Chapter 4

Event Infrastructure

4.1 Introduction

Hopefully we have convinced you in the first several chapters that event based programming is a distinct paradigm from algorithmic programming (either procedural or object–oriented). An event firing has different semantics than a method call. Its unique semantics require additional support from the underlying system. The support infrastructure may be implemented in hardware or software, but must be there.

Languages and libraries that support event based programming include an infrastructure of event services that makes developing and running event based programs easier. Programmers frequently write code that use these services without really understanding their details, treating event processing as a black box. Their code compiles and runs, using the event infrastructure, but the programmer never really knows how the pieces fit together to get the job done.

Our purpose in this chapter is to open up the black box and look in detail at the various features that support event processing. The details obviously depend on the language, library or hardware used. However, there is a common core of services needed in any event based infrastructure. Understanding the ideas behind the core services helps us gain a deeper understanding of how event based systems work. This is necessary if we are going to do more complex event based programming, like designing our own event classes.

Event based programming is distinct from algorithmic programming. Algorithm design deals with how to represent and process the data necessary to carry out a computation and how to decompose the algorithm into simpler, more manageable parts. Algorithm decomposition is often accomplished through the use of procedures (or subroutines) that carry out sub-tasks of the algorithm. The algorithm is accomplished by carrying out processing steps in the procedures, each of which contributes to achieving the goals of the algorithm.

How are event based systems different from algorithms? First, an event based system is a *system* that has parts that interact with each other and that has behaviors that change over time. The parts of an event based system are typically heterogeneous, as well as spatially and temporally separated. The relationships among the parts are dynamic. Moreover, event based systems interact with the external environment, making their inputs – and therefore their behaviors – unpredictable.

As we have already described in section 1.4.2, event based systems are comprised of loosely coupled parts that interact with each other via events. In a procedural setting, when algorithm A calls procedure P, control is passed from A to P, and P maintains control until it returns to A. In an event based setting, when an event E is fired in the context of an event source S, processing in S can continue to take place in parallel with the handling of the event E.

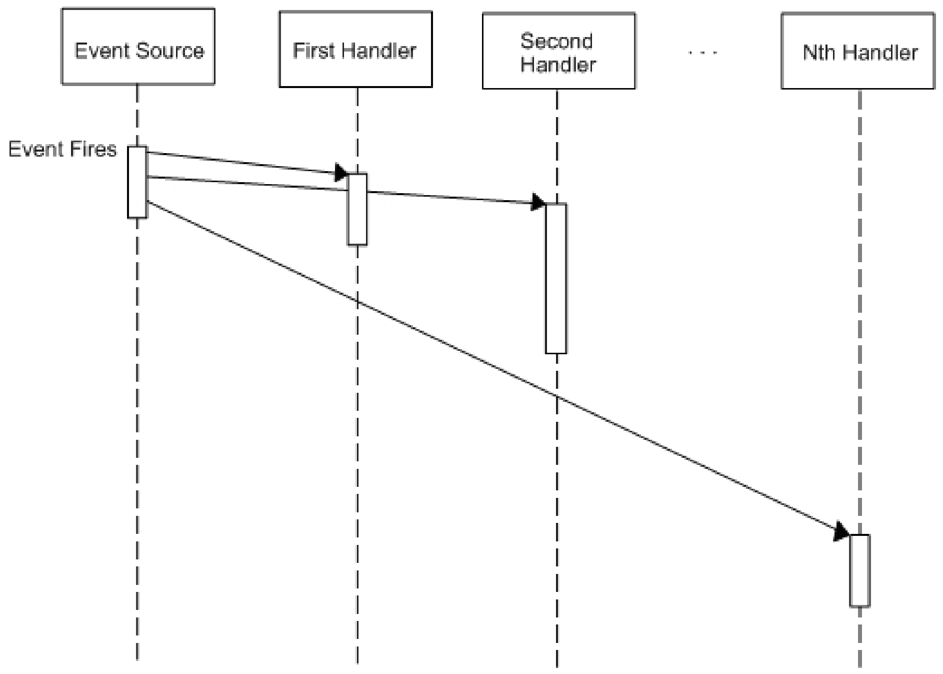


Figure 4.1: This figure shows the critical times during an event's life, when the event fires, when event handling begins and when the last handler completes its processing.

Finally, as an event propagates through a system, it may *morph* or change. "Low-level" events morph into "high-level" events taking on meaning within their current context. For example, a mouse click at the hardware level may be represented by a bit that is set in a device controller, but as the click is processed it takes on successively more characteristics such as screen coordinates, being identified with a process, and finally being regarded as activating a menu item.

As we saw in the previous chapter, event handlers are programmed in much the same way we program procedures and methods. As we have just described, however, the semantics of events are quite different than procedure calls. The event processing infrastructure exists to support the event semantics.

