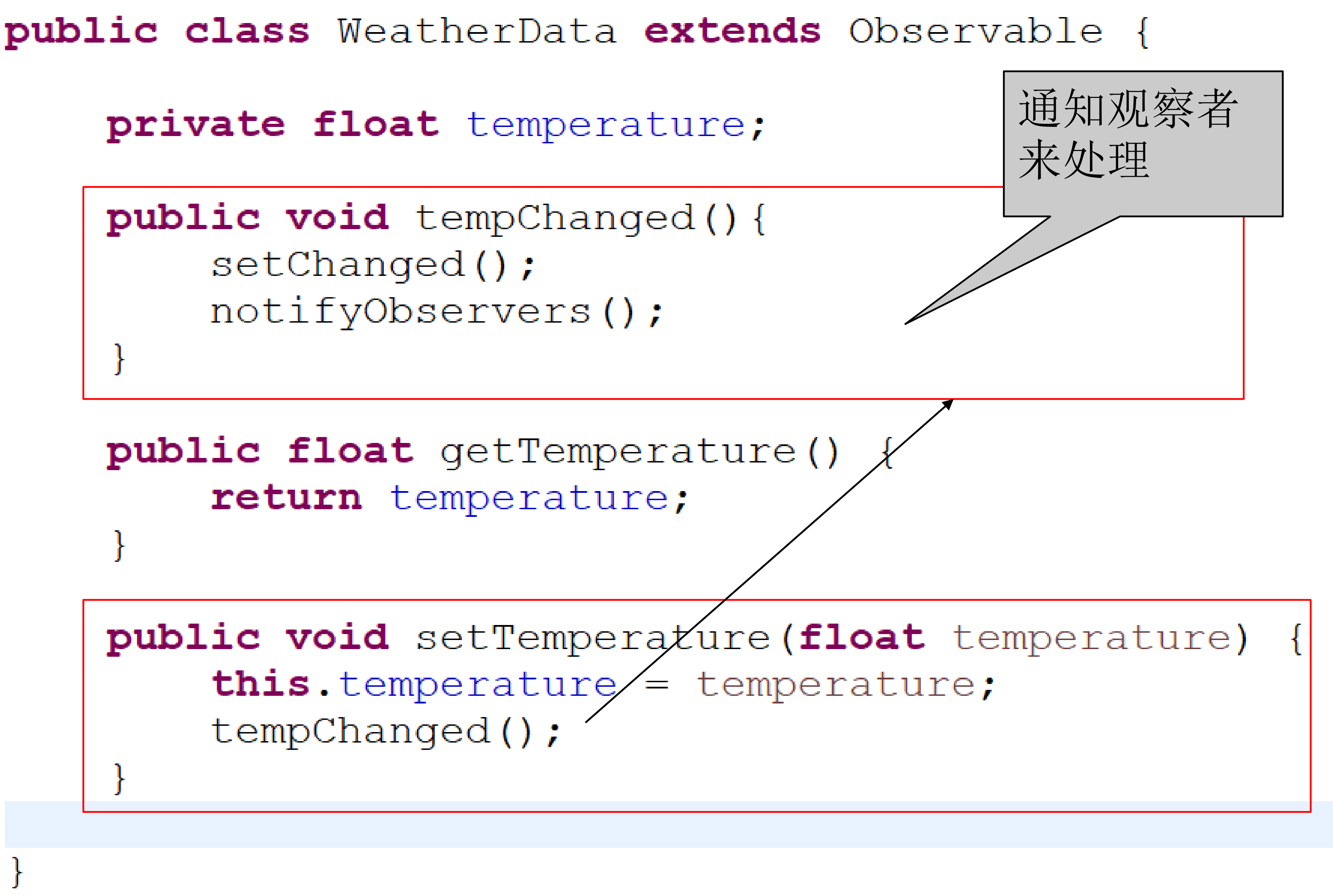


```java
protected void fireActionPerformed(ActionEvent event) {
    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();
    ActionEvent e = null;
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length-2; i>=0; i-=2) {
        if (listeners[i]==ActionListener.class) {
            // Lazily create the event:
            if (e == null) {
                  String actionCommand = event.getActionCommand();
                  if(actionCommand == null) {
                     actionCommand = getActionCommand();
                  }
                  e = new ActionEvent(AbstractButton.this,
                                      ActionEvent.ACTION_PERFORMED,
                                      actionCommand,
                                      event.getWhen(),
                                      event.getModifiers());
            }
            ((ActionListener)listeners[i+1]).actionPerformed(e);
        }
    }
}
```

