Introduction to Interactive Programming
by Lynn Andrea Stein
A Rethinking CS101 Project

Event Delegation and java.awt

Chapter Overview

- How do I separate an entity's core behavior (model) from its on-screen appearance (view)?
- How do intermediate (listener) objects couple together system components that don't know about one another?

In the simple event model of the previous chapter, each visible component provides an event handler method (e.g., `paint`) that is invoked every time that the appropriate event is triggered (e.g., by uncovering a window or by an explicit call to `repaint()`). The component doesn't (necessarily) have an always-active animacy (Thread); instead, it is woken -- invoked by the event dispatcher instruction follower -- whenever an appropriate event occurs.

In the previous chapter, we saw how event driven programming focuses a system's design on what to do when certain events happen. The mechanism that recognizes and dispatches these events fades into the background. We saw how this approach is used to implement painting in java.awt components. In that system, each Component handles its own events. In this chapter, we will look at a more complicated two-layer model which further separates the event producer from the event consumer. This mechanism, which relies on an explicit listener registration protocol, is at the heart of the event handling system in Java's AWT versions 1.1 and later.

The problem of GUI design is illustrative of larger design issues. The event-delegation approach described in this chapter arises from our desire to separate what happens in the GUI (such as clicking a button) from the behavior that this causes (such as playing a song). To make this work, we connect GUI objects (such as Buttons) to application objects (such as SongPlayers) indirectly, through special EventListener objects. The EventListener records the appropriate connection between GUI events and application behavior, keeping these details out of both GUI and application components. This allows significant flexibility: a single application behavior may be invoked by many different GUI events; one GUI event may give rise to many application behaviors; or the relationship between GUI events and application behavior may be remapped by a running program, for example.

This kind of indirect coupling through a Listener object is a useful technique in a wide range of applications.

Model/View: Separating GUI Behavior from Application Behavior

In the previous chapter we explored event-driven programming as a way of focusing on the important things that happen in a program. An event handler is a method that responds to some important circumstance, or event. It answers the question, "What should I do when xxx happens?" It shifts the emphasis from figuring out what has happened and deciding what to do (the dispatcher) to the actual code that handles the event, whenever it may arise. Event driven programming is the idea that an object simply provides an event handler method -- instructions to follow -- and does not worry about how or when those instructions are executed. Somehow, an instruction-follower will invoke this method -- and follow its instructions -- when appropriate.

Java's AWT graphical user interface toolkit uses event-driven programming to coordinate the display of GUI objects on your computer screen. Each java.awt.Component implements its own paint(Graphics g) method, which supplies the instructions for making that Component appear in the coordinate space described by g. As in all event-driven programs, the event handler paint method does not worry about when, why, or whether it is time to paint. When the paint method is invoked, it means that the need for painting has arisen -- the event has occurred -- and the paint method's execution simply responds to that event.

In AWT painting, the need-to-paint event happens to a particular Component. When a need-to-paint event arises, AWT makes it clear who is responsible for handling that event: the Component that needs to be painted. But there are many other kinds of events for which the question, "Who should handle this event?" does not have such an obvious answer. This chapter is about more general mechanisms that let programmers answer that question in a more flexible way, separating the Component to which the event happens from the object that handles the happening.

In many cases, the appearance of a GUI object and its underlying behavior may actually be implemented by two different Java Objects. For example, the GUI object that implements a set of radio buttons may be a `Panel` containing a number of `Checkbox`es. This is called the **view**: what the mechanism looks like, its screen appearance. In addition, when the appropriate buttons are pressed, a song may be played. This is called the **model**: how the mechanism behaves. The view -- in this case, the `Panel` -- is responsible for keeping track of the on-screen appearance of the `CheckBox`es (with their help, of course). The `Panel` need ***not*** be responsible for playing the song, though. The model, which provides the song-playing behavior, may in fact be implemented by a different object. Logically, we want to separate out the GUI appearance (and GUI behavior, e.g., buttons looking pressed or not pressed) from the underlying application behavior. Java's AWT event delegation mechanism lets us do just that.[Footnote: The event delegation mechanism described in this chapter is used in Java's AWT version 1.1 and later and also in the Java Swing toolkit. In Java's AWT version 1.0, all event handling was done using a system closer to that of chapter 15.]