4.2 The Event Life Cycle

Naively, we have described events as being fired and handled almost simultaneously. That is, when an event such as a mouse click fires, we often perceive no delay between the firing of the event and the handling of the event. In truth, when an event occurs, a possibly complex sequence of steps is performed that ultimately ends up with the event being processed. Firing an event includes creating a data structure or object that encapsulates the critical event data and storing it in a place accessible to the event dispatching infrastructure. Sometime thereafter, the event handlers begin executing, and still later, with possible interruptions in-between, the last handler for the event finishes. These steps are illustrated in Figure 4.1. If many events occur in rapid succession, there may be a noticeable delay between when an event fires and when handling begins.

4.2.1 Event Handling Requirements

There is one universal requirement for event based systems:

> Sometime after an event fires, the system achieves a state consistent with the behavioral requirements defined for the event.

For example, in our drawing program in Chapter 2, we pressed and dragged the mouse and we expected a line to be drawn. If the line wasn't drawn, the system wasn't functioning correctly. *Behavioral requirements* tell us how a system is supposed to function. Every event has behavioral requirements, possibly varying based on the control state of the system.

Note that there is wiggle room in this definition. Achieving "a state consistent with the behavioral requirements" is not the same as saying that the handlers for this event executed in a particular order. Reordering or overlapping event handler execution and possibly combining handlers are both acceptable, as long as the system achieves a consistent state.

4.2.2 Timing Requirements

In event based systems, the requirements may include deadlines for completing event processing. As we saw in Chapter 1, event based programming is used to implement many types of systems. To a certain extent, the type of system guides the requirements. For instance, real–time and embedded systems frequently have hard deadlines for their event handlers to complete.

- Consider driving down the highway with your cruise control system on. A deer leaps out in front of you and you slam on the brakes. You want the cruise control system to disengage within a few milliseconds after touching the brakes, not one second or even a half second later. There is a very important deadline for handling the braking event that must be met for the system to work correctly.

- By contrast, when you press the down button to summon an elevator, the button should be backlit almost instantly, and you expect the elevator to stop at your floor – but if you wait one or two minutes for the elevator to arrive, it is not likely to matter.

If systems have timing requirements we should recognize those requirements explicitly and design our systems to guarantee that they are met.

4.2.3 Concurrent Event Handling

The final piece of background we need before getting into the infrastructure details is an understanding of *concurrency.* If two things are happen in overlapping time spans, we say they happen *concurrently.* For example, in real–time systems it is common to find multiple dedicated processors handling events. The events are handled concurrently making it easier to meet the required deadlines.

Similarly, many modern desktop computers have with multiple *cores.* You can think of cores as multiple processors, all physically located on one chip. Software doesn't automatically use more than one core. Programmers must design and implement their systems to take advantage of the additional computing power.

Modern languages like Java use an abstraction called a *thread* to implement concurrency within a program. For example, GUIs often use a separate thread for event handling. The thread loops forever, repeatedly looking for the next event and invoking its handlers. This topic is discussed more completely in the next chapter.

4.3 Event Infrastructure Services

There are two core event services that any event infrastructure should supply: *registering/unregistering handlers* and *event dispatching*. There are other event services that may be provided, too. For example, logging events can be a great help in tracking down problems, and support for remote events can help the programmer develop distributed systems. These advanced services are not covered in this chapter.

4.3.1 Event Registration

As we have seen, an event handler registers its interest in a particular event with the event's source. The event source keeps track of the handler until either it is unregistered or the application terminates.

Some event systems use *unicast* events. Unicast event sources have at most one handler for each event. For example, *The GL Utility Toolkit, (GLUT)* which is part of *OpenGL* uses unicast callback functions for handlers. In GLUT there is one callback function to handle an application's mouse events. Registration of unicast event handlers is simple: The event source contains a reference to the handler.

Multicast event systems may have multiple handlers for each event. When an event fires each of the handlers is notified. This may sound confusing, but multicasting is not as unusual or complicated as it may first seem. For example, consider our double list example from the previous chapter. There were three handlers registered to receive the data changed events: the list's view, the average calculator and the maximum calculator. Each had clearly defined responsibilities. All three were notified when the event occurred.

Supporting multicasting requires event sources to maintain a list (or other data structure) of handlers. It also makes event dispatching a bit more complicated.

4.3.2 Event Dispatching

The *event dispatcher* is responsible for invoking the event handlers when an event fires[1]. Broadly speaking, we divide event dispatching approaches into two categories: *push* and *pull*.

- In *push* dispatching, the event source is responsible for activating the dispatcher when an event occurs. We say that the event source pushes the event to the dispatcher. The dispatcher is then responsible for invoking the handlers. Note that it is possible to have the event source call the event handlers directly. In this case the source is also the dispatcher.

- In *pull* dispatching, the dispatcher periodically queries or *polls* the event sources for events. When it finds one, it calls the relevant handlers.

The difference between *push* and *pull* dispatching is really one of active responsibility. In a pull system, the dispatcher is responsible for detecting that an event has occurred, typically by polling the event source. In a push system, the event source activates the dispatcher or enqueues the event to be handled later. A few examples may help clarify the differences.

[1]In some sense the event dispatcher is itself an event handler. It is code that executes as a result of an event occurring. For purposes of our discussion, however, we distinguish between code that is supplied by the infrastructure, the dispatcher, and code that is application specific, the handlers.