使用Observable实现观察者模式

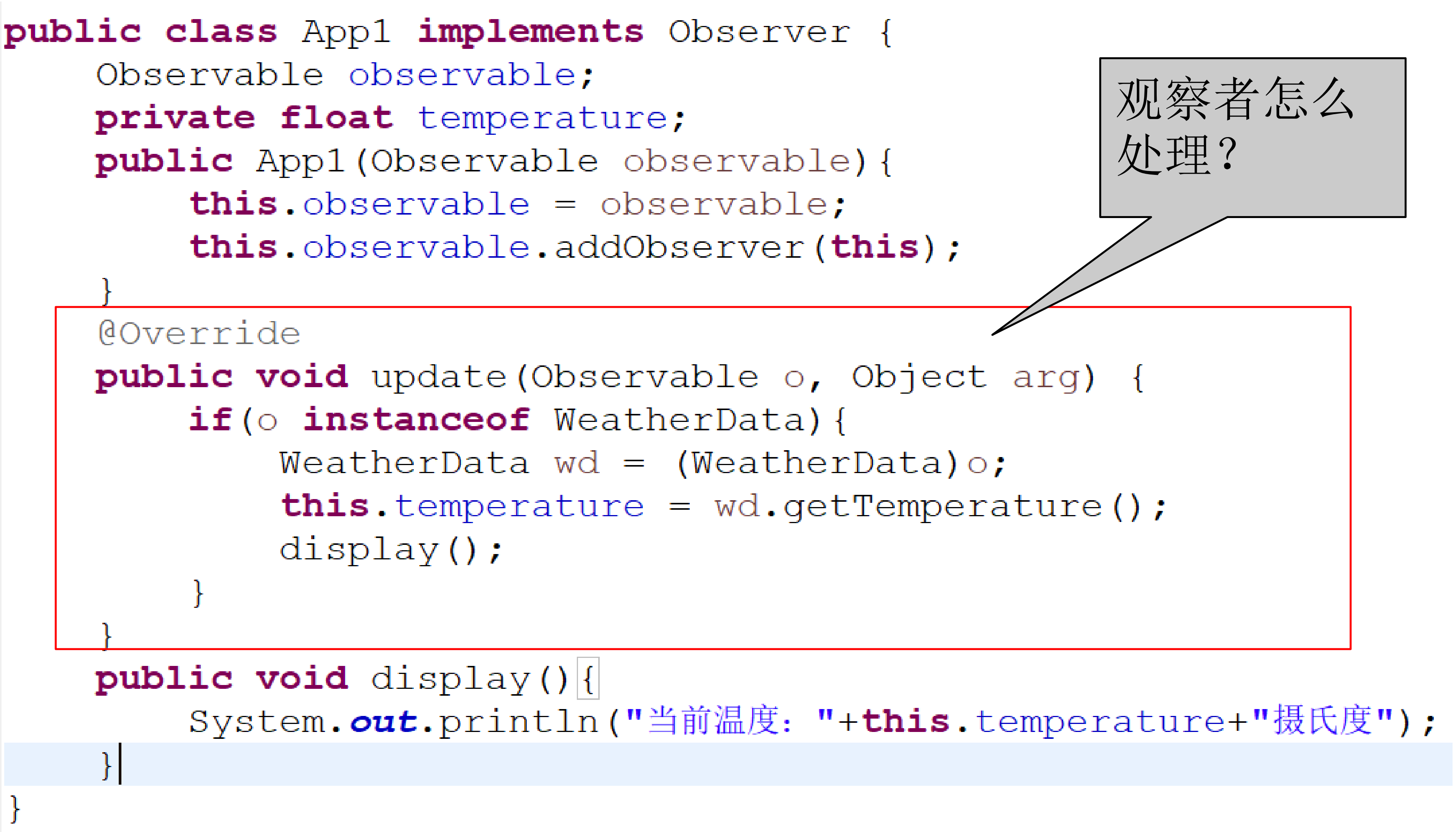


- Subject状态发生变化，委托给观察者处理



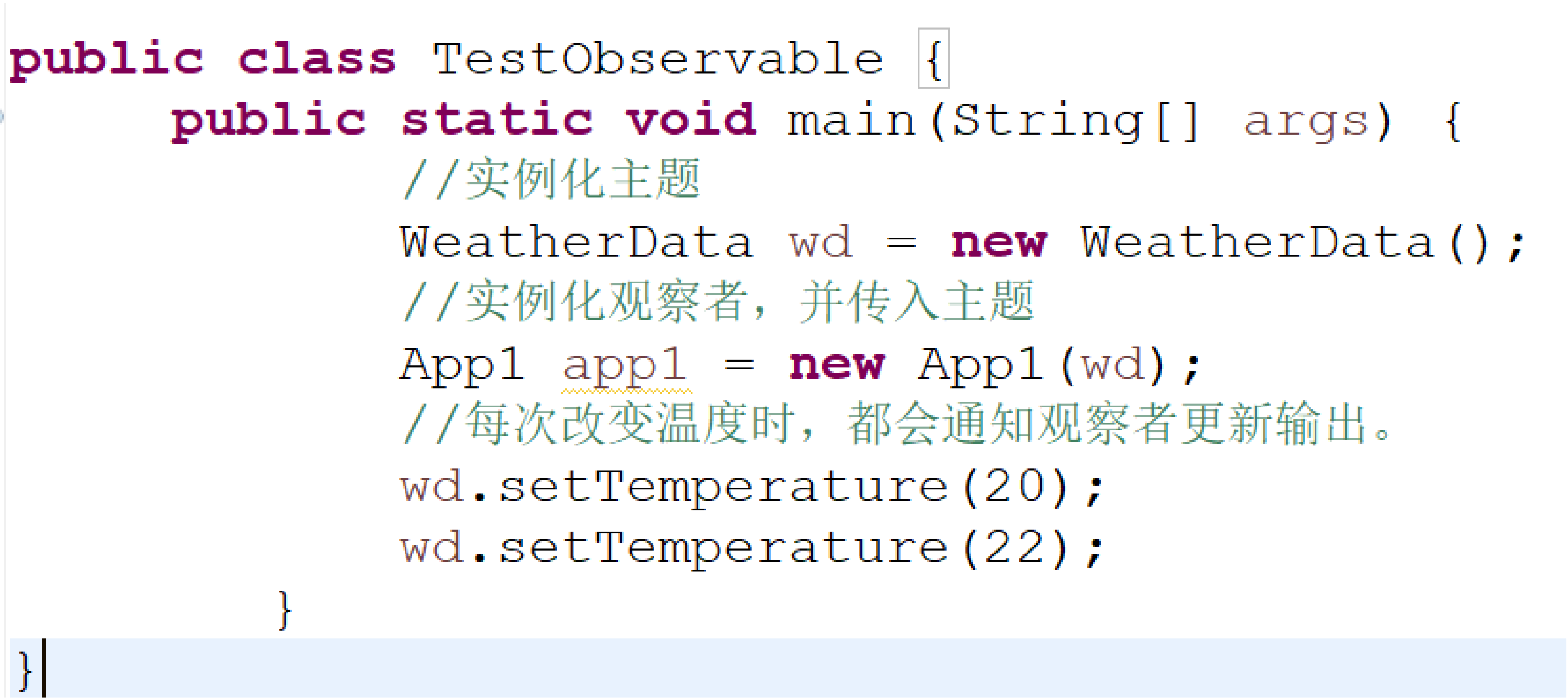
```java
public class WeatherData extends Observable {

    private float temperature;

    public void tempChanged(){
        setChanged();
        notifyObservers();
    }

    public float getTemperature() {
        return temperature;
    }

    public void setTemperature(float temperature) {
        this.temperature = temperature;
        tempChanged();
    }

}
```