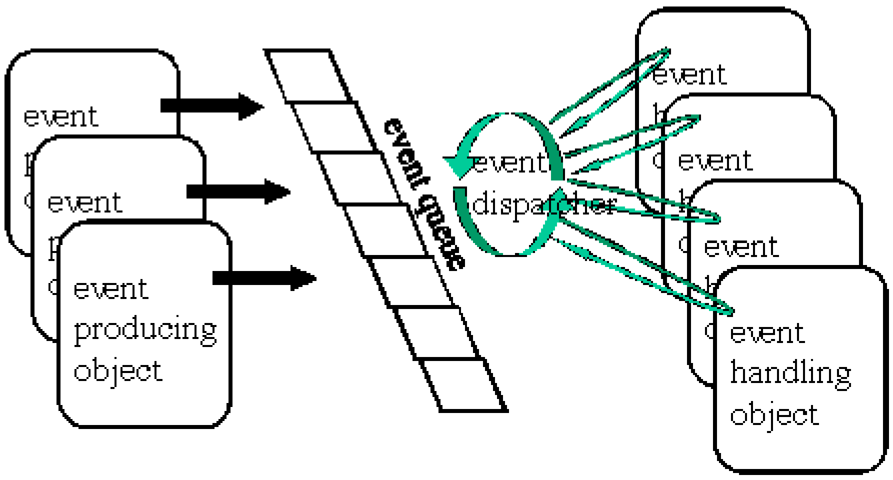


Figure 1. Dispatching Events: A fully general scenario.

The Event Queue, Revisited

In the previous chapter, we saw that we can separate the generator of an event from the actual invocation of an event handler through the use of an event queue. The event queue is a place where an event producer can "drop off" the information that an event had occurred. For example, code can call a Component's repaint() method. This adds a painting request to the event queue. Paint requests can also get added to the queue by screen events, such as a Window moving to uncover a Component or a new Window being asked to show(). Inside the queue, it doesn't matter how the event got added. A separate active event dispatcher looks at the requests in the queue and figures out which event handlers need to be called when. The event dispatcher picks up an event (or, in the case of repaint requests, perhaps several requests) and invokes the appropriate method (e.g., paint(Graphics g)).

In the case of painting, you can imagine that there is one event queue per Component. The dispatcher doesn't need to figure out what code to call; all of the requests in that queue are for the associated Component. When a need-to-paint request arises, Java ensures that that Component's paint(Graphics) method is called. The Component doesn't have to do anything more than provide a (possibly inherited) implementation for this method.

All GUI events -- not just painting -- happen to particular Components. The mouse is clicked inside a particular Component. Only one Component at a time can be listening to the keyboard.(Being the Component that is listening to the keyboard is called "having the focus".) So when an event occurs, it will still get added to the queue belonging to the Component with which it is associated.

But suppose that we want to separate even event ownership from the responsibility for handling the event. Suppose, for example, that clicking a radio button (GUI Component) causes another object -- a SongPlayer -- to play a song. If responsibility for handling the event doesn't necessarily belong to the Component -- if we are separating the Component view from a distinct Object implementing the model -- the event queue's dispatcher needs to figure out who to notify that the event has occurred. We need a mechanism for associating the events that happen (and the objects to which they happen) with interested parties that are willing to handle those events. We call these interested parties **listeners**. The system by which a separate event-handling object listens for events that occur to another (GUI Component) object is sometimes called **event delegation**.