Push and Pull Examples

Suppose that you work for XYZ company as a sales representative. Your objective is to take orders from customers and to process these orders. In this example, receiving an order from a customer is the particular event we are interested in.

As a sales representative, you might simply sit at your desk waiting for customers to call you to place orders. You register your willingness to accept orders by giving your customers your office phone number. This is an example of push dispatching, since the customer (the event source) is calling you (and thereby activating you, the event dispatcher) to process the order (handle the event). You can sleep at your desk until your phone rings.

On the other hand, you might have a set of customers that you visit periodically on a sales route. As you visit each customer, you ask if they have any orders to place (event polling), and you process any orders that they may have. It may well be that a particular customer has no orders to place during your visit, in which case you will simply return on your next visit to check again.

It's possible, of course, for XYZ company to combine these two approaches – the company may well want to do so to maximize their opportunities to receive orders by phone as well as to pay personal attention to their best customers. However, in event based computing systems it is rare to have events dispatched with both push and pull approaches.

Systems Using Pull Dispatching

Operating systems are responsible for interacting with peripheral devices such as a mouse, keyboard, or disk drive. For example, when you click a mouse button, the hardware device controller for the mouse (typically a chip on a computer motherboard, but not part of the main processor) may register the fact that the button has been clicked by setting a bit in an internal device controller register. The operating system can periodically check the device bit to see if it is set, and if so, the operating system can treat it as a mouse click event. The device controller will normally reset the bit after it has been read, so that the device can register multiple mouse click events.

Both the **X11 Window System** and **Microsoft Windows** generate myriad events such as mouse clicks (at a higher level of abstraction than described above) and window expose events. In an application, these events are typically processed in an *event loop* that might look something like this (*WARNING:* `C` *code alert*):

```
/* Xlib event loop */
while (1) {
  XNextEvent(display, &report);
  switch (report.type) {
    case Expose:
      /* handle an Expose event ... */
      break;
    case ConfigureNotify:
      /* handle a ConfigureNotify event ... */
      break;
    case ButtonPress:
      /* handle a ButtonPress event ... */
      break;
    ...
    default:
      break;
  } /* end switch */
} /* end while */
```

The loop repeatedly requests the next `X11` event and, based on the event type, handles the event appropriately. The event loop also elucidates why we use the term *dispatcher*. It receives events of many different types and, based on the type of an event, dispatches each to the appropriate handler.

The graphics packages `OpenGL` and `GLUT` similarly use event loops to handle windowing events [Angel, 2008, Shreiner et al., 2007]. After registering various event handlers, the `GLUT` event loop `glutMainLoop()` is called, which serves the same purpose as the `Xlib` event loop given above.

Systems Using Push Dispatching

- In our example using pull dispatching, we described how an operating system could poll a mouse device controller bit to determine when a mouse button has been pressed. Most high-performance operating systems do not use polling, however, because of the overhead associated with it. Instead, high-performance systems (including most consumer-oriented workstations and laptops) use *interrupt processing* to handle such events.

  An interrupt-driven device has the capability of notifying the computer processor that an event (such as a mouse press) has occurred. When an interrupt occurs, the processor temporarily suspends the current program and identifies what device caused the interrupt by probing external bus lines. The processor then looks up the address of the device interrupt service routine (found at a fixed location in memory based on the device identification) and executes the code found at that address. When the interrupt service routine has finished, the processor resumes the suspended program. In this case, the interrupt service route plays the role of the event handler.

  Notice that for interrupts to work, the address of the interrupt service routine must be installed at the proper location in memory corresponding to the device. This is how the handler gets registered with the event source.

- Java and Microsoft .NET both use push dispatching. You have already seen examples in Chapter 2 that illustrate how handlers are registered with event sources in Java.

The Event Queue

In push dispatching, the event dispatcher may activate an event handler by calling it directly. If the handler does not return quickly enough, a second event may arrive at the dispatcher while the first event is still being handled. Two potentially bad things could happen in this case: either the dispatcher could try to call the event handler again – with possibly unpredictable results [Stallings, 2008, Tanenbaum, 2007], or the dispatcher could discard the second event – again with unpredictable results.

A good way to resolve this situation is to store the second event until the first one completes its processing. What if multiple events occur while the first is being handled? Each of the events should be stored until processed. In many systems, a dedicated *event queue* exists to handle this situation. The event queue is a data structure holding events waiting to be handled. When the event dispatcher finishes processing one event it gets the next event from the queue.

An event queue must have a large enough capacity to store pending events without the likelihood of running out of queue space. If the event handler takes too much time to finish, events can pile up in the queue so that the queue fills to capacity. In this case, subsequent events may be discarded, or the system may crash – neither of which is desirable. Good event based system design will have handlers that return quickly and event queues that are sufficiently large to handle pending events. Obviously ,such designs must take into account processor and device speed and must carefully craft the handlers to optimize their performance.