通知观察者来处理

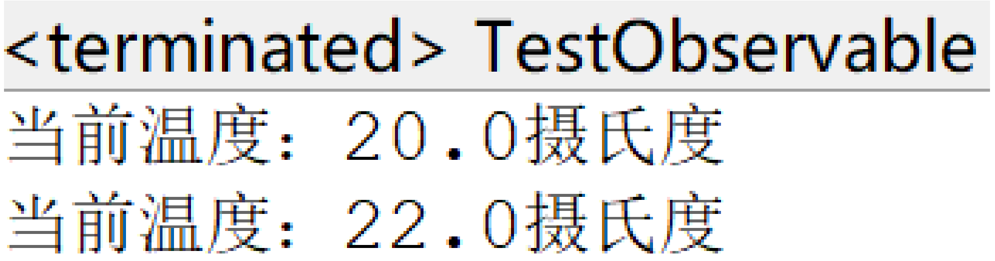


```java
public class App1 implements Observer {
    Observable observable;
    private float temperature;
    public App1(Observable observable){
        this.observable = observable;
        this.observable.addObserver(this);
    }
    @Override
    public void update(Observable o, Object arg) {
        if(o instanceof WeatherData){
            WeatherData wd = (WeatherData)o;
            this.temperature = wd.getTemperature();
            display();
        }
    }
    public void display(){
        System.out.println("当前温度: "+this.temperature+"摄氏度");
    }
}
```