Java solves the "who to notify" problem by introducing the idea of listener registration. You can think of this as being something like subscribing to a newspaper clipping service or personalized online news service. When you subscribe to such a service, you give the service a list of topics that you're interested in. This is registering your interest with the event queue, or listening. The service maintains a list of subscribers along with their interests. These are the registered listeners. Each time that a new article comes in, it is added to the pile of clippings to be considered. This is putting an event into the queue. An employee of the clipping service picks up a clipping (typically the oldest one) and checks to see who might be interested. If the article matches your interests, the clipping service sends you a copy. This is dispatching to the event handler methods.

Events -- such as mouse clicks or being uncovered when a Window moves -- still happen to individual Components. But -- for many such GUI events -- each `java.awt.Component` has its own event queue that can dispatch to the appropriate registered event handlers. These event handlers need to know about and register with the Component whose events they want to listen for; they need to tell the event queue which events they are interested in handling. The Component maintains a list of listeners who will handle its events.

Registering a listener is like leaving a (specialized) request with the clipping service: If any articles about Indonesian coffee come, please send them to Working Joe, and if any mouse motion events occur, please send them to the mouse motion listener that's waiting for them.

Reading What the User Types: An Example

Imagine that we want to have the user type her name into a GUI widget. When she does so, we will print a friendly greeting. This section walks through this example, providing a pragmatic introduction to the actual AWT mechanisms required to implement event delegation.

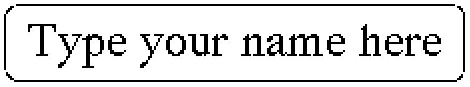
The code that follows assumes that it appears in a method within a class within a file that imports cs101.awt.DefaultFrame, java.awt.TextField, and java.awt.event.ActionListener, and java.awt.event.ActionEvent. In general in this chapter we will omit package names unless they are needed for clarity.

Setting Up a User Interaction

The first thing that we need to do is to create a place where the user can type her name. Java provides an AWT widget that is useful for just such occasions, a TextField.

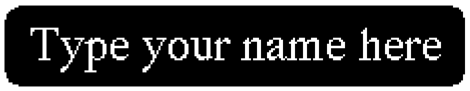
```
TextField nameField = new TextField( "Type your name here" );
```

This line creates a TextField, a rectangular box containing text. The constructor argument is the text initially displayed in this box.



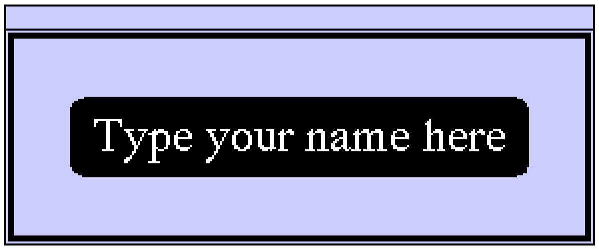
```
nameField.setEditable( true );  // Make it possible for the user
                                // to type into the TextField.
   nameField.selectAll();  // Highlight the original text so that
                        // what the user types replaces it.
```

The first of these lines makes it possible for the user to type in the TextField. The second highlights all of the text in the TextField, so that what the user types will replace the text displayed there.

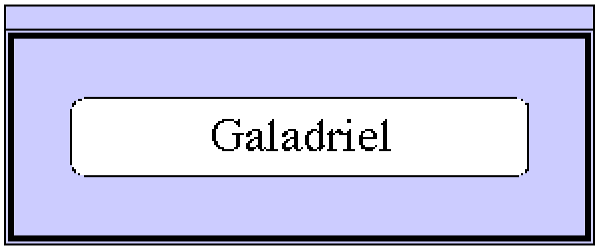


```
new DefaultFrame( nameField ).init();  // Create a Frame around the TextField.
```

Finally, this line creates a cs101.awt.DefaultFrame, an awt Window in which a single Component can be displayed. DefaultFrame is a restricted kind of Frame, but has the advantage that it takes care of certain housekeeping details for you. DefaultFrame's init() method actually makes the window appear on the screen. See the sidebar on DefaultFrame for details.