As mentioned above, it is possible for each event source to function as its own event dispatcher; having its own event queue, but this would be a waste of system resources. Each event queue would need to have a capacity large enough to accommodate the worst case number of pending events. Instead, most systems have a single event dispatcher with a single event queue that holds all pending events. Each event source is responsible for enqueuing (pushing) its events. The dispatcher, running in a separate system thread, dequeues the events and activates their respective handlers.

Event Ordering

Events are generally put on the event queue in the time sequence order in which they were fired, so that if event A occurs before event B, event A will be put on the event queue and handled before event B. This behavior is called *event ordering*. Observe that event ordering is another reason to have a single event queue rather than one for each event source: having multiple queues would mean that event ordering might be preserved by each queue, but there would be no guarantee that event ordering would be preserved between queues.

Even with multiple queues, event ordering can be preserved if the queued events have *timestamps* that record the time at which they were fired. The dispatching mechanism would then be expected to handle events in the proper order based on their timestamps. The situation becomes more complicated in a distributed system: one in which the event sources are on different hardware platforms and perhaps are physically distant from one another. In this case, timestamps would be necessary to ensure that event ordering was preserved in the presence of network delays. But this also requires that all the entities in a distributed system would need to share exactly the same system time – a problem in its own right [Lamport, 1978].

Event Coalescing

An event dispatcher may also implement *event coalescing*. Coalescing events is the process of combining multiple events into one. Sometimes nearly identical events occur such that handling the second event separately would leave the system in the same state as having the handler called only once. Also, duplicate events can occur in rapid succession because of physical phenomena. Events waiting in the queue can be examined, and if there are events that can be safely coalesced, the system does so.

A practical example of event coalescing is an elevator interface. A hotel guest requests elevator service at a particular floor by pressing the down button. If more than one guest is waiting before the elevator arrives, or if a guest grows impatient, the button may be pressed multiple times, but there is no need for a downward traveling elevator to service the floor more than once. The button presses are coalesced into a single event.

Another example is window repainting. Some graphics systems use events to signal that a display window (or region) needs repainting. If a program tells the system repeatedly that a window needs repainting, there is no need to repaint more than once, as after one repainting, the window will be up-to-date[2].

A final example is keyboard debouncing. A human operating a keyboard or keypad is able to press keys at a maximum rate of less than 20 per second [Norris McWhirter, 1985], giving a minimum 50ms delay between key presses. Sometimes keyboard devices, because of physical or electrical properties, may record two key presses within a short time frame (1ms to 2ms) that cannot possibly be the result of user interaction with the device. Such a phenomenon is called *keybounce* [Hyde, 2003]. To counteract keybounce, two key press events from the same human-operated device should be coalesced into one if they occur within a few milliseconds of each other (based on

[2]In modern versions of Java, repaint coalescing is handled by the `RepaintManager`, not by the event dispatcher.

their timestamps, for example). The failure to do so can have serious unintended consequences [Food and Drug Adminstration, 2006][3].

4.3.3 Hybrid Dispatching

The pull-push differentiation provides a nice conceptual model and does a good job describing many dispatching systems. More complex models may be required when three or more active agents are involved.

In distributed event systems, the source, the event dispatcher, and the handler(s) may exist on separate computers. The source may push an event onto the dispatcher's queue, or the dispatcher may poll the source. Similarly, the dispatcher may push an event to the handlers, or handlers may poll the dispatcher's queue to see if there are any events. We will discuss these issues more completely in a later chapter.

Publish-subscribe systems have a similar three-tier structure [Muhl et al., 1998]. Publishers deliver (push) content to a repository agent. Subscribers either register their interest in published content with the repository agent which then delivers (push) the content to the subscribers, or the subscribers scan the repository (pull) for content that they want to receive.

4.4 Java's Support for Events

To illustrate the practical issues of implementing an event infrastructure, we look at how it is done in Java.

4.4.1 Handler Registration

Multicast events require the event source to maintain a list of handlers. Java Swing components can multicast their events, so all Swing components, e.g. `JButtons`, `JTextfields`, `JPanels`, etc., maintain a list of listeners. All these components inherit (directly or indirectly) from `JComponent`, so it makes sense to declare the list there and use inheritance to obtain access. `JComponent` contains the declaration:

```
    protected EventListenerList listenerList = new EventListenerList();
```

`listenerList` is the name of the list. The `EventListenerList` class is a wrapper around an array of `Objects`. This give the Java virtual machine fast access to all the listeners.

To add handlers to the list, we invoke an `add***Listener()` method. For example, `JButton` inherits its `addActionListener()` method from its parent class, `AbstractButton`. The `addActionListener()` code is:

```
1 public void addActionListener(ActionListener listen) {
2     listenerList.add(ActionListener.class, listen);
3 }
```

Note that on line 2, `listenerList`'s `add()` method takes two parameters. This requires a bit of explanation. As you should recall from Chapter 2, Swing components can fire all sorts of events, e.g. `ActionEvents`, `MouseEvents`, `MouseMotionEvents`, `KeyEvents`, etc. The `listenerList` is used for all of these, not just `ActionEvents`. When an event fires, the component needs to search the list for just the appropriate listeners. The code above adds both the class (`ActionListener.class`) and the handler (`listen`), to speed up searching the list. An argument can be made that multiple

[3]Note that keybounces can also be caused by the user and can be a serious impediment to typing for individuals suffering from tremors [Trewin and Pain, 1998].

lists of handlers, one for each type of event, should be maintained. While this would alleviate the need for searching the list, there would be significant overhead within each component, much of it never used.