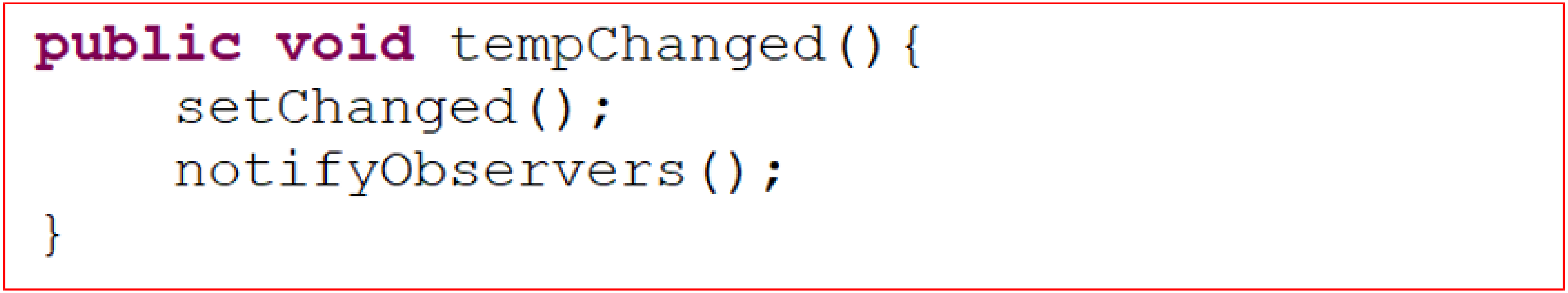
观察者怎么处理？



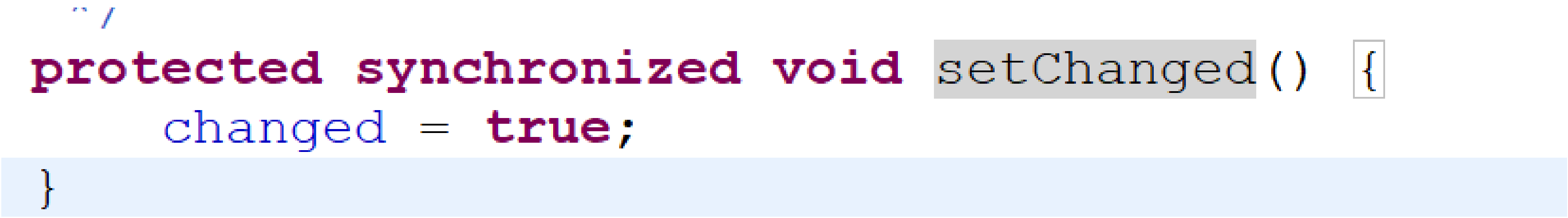
```java
public class TestObservable {
    public static void main(String[] args) {
            //实例化主题
            WeatherData wd = new WeatherData();
            //实例化观察者，并传入主题
            App1 app1 = new App1(wd);
            //每次改变温度时，都会通知观察者更新输出。
            wd.setTemperature(20);
            wd.setTemperature(22);
        }
}
```



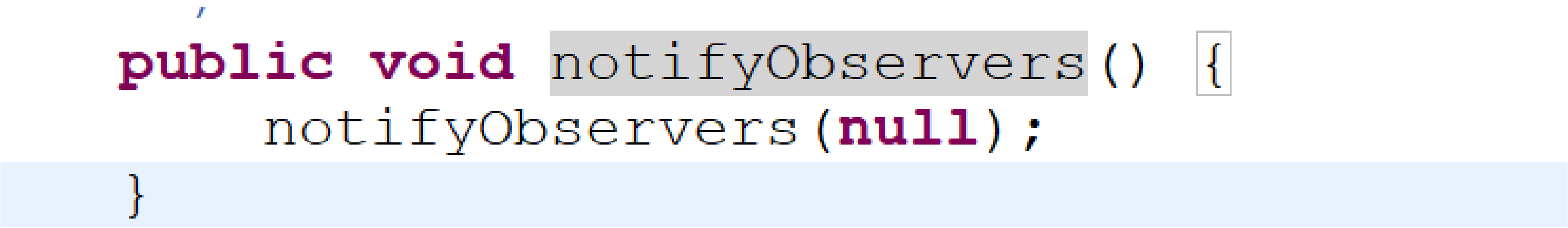
```
<terminated> TestObservable
当前温度：20.0摄氏度
当前温度：22.0摄氏度
```



```
public void tempChanged(){
    setChanged();
    notifyObservers();
}
```