Now suppose that the user types her name into the TextField box, replacing the highlighted text previously displayed. If the user ends her name by typing the return key, this causes an action event to be registered on the TextField. In other words, something has happened and we are ready to invoke the appropriate event handler.



Now, we are ready to print our greeting. For example, we might say

```
          Console.println( "Hello, " + reference_to_nameText.getText() );
```

Each TextField has a getText() method that returns the String displayed in the TextField at the time of the getText() invocation. So, if we execute code along these lines, the text

```
          Hello, Galadriel
```

should appear on the Java Console. There are, of course, a few issues:

1. Where does this code appear? That is, who is handling the event, and in what method?
2. How does that event handler access the TextField called `nameText` (in order to ask it to `getText()`)?

This is where Java's event delegation system comes in.

cs101.awt.DefaultFrame

A cs101.awt.DefaultFrame is a cs101 utility provided to make it easy to put up a window containing a single Component. The DefaultFrame takes care of sizing, activating the window's close box, causing the window to appear on the screen, etc.

If c is a Java component, it can be made to appear on the screen using

```
new cs101.awt.DefaultFrame( c ).init();
            // Create a Frame around the component.
```

The first half of this statement is an object construction expression that creates a DefaultFrame around c. The second half of the statement invokes this DefaultFrame's init() method, which is useful for its side effect: it displays Component inside the DefaultFrame, i.e., in its own window. Of course, you can use a more complex version of this code that names the new DefaultFrame, allowing you to use it elsewhere in your program, if you wish:

```
cs101.awt.DefaultFrame frame = new cs101.awt.DefaultFrame( c );
         frame.init();
            // Create a Frame around the component.
```

The class cs101.awt.DefaultFrame extends java.awt.Frame, documented in the AWT Quick Reference appendix to this book. For the complete code implementing cs101.awt.DefaultFrame -- which is straightforward -- see the online supplement to this book.

Listening for the Event

The event generated by Galadriel's return is associated with the TextField called nameField. That TextField is like a clipping service, and a new item of potential interest -- the action taken by Galadriel -- has just arrived. Now, Java needs to determine who is interested in nameField's action events.

Who might be interested? There is a special interface, called ActionListener, that describes the contract to be implemented by any object interested in handling action events. Here is the definition of the `ActionListener` interface:

```
public interface ActionListener extends EventListener {

     public void actionPerformed( ActionEvent ae );

}
```

The actionPerformed method is an event handler, so its implementation will answer the question, "What should I do when an action is performed?" In this case, the answer is to print out the text currently displayed by the TextField in which Galadriel typed her name. The object whose actionPerformed method is invoked is not responsible for deciding whether, when, or why the actionPerformed method should be called. It is only responsible for behaving appropriately when the event handler method is called.

We can build an action listener by providing a class that implements this interface. The implementation of actionPerformed in this class is an answer to the question, "What should I do when an action is performed?"

```
public class FieldHandler implements ActionListener {

     private TextField whichText;

     public FieldHandler( TextField whichTextToHandle )
```

```
        {
            this.whichText = whichTextToHandle;
        }

        public void actionPerformed( ActionEvent ae ) {
            Console.println( "Hello, " + this.whichText.getText() );
        }

    }
```

This class actually keeps track of which TextField it wants to associate itself with. We can create a particular FieldHandler associated with nameText using the construction expression

```
new FieldHandler( nameText )
```

Now, when this FieldHandler's actionPerformed method is invoked -- when the action happens -- the FieldHandler will use nameText's getText() method to print a greeting to Galadriel.

Of course, we might want to hang on to that FieldHandler once we've created it....It will come in handy in another few paragraphs.

Registering Listeners

So far, so good. However, we haven't specified how the `FieldHandler` gets notified about the event in the first place. Of course, part of the story is that Java's event manager identifies that a carriage return has been hit in the TextField and generates an appropriate `ActionEvent`. But this event happens to the `TextField`; how does the `FieldHandler` get hold of it?

The answer is that Java needs to be notified that the `FieldHandler` is interested in this `TextField`'s action events. To return to our earlier analogy, the FieldHandler needs to subscribe to the TextField's action event clipping service.

This is accomplished with the `TextField`'s `addActionListener` method, which takes an `ActionListener` as an argument. The `addActionListener` method tells Java that the `ActionListener` argument `addActionListener` is wants to know about any `ActionEvent`s that occur to this `TextField`. For example,

```
ActionListener nameHandler = new FieldHandler( nameText );
nameText.addActionListener( nameHandler );
```

[Footnote: or simply `nameText.addActionListener( new FieldHandler( nameText ) );`]

registers the `actionListener` called `nameHandler` as a listener for any `ActionEvent`s that occur to `nameText`.

Now, when Galadriel finishes typing, an action event will not only be generated but also forwarded to `nameHandler` to handle.

Recap

The code that creates this situation is distributed over the paragraphs above. Here is the entire setup code. It might, for example, appear in a main method or in the constructor of an entity that provided the name-greeting behavior described at the beginning of this section.

```
        // Set up the TextField.
              TextField nameField = new TextField( "Type your name here" );
        nameField.setEditable( true );  // Allows user typing.
              nameField.selectAll();          // Highlights current text.

        // Now create and register the ActionListener
              ActionListener nameHandler = new FieldHandler( nameText );
        nameText.addActionListener( nameHandler );

              // Finally, create a Frame around the TextField.
              new DefaultFrame( nameField );
```

The only additional code required is the FieldHandler definition:

```
     public class FieldHandler implements ActionListener {

          private TextField whichText;

          public FieldHandler( TextField whichTextToHandle )
          {
              this.whichText = whichTextToHandle;
          }

          public void actionPerformed( ActionEvent ae ) {
              Console.println( "Hello, " + this.whichText.getText() );
          }

     }
```

Specialized Event Objects

In Galadriel's example, we encountered an object whose type was ActionEvent. It appears as a parameter in the actionPerformed method of ActionListener. In that example, we blithely ignored the ActionEvent -- as one often does in an action Performed method -- but this begs the question of what that object is and why it appears. In this section, we'll look at ActionEvent and other similar event objects, and explore cases in which these event objects have important roles to play.

In the previous chapter, we looked at an event handler method called paint. That method needed to be supplied with a fairly specific kind of object, a Graphics, before it could do anything. In contrast, other handler methods of the previous chapter -- such as handeTimeout() and handleReset() -- needed no arguments at all. The event handlers in this chapter do need some information, but that information is of a fairly generic (though specializable) type. The information supplied to one of these AWT event handlers is a special Java object called an AWTEvent. Such an object inherits from `java.awt.AWTEvent` (which is itself a `java.util.EventObject`). The subclasses of `java.awt.AWTEvent` live in a separate package, called `java.awt.event`.

In a general GUI, what kinds of things can happen? The mouse can be moved and clicked and dragged, the keys can be pressed, windows can be closed, menu items can be selected, text can be entered, and many, many more things can happen. A listing of the major event types used in this book may be found in the AWT Quick Reference appendix in the AWT Events segment. For example, a mouse click generates a `MouseEvent`, while clicking in the close box of a window generates a `WindowEvent` and clicking a button (or typing return in a text field) causes an `ActionEvent`.

Some kinds of events, like `ActionEvent`s, are notable mostly for happening. For example, when a Button is clicked, an `ActionEvent` is generated. If you know what Button was clicked to generate the `ActionEvent`, you really know everything worth knowing about the `ActionEvent`. (If you don't know what Button was clicked, you can find out by asking the `ActionEvent`; see below.) An `ActionEvent` is also generated when the return key is typed in a TextField (as we have seen), indicating that the text is complete. In this case, you need to know both which TextField and, perhaps, what text was typed. But once you know what TextField generated the ActionEvent, you can ask the TextField for its text. So the internal structure of an ActionEvent is not likely to be of much interest.

Different kinds of events have methods that provide access to the different kinds of information that you'd want if you were dealing with a mouse click or a window close. These event methods are summarized in the AWT Events segment of the appendix AWT Quick Reference. For example, a `MouseEvent` has a few methods that are especially worth noting. If the `MouseEvent` is labelled `mickey`, then

- `mickey.getX()` returns an int specifying the mouse's location at the time of the MouseEvent (in pixels starting at the upper left-hand corner of `mickey`'s screen-space). .
- `mickey.getY()` similarly returns `mickey`'s y coordinate.
- If you prefer to get both coordinates at once, you can retrieve a `java.awt.Point` object using `mickey.getPoint()`.

Every AWTEvent also has a getSource() method. This method returns the Object to whom the event happened. For example, we could have replaced the actionPerformed method of our FieldHandler class with the definition

```
          public void actionPerformed( ActionEvent ae ) {
              TextField theField = (TextField) ae.getSource();
                    Console.println( "Hello, " + theField.getText() );
          }
```

This text uses the TextField that is the source of the action event, rather than the TextField that is handed to the FieldHandler constructor, as the target of the getText() method.[Footnote: In this case, we could simply eliminate the constructor, making the FieldHandler definition look like this:

```
     public class FieldHandler implements ActionListener {

          public void actionPerformed( ActionEvent ae ) {
              TextField theField = (TextField) ae.getSource();
                    Console.println( "Hello, " + theField.getText() );
          }

     }
```

]

Some AWTEvents, such as MouseEvent, are ComponentEvents. Every `ComponentEvent` also has a `getComponent()` method that returns the same thing as its `getSource()` method, but typed as a Component.

A variety of useful event types and their methods are documented in the AWT Events segment of the AWT Quick Reference appendix.

Listeners and Adapters: A Pragmatic Detail

Every `AWTEvent` type has an associated `Listener` type.[Footnote: Except PaintEvent, which uses the mechanism described in the previous chapter rather than the listener registration system described here.] This means that when the AWT event occurs -- the mouse is clicked or the key is pressed, etc. -- there's a type of object equipped to handle that event. (Actually, `MouseEvent` is an exception, as it has two associated listener types: `MouseListener`, which handles clicks, entry and exit, presses and releases, and `MouseMotionListener`, which handles drags and moves. Most event types only have one Listener.)

The `ActionListener` defined above will do the trick quite nicely for our `TextField`. The `ActionListener` interface only had a single method to implement. Other listener interfaces are more complex, though. For example, the `MouseListener` interface defines five methods:

```
public interface MouseListener extends EventListener {

     public void mouseClicked( MouseEvent mickey );
     public void mouseEntered( MouseEvent mickey );
     public void mouseExited( MouseEvent mickey );
     public void mousePressed( MouseEvent mickey );
     public void mouseReleased( MouseEvent mickey );

}
```

If you want to be able to respond to mouse clicks, you will need to implement a class that has an appropriate mouseClicked method. But the MouseMotionListener interface specifies a contract with five distinct methods. If clicks are the only kind of MouseEvent that you want to respond to, it would be rather annoying to have to implement each of the other four methods just to be able to write the one (`mouseClicked`) that we need. Our class definition might say

```
public class MouseHandler implements MouseListener {

     public void mouseClicked( MouseEvent mickey ) {
         // Interesting code goes here...
     }
     public void mouseEntered( MouseEvent mickey ) {}
     public void mouseExited( MouseEvent mickey ) {}
     public void mousePressed( MouseEvent mickey ) {}
     public void mouseReleased( MouseEvent mickey ) {}

}
```

Not very concise or beautiful, but necessary if we are to implement the interface directly. After all, an interface is a contract and implementing the interface means fulfilling the whole contract, not just a part of it.

To avoid this ugliness, `java.awt.event` gives us a more concise way of saying the same thing. There is a class called `MouseAdapter` that `implements MouseListener`, providing all of the (non-interesting but also non-abstract) method bodies required. We can just `extend MouseAdapter` in our class, eliminating the need to implement all of the extra (extraneous) methods:

```
public class MouseHandler extends MouseAdapter {

     public void mouseClicked( MouseEvent mickey ) {
         // Overrides MouseAdapter's mouseClicked method.
         // Interesting code goes here...
     }

}
```

Much nicer!

Each of the listener interfaces that declares more than one method has a corresponding adapter class. These are listed in the AWT Listeners and Adapters segment of the AWT Quick Reference appendix.

Inner Class Niceties

Let's return to the TextField handler class from the Galadriel example, above. There are still some improvements in functionality that we can make.

We might, for example, make our own class -- our own specialized `TextField` -- that is born with its own `FieldHandler`:

```
public class HandledTextField extends TextField {

     public HandledTextField() {
         ActionListener nameHandler = new FieldHandler( nameText );
         nameText.addActionListener( nameHandler );
     }

}
```

Now each HandledTextField is born with its own FieldHandler. This is similar to AnimateObject's creating its own AnimatorThread, rather than expecting someone else to create the AnimatorThread on its behalf.

Using inner classes,[Footnote: See chapter 12 for details.] we can make this innovation do even more work for us. Inner classes are a relatively advanced feature of Java, and they add only to the aesthetics of this program, not to its functionality. They do provide a little bit more protection for code from unanticipated use, a feature that we can exploit. After all, a FieldHandler as we have defined it is not really of much general interest. We can embed the definition of that class inside the HandledTextField class definition, hiding it from the rest of the world and simultaneously taking advantage of inner class's privileged access to their containing instance's state.

Using inner classes, we can write:

```
public class HandledTextField extends TextField {

     public HandledTextField() {
         ActionListener nameHandler = new FieldHandler();
         nameText.addActionListener( nameHandler );
     }

     private class FieldHandler implements ActionListener {

         public void actionPerformed( ActionEvent ae ) {
             Console.println( "Hello, "
                                  + HandledTextField.this.getText() );
         }
     }

}
```

Since `FieldHandler` is defined inside `HandledTextField`, it has access to its containing instance directly (through `HandledTextField.this`), and we can eliminate the constructor argument (and the constructor itself!) for `FieldHandler`. Pretty neat, huh?

Chapter Summary

- EventListeners are interfaces promising particular sets of event handler methods. There are Listeners for groups of related AWT event types, such as mouse motion events, in the package java.awt.event.
- That package also includes adapter classes to make implementing these interfaces easier.
- Listeners are connected to AWT components using a component's `add`*`EventClass`*`Listener()` (registration) method.
- `java.awt.AWTEvent` and its subclasses are data repositories that record relevant information about individual (GUI) events.
- Each event handler method takes one of these Event objects as an argument, in much the same way that `paint()` requires a `Graphics`. Like `paint(),` the event handlers of an EventListener are called by the system, not by your code.
- Inner classes provide a nice way of packaging the definitions of subsidiary classes (such as EventListeners) inside other class definitions.

Exercises

1. Define a class that implements `java.awt.event.MouseListener` and extends `the mouseClicked(MouseEvent) method` by printing the coordinates of the point on which the mouse had clicked. You may also want to make use of the class `java.awt.event.MouseAdapter`. (Bonus: also print the components of the *previous* mouse click.)

2. Now define a class that extends `java.awt.Canvas` and sends its mouse events to your `MouseListener`.
3. Define a class that implements `java.awt.event.WindowListener` and extends the `windowClosing()method` by printing `"Nah, nah, you can't kill me!"` (Alternately, you can do the potentially more useful thing and (1) call the object's `dispose()` method and (2) call `System.exit(0)`.) What class do you think would be useful when implementing `WindowListener`?
4. Define a class that extends `java.awt.Canvas` and looks like a (black and white) Japanese flag, i.e., it has a circle at `(100,100).` Make the circle change color when the mouse is over your Canvas. (Hint: mouse enter, mouse leave.)