4.4.2 JCL's Event Classes

Java's class libraries come with hundreds of event classes. The classes form a subtree within Java's class hierarchy: see Figure 4.2. At the root of the subtree is `EventObject`. `EventObject` is a very simple class. Since all Java events contain references to their source, there is a protected instance variable named `source`, as well as a public `getSource()` method.

The event tree's complexity is derived in part from Java evolving over time. Initially, Java just included the *Abstract Windows Toolkit (AWT)* for GUI development. `AWTEvent` is a direct subclass of `EventObject`. All events that work with the AWT GUI classes inherit from `AWTEvent`. When Swing was added to Java, newer Swing event classes were added into the hierarchy. Many were added as subclasses to `AWTEvent`. To make matters a bit more confusing, other Swing event classes inherit directly from `EventObject`.

AWT Event Classes

All our Java GUI programs used events that inherited from Java's `AWTEvent` class.

`AWTEvent`s augment `EventObject`'s data with an integer event ID that is used to speed searching the `listenerList` for events of a particular type, as well as numerous static constants and specialized fields.

There are many subclasses to `AWTEvent`. These classes store the information needed to describe the specific type of event. The instance methods are primarily "getters" for the values. Because events may be processed by multiple handlers, it is a bad idea to change event attributes during processing. Therefore, most event objects are *immutable*. They do not include "setters" that would allow us to change attributes.

Take, for example, `MouseEvent`. `MouseEvent`s occur when a button is pressed, released, or clicked; when a component was entered or exited; or when the mouse wheel was moved. The `MouseEvent` class implements the following methods:

- `getButton()` which returns an integer telling which, if any, button was pressed.
- `getX()` which returns the X location (in the source's coordinate system) where the event occurred.
- `getY()` which returns the Y location (in the source's coordinate system) where the event occurred.
- Several other getters used in various situations.

4.4.3 Java Event Processing

The steps Java takes to process an event depends on the event's location in the class hierarchy.

Dispatching AWTEvents

Java's virtual machine uses a separate thread, known as the *event-dispatching thread*, to process `AWTEvent`s. In GUI programs the thread is typically started when `setVisible(true)` is called, but the thread may also be started in a number of other ways. All `AWTEvent`s are processed by this

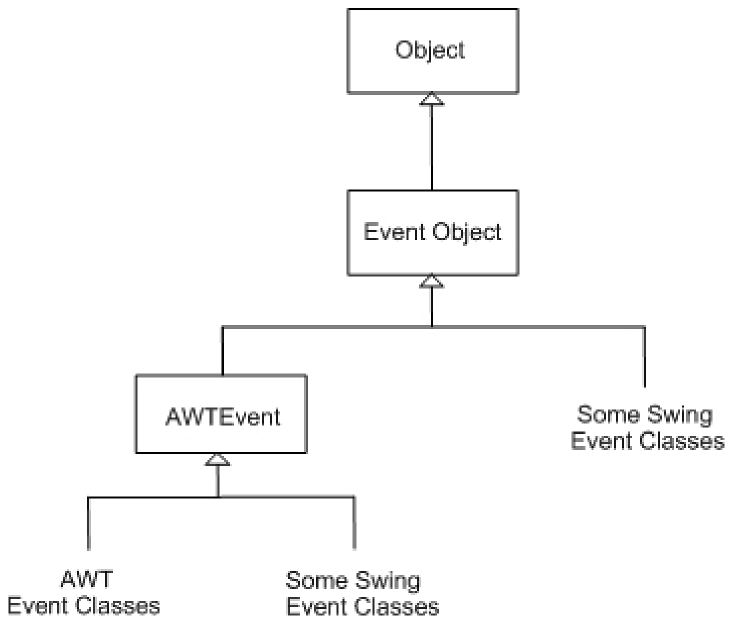


Figure 4.2: Overview of the Java event class hierarchy.

thread. The event-dispatching thread handles events in a first-fired first handled order, see Figure 4.3.

Thread Safe Components

As discussed earlier, if a handler takes too much time, the GUI will appear unresponsive and the programmer should consider using a separate thread for that particular handler.

By contrast, some handler code may be required to execute only in the event-dispatching thread, such as handlers that access GUI components. For example, a textfield may have its text reset by a handler, or a component may have its background color changed. These types of activities need to be done within the event-dispatching thread, as Java's Swing classes were designed using the *single thread rule*. Simply stated, once a Swing component is visible, only the event-dispatching thread should access it. A few of a Swing component's methods are considered *thread safe*, like adding and removing listeners and `repaint()`, and are exempt from the rule. All other component method calls should follow the single thread rule.

This leaves us with a bit of an awkward situation: If you have a long running handler, it should execute in its own thread, but if it needs to access its source component, it should run in the event-dispatching thread. Java provides a work around. The programmer can specify that code be executed in the event-dispatching thread using the `SwingUtilities` class methods `invokeLater()` and `invokeAndWait()`.

Processing AWT Events

When an `AWTEvent` fires, the source creates an event object and enqueues it in the system event queue. The event queue is an instance of `EventQueue` and exists as an object in the Java virtual machine. It is possible to obtain a reference to the queue by calling the default `Toolkit` object's

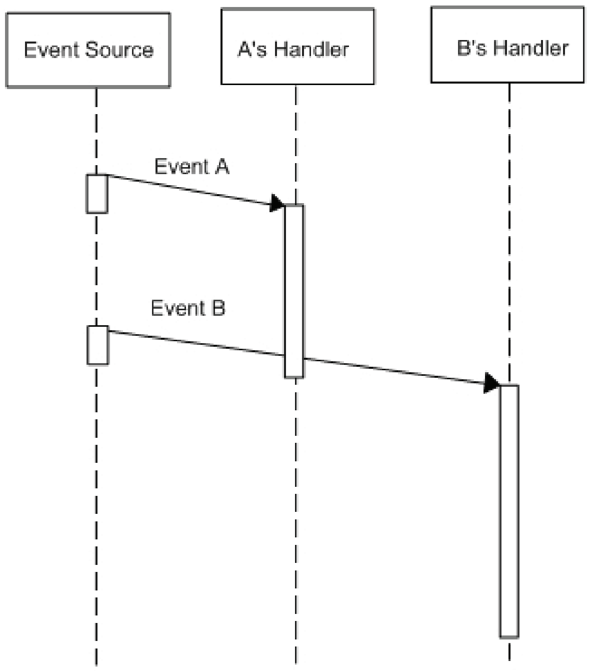


Figure 4.3: Java's AWT event model requires that events are handled sequentially, in the order they are fired. Thus, *Event A*'s handling completes before *Event B*'s handling begins.

`getSystemEventQueue()` method. The reference is obviously necessary if we want to explicitly place events into the queue. It is also possible to replace the default event queue with one of your own devising using the event queue's `push()` method. This is useful if you want to instrument the event queue to log the events as they take place, i.e. for debugging or testing purposes, or if you want to change the semantics of the event queue, such as implementing a priority hierarchy of events for a particular application.

The event-dispatching thread is responsible for processing `AWTEvent`s after they arrive in the queue. The processing includes:

1. Get the next event from the queue by calling the queue's `getNextEvent()` method.

2. Pass the event along as an argument to the queue's `dispatchEvent()` method. `dispatchEvent()` behaves as follows:

   (a) If the event implements the `ActiveEvent` interface, it calls the event's `dispatch()` method. Java uses the `ActiveEvent` interface for events that know how to process themselves. As we will see in the last section of the chapter, `ActiveEvent`s are a good way to develop our own event classes.

   (b) Otherwise, if the event source is a `Component`, it calls the `Component`'s `dispatchEvent()` method, again passing along the event as a parameter. Note that Swing components inherit from `JComponent` which in turn inherits from AWT's `Component`, so this is the path followed when processing most GUI events. `Component`'s `dispatchEvent()` runs through a variety of cases, handling both *low-level* and *high-level* events.

Low-Level and High-Level Events

Low-level events include mouse events, keyboard events and windowing events, like resizing the window. These events are created at the operating system level and passed into our application for handling.

By contrast, *high-level* events are those that aren't tied to any particular device. For example, `ActionEvents` are high level, as they may be fired by a mouse click, or a key press, choosing a menu item. You may think of a high-level event as more loosely coupled to hardware and more tightly coupled to the application than low-level events.

Some low-level and high-level event types are paired. For example, clicking the left mouse button may be both a `MouseEvent` (a low-level event) and an `ActionEvent` (a high-level event). In Java, if both a high-level handler and a low-level handler are registered for an event, the low-level handler is ignored.

Since there is a small fixed number of low-level event types, `dispatchEvent()` handles them directly.

For high-level events, the source's `dispatchEvent()` calls its `processEvent()` method, again passing along the event object. `processEvent()` checks if any event handlers have been registered for this type of event. If so, it calls each handler in turn.

Method Calls vs. Events Revisited

For many events, `processEvent()` could have been called directly by the source when the event fired, but wasn't. Instead, the event is placed in the event queue and the event dispatching thread ends up invoking `processEvent()`. We gain two things by having `processEvent()` called via this more indirect route. First, the processing is now done asynchronously in the event-dispatching thread, freeing up the GUI thread for other work; and second, events are processed sequentially, in the order they are fired. Both of these provide further decoupling between the event source and its handlers.

We also note that there is only one `processEvent()` method in any component, even if the component fires many different types of events. It is up to `processEvent()` to sort out what type of event has fired and process it appropriately. `Component`'s `processEvent()` may be overridden, but one should be *VERY* careful doing so, to guarantee that all events are processed appropriately.

(c) Otherwise, handle AWT menu events. Note that Swing does not share these menu classes.

(d) Otherwise, ignore the event.

3. Repeat the process for the next event.

Processing Beans-Like Events

The event processing discussed in the previous section works well. It was introduced in Java Version 1.1 and has changed little since. There are a couple of drawbacks to this approach, however.

First, developing *test harnesses* for the events is difficult. A test harness is a collection of code that tests the various parts of our system. Testing should not be done haphazardly, but should be repeatable. A test harness should be able to automatically reproduce the same sequence of events each time the test is run. Unfortunately, with a few exceptions, it is not possible to explicitly fire `AWTEvents` from within code. The firing of events – creating and enqueuing of the event objects – is done by the event sources, but the sources do not contain public methods that allow the tester to recreate the process. For example, there is no `fireMouseClickedEvent()` method available. The easiest way to test the code is to physically click the mouse. This problem was partially addressed by the introduction of the `Robot` class in Java 1.3. The `Robot` class was specifically designed to generate low-level events and pass them on to the application for testing purposes.

A second problem is that it takes detailed knowledge of the event queue and the `AWTEvent` class to create new `AWTEvent` subclasses. For example, developing new event sources requires the programmer to obtain a reference to the event queue and add events to it. She must also provide processing methods that handle all event types that the new sources may fire.

Event classes that inherit directly from `EventObject` avoid these problems. They are are coupled with event sources that process the events directly, avoiding the need for the event queue. This is the approach taken by Java Beans, Java's component architecture [Englander, 1997]. Each event source contains `fire***Event()` for each type of event it may fire. For example, `JMenu`s work with `MenuEvent`s. `MenuEvent`s include such things as selecting a menu, and canceling a menu[4]. The `MenuEvent` class inherits from `EventObject`, so the `JMenu` class contains the following methods:

- `fireMenuSelected()`
- `fireMenuDeselected()`, and
- `fireMenuCancelled()`

Each of these methods creates an immutable `MenuEvent` object, and passes that object to each of the registered `MenuListener`s. When processing the list of listeners, the list must be cloned, so that if a handler adds or removes listeners, it will not affect which handlers are called for this event, only future events.

Non-`AWTEvent`s are handled within the thread where they originate. This means that cascading events (events firing other events) are processed in a way analogous to procedure calls. If an event handler fires a second event, the second event is processed to completion before the thread of execution returns control to the original handler, see Figure 4.4.

4.5 Programmer Defined AWT Events

One way to get a better grasp on all the ideas in this chapter is to look at an example. In this section we will develop an example using the `AWTEvent` classes. In the next section we will contrast this implementation with a second one developed using a more beans like approach.

Only code snippets are presented here, but the entire listing is available on the text's web site.

> *Patients in a hospital may be hooked up to numerous different types of monitors. The output from these monitors is displayed at the nursing station at the end of the hall. This allows one or two nurses to care for an entire wing of patients. If everything remains within normal ranges, a display shows the monitor's values. If some value exceeds a threshold, warnings are generated, alerting the nurses that a patient needs attention.*
>
> *As the heart pumps, a patient's blood pressure goes up and down with each beat. The higher pressure is known as the* systolic pressure*. The lower pressure is the* diastolic pressure*. Normal values are 120 and 80 millimeters of mercury, respectively. In this example, we will design a system that will sound a warning if the systolic pressure exceeds 140 or goes below 90.*

Figures 4.5 and 4.6 show the simulated patient GUI and monitor windows, respectively.

[4]Note that selecting menu items fires `ActionEvents`, not `MenuEvents`.

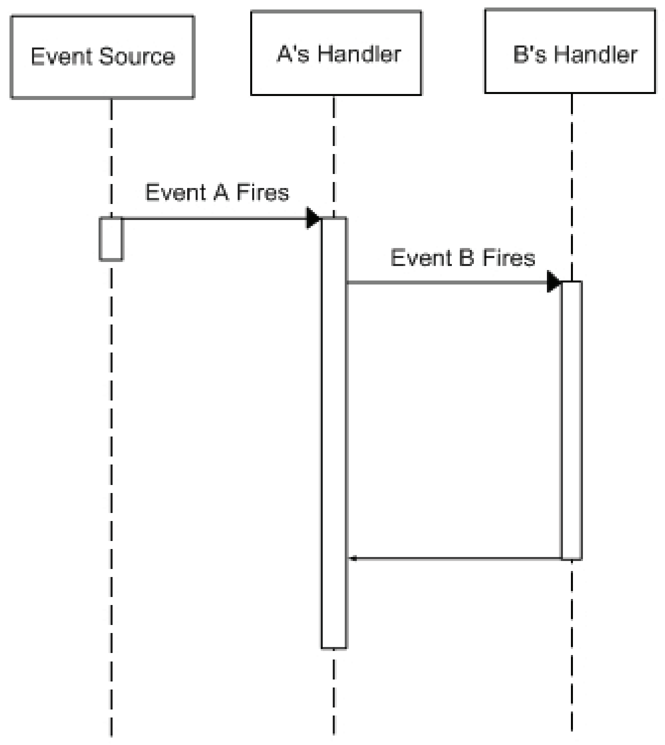


Figure 4.4: Java events that do not inherit from `AWTEvent` are handled in a depth first fashion, much like procedure calls.

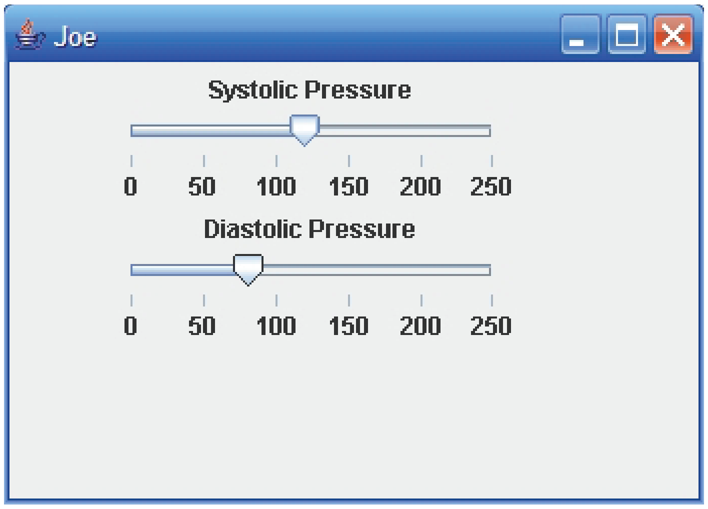


Figure 4.5: The GUI interface for a simulated patient. The patient's blood pressure values are changed by moving the sliders.

There are really three different, but closely related events being used in this example. Any change to the systolic or diastolic pressure will fire an event that updates the display at the nurse's station. If the systolic pressure goes too high or too low, a warning event (different than an ordinary systolic event) is fired to alert the nurses that a patient needs attention. We could define three different event classes: a `SystolicEvent`, a `DiastolicEvent`, and a `WarningEvent`. Very often, however, a handler interested in one of these events will be interested in multiple of them, so it make sense to define only one type of event, a `BloodPressureEvent`

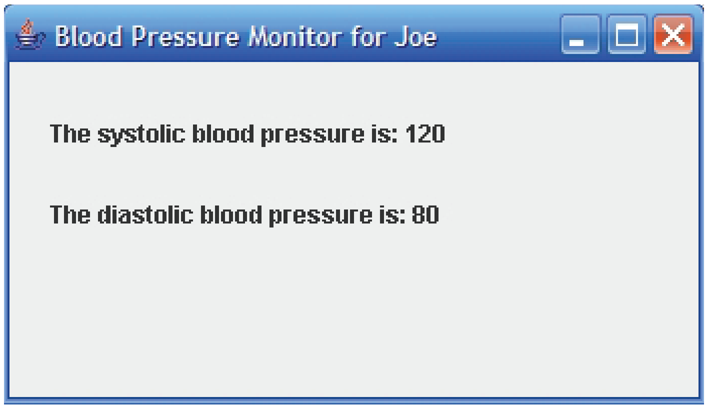
4.5.1 New Event Classes and Interfaces

Central to this example is the need for a new type of event, a `BloodPressureEvent`. As we saw illustrated in Chapter 2, there are really a collection of classes and interfaces needed for each type of event. These include:

- The `BloodPressureListener` interface which defines the methods the event handlers must implement. These include methods for each sub-type of event. We will need methods declared to handle: changes to the systolic pressure, changes to the diastolic pressure, and warning events – where the systolic pressure goes out of range. We do not include the actual event handlers here, since application programmers will define the event handlers, specifying application-specific actions to take when a `BloodPressureEvent` fires.
- The `BloodPressureEvent` class which encapsulates all the critical data about the event. A `BloodPressureEvent` must contain, the event source, the changed value, and possibly flag telling whether the event is for the systolic or diastolic pressure, or a warning.
- The `Patient`, the event source, which fires the events.

4.5.2 The `BloodPressureListener` Interface

The `BloodPressureListener` interface specifies the methods that each handler must implement. We have three very closely related types of events that are being handled: changes to the systolic



The systolic blood pressure is: 120

The diastolic blood pressure is: 80

Figure 4.6: The blood pressure monitoring display at the nurse's station. When a warning occurs, a message box pops up, as well.

pressure, changes to the diastolic pressure, and warnings. Because there are three closely related types of events, the interface declares three methods: `systolicChange()`, `diastolicChange()`, and `warning()`. `BloodPressureListener` is given in its entirety below.

```java
/**
 * This interface defines the methods that must be provided by
 * all objects receiving BloodPressure events.
 *
 * @author Stuart Hansen
 */
public interface BloodPressureListener extends EventListener {
    // receives the event if the upper value (systolic) changes
    public void systolicChange(BloodPressureEvent e);

    // receives the event if the lower value (diastolic) changes
    public void diastolicChange(BloodPressureEvent e);

    // receives the event if the systolic pressure goes extreme
    public void warning(BloodPressureEvent e);
}
```

4.5.3 The `BloodPressureEvent` class

Each of the methods in the listener interface takes a `BloodPressureEvