In Java, there are several mechanisms to deal with events and the actions they should trigger. For GUI components, Java defines the ActionEvent class and an interface ActionListener for objects that should respond to such events. The ActionListener interface requires just one method:

```java
public void actionPerformed(ActionEvent e)
```

Thus, ActionListener represents pure encapsulation of behavior. Java also defines an interface Action that extends ActionListener. The Action interface requires that the object maintain a property list for auxiliary settings and that it be capable of being enabled or disabled. Java provides a base implementation of Action in the AbstractAction class that leaves actionPerformed undefined.

When Java deals with GUI components, it normally provides methods that allow programmers to add ActionListeners to react to events. However, in the construction of a component such as a JButton, Java requires an Action because the auxiliary settings are used to define the button label or icon.

Although this is a nice design, there are problems:

- In dealing with button clicks and similar events, one almost never needs to deal with the actual ActionEvent. Instead one just wants to define the response behavior in a simple fashion.
- There is no provision for response behavior to be done in a separate thread. This is often vital since performing intense algorithms in the thread that listens to the GUI will cause the GUI to appear to die.
- Java has many event classes. Often one wants to do the same response to different events but this is difficult due to the multiplicity of event classes.

### Solution Overview
In this section, we will describe various classes that solve some of the problems mentioned above.

#### SimpleAction
To make it easy for faculty and students to define Action objects that do not need the ActionEvent, JPT defines the SimpleAction class that extends the Java AbstractAction class. To complete the definition of a SimpleAction, one must define:

```java
public void perform()
```

The actionPerformed method is a final method that simply calls perform. In fact, for flexibility, SimpleAction also implements the interfaces ChangeListener and PropertyChangeListener by the same technique.

**Example:** In the submission on GUI Composition, we illustrated aspects of GUI building with the Circle Sample example.

At that time, we said that the Paint button was defined using an action. Here is the definition of that action:

```java
protected Action paint = new SimpleAction("Paint")
{ public void perform() { paint(); } }
```

As you can see, the paint action calls the paint method which we sketch below:

```java
protected void paint() {
  // define the circle using radius, x, y ...
  // get the graphics context of the window
  Graphics2D g = window.getBufferGraphics();
  // set anti-aliasing on ...
  // set the paint color and fill the circle
  g.setPaint(colorView.getColor());
  g.fillRect(circle);
  // repaint the window
  window.repaint();
}
```

Now let us explain why this works and how we teach it.

The paint action and paint method are permitted to have the same name because data and methods are in two different namespaces. Given this, we adopt a pedagogical convention that an action will be defined using a method of the same name. This makes the action definition short and focuses the behavior definition in a method where students would normally expect to see behavior.

The action definition uses anonymous inner classes but we would never use such scary words in teaching. Instead, we explain that the braces `{ ... }` provide syntax to extend the definition of an object by supplying missing methods or by permitting existing methods to be overridden. To students, this makes sense. The perform method is missing and the braces provide a simple way to supply it.

Finally, the data parameter "Paint" is attached to the NAME property of the Action and therefore becomes the button label in the GUI. We use the fact that TablePanel automatically turns actions into buttons.

In summary, the students see that we get a button from an action whose behavior is defined in a corresponding method.

#### ThreadedAction
The class ThreadedAction is a wrapper class that encapsulates an action for the purpose of performing that action in a separate daemon thread. Here is how the actionPerformed method is defined:
The abstract class `MouseAction` extends `AbstractAction` and encapsulates an action that is performed as a result of a Java `MouseEvent`. To instantiate a `MouseAction` object, one must define the method:

```java
public void mouseActionPerformed(MouseEvent mevt)
```

A `MouseActionAdapter` object is designed to listen to a specific component for `MouseEvents`. The `MouseActionAdapter` can add, remove, set, or get actions to take place if any of the seven standard `MouseEvents` takes place within the component. To illustrate what is possible, we list the seven `add` methods that add a single `Action`:

```java
public void addMouseClickedAction(Action a)
public void addMouseEnteredAction(Action a)
public void addMouseExitedAction(Action a)
public void addMousePressedAction(Action a)
public void addMouseReleasedAction(Action a)
public void addMouseMovedAction(Action a)
```

In practice, the action that is passed to one of these methods is either a `MouseAction` or a `SimpleAction`. A `SimpleAction` will perform the same task independent of the `MouseEvent` data.

The class `MouseActionEvent` is an adapter event class that extends `ActionEvent` and represents an action triggered by a `MouseEvent`. Normally, the use of `MouseActionEvent` is internal to JPT.

Example: Let us show how to add an action that will track the mouse position by printing its coordinates into 2 text fields called `xTFV` and `yTFV`. The tracking action is defined to call a method as is our standard paradigm.

```java
public void mouseActionPerformed(MouseEvent mevt)
{
    trackMouse(mevt);
}
```

The corresponding method is defined as follows.

```java
protected void trackMouse(MouseEvent mevt) {
    xTFV.setViewState(mevt.getX() + "");
    yTFV.setViewState(mevt.getY() + "");
}
```

Finally, if `adapter` is the name of the `MouseActionAdapter` for the component, then the action is installed into `adapter` as follows.

```java
adapter.addMouseMovedAction(trackMouse);
```

For many additional examples of installing mouse behavior, see Chapter 1 of the JPT Book on the JPT web site.

Experience with the Solution

The use of actions and action adapters follows a strict paradigm that is easy for students to learn and use in the context of building GUIs. More generally, students become comfortable with the idea of encapsulation of behavior in objects. This prepares them for learning the strategy pattern and for concepts in higher order languages.

API Documentation & Related Materials
The main JPT site to access documentation, code, and the jpt.jar:
http://www.ccs.neu.edu/jpt/