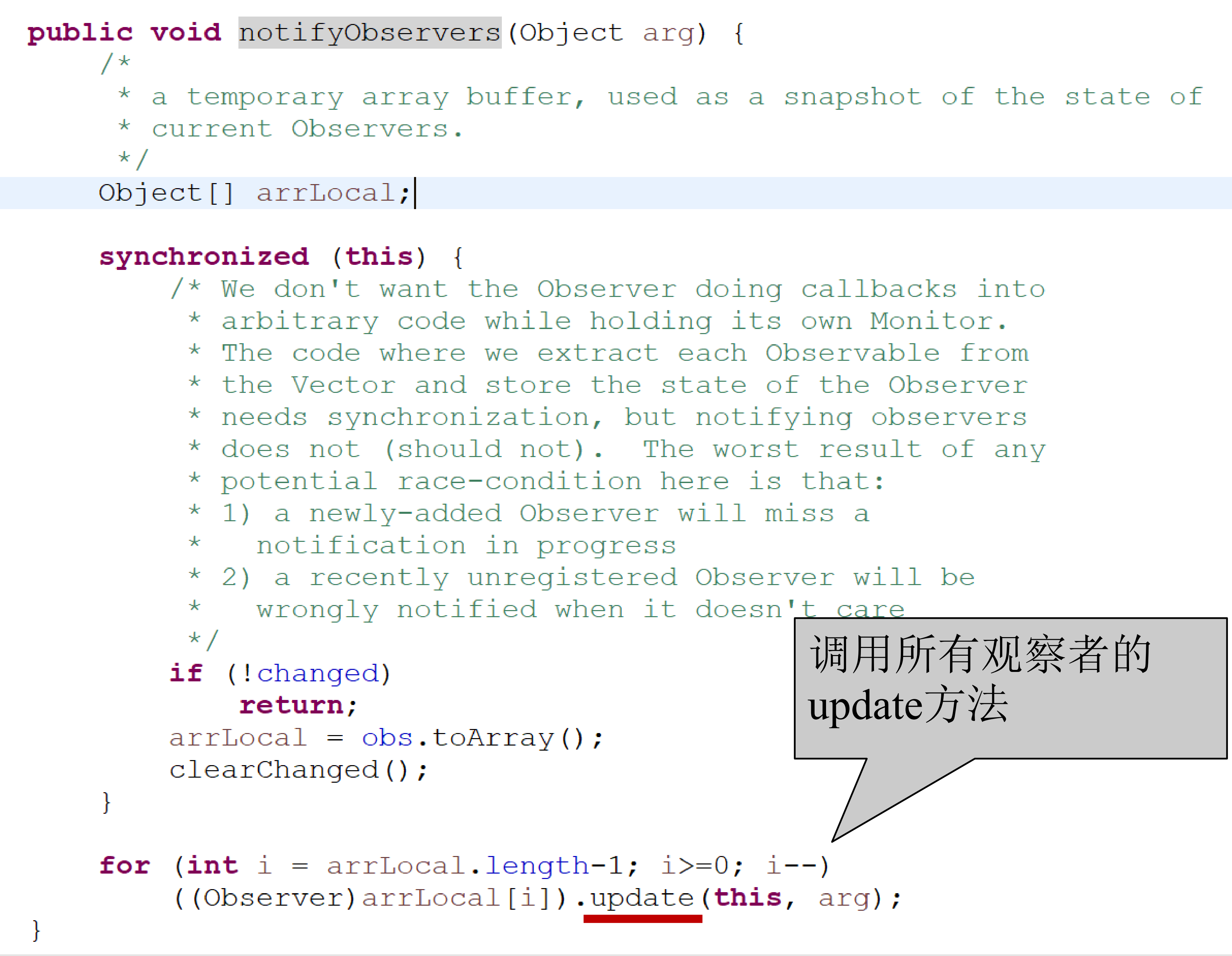


```
 */
protected synchronized void setChanged() {
    changed = true;
}
```



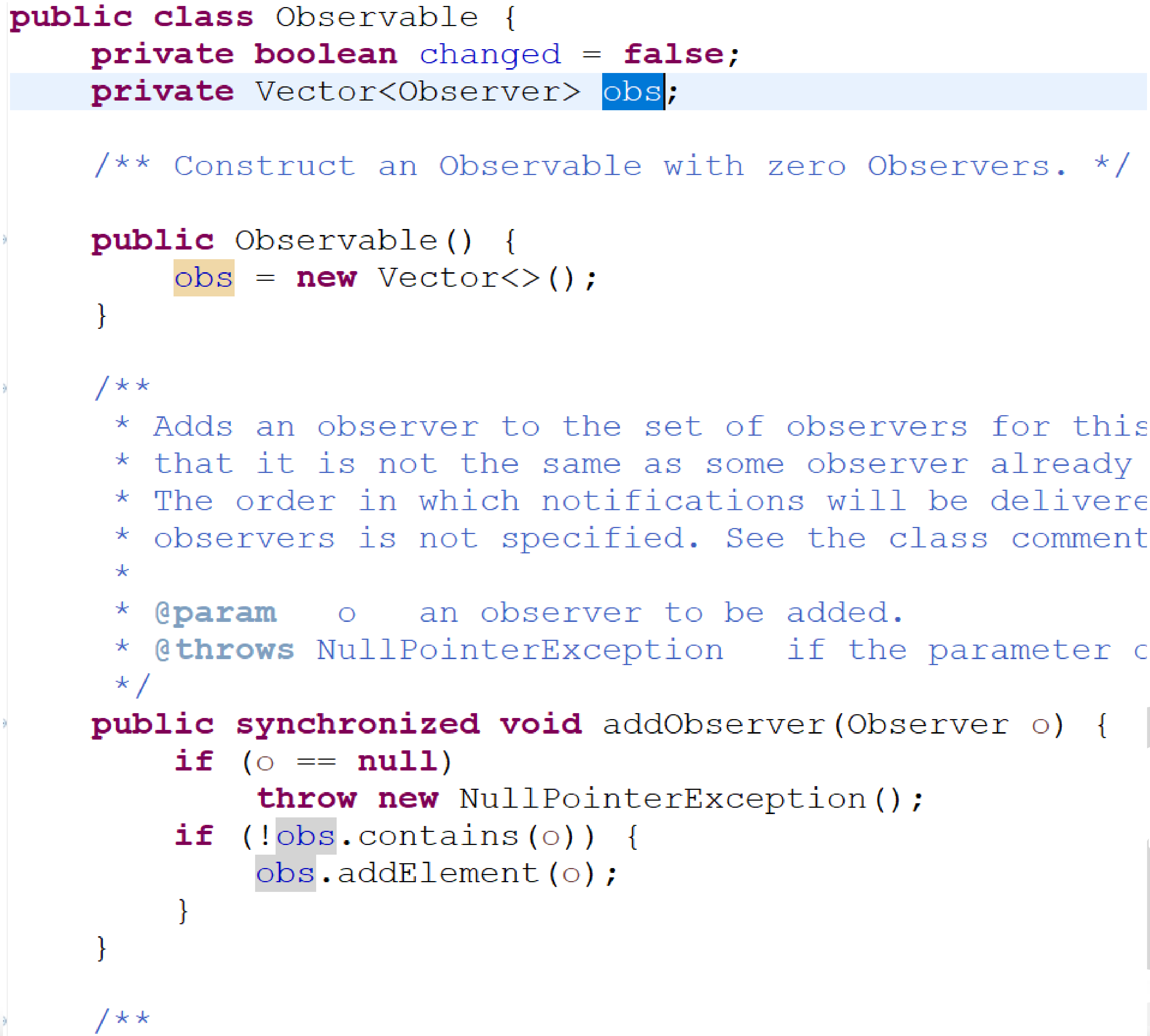
```
  /
public void notifyObservers() {
    notifyObservers(null);
}
```

通知了所有的Observer（观察者），这里观察者是App1。后面我们会发现通知后干啥？就是调用Observer的update方法



```java
public void notifyObservers(Object arg) {
    /*
     * a temporary array buffer, used as a snapshot of the state of
     * current Observers.
     */
    Object[] arrLocal;

    synchronized (this) {
        /* We don't want the Observer doing callbacks into
         * arbitrary code while holding its own Monitor.
         * The code where we extract each Observable from
         * the Vector and store the state of the Observer
         * needs synchronization, but notifying observers
         * does not (should not).  The worst result of any
         * potential race-condition here is that:
         * 1) a newly-added Observer will miss a
         *   notification in progress
         * 2) a recently unregistered Observer will be
         *   wrongly notified when it doesn't care
         */
        if (!changed)
            return;
        arrLocal = obs.toArray();
        clearChanged();
    }

    for (int i = arrLocal.length-1; i>=0; i--)
        ((Observer)arrLocal[i]).update(this, arg);
}
```

调用所有观察者的update方法



```
public class Observable {
    private boolean changed = false;
    private Vector<Observer> obs;

    /** Construct an Observable with zero Observers. */

    public Observable() {
        obs = new Vector<>();
    }

    /**
     * Adds an observer to the set of observers for this
     * that it is not the same as some observer already
     * The order in which notifications will be delivere
     * observers is not specified. See the class comment
     *
     * @param   o   an observer to be added.
     * @throws NullPointerException   if the parameter c
     */
    public synchronized void addObserver(Observer o) {
        if (o == null)
            throw new NullPointerException();
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }

    /**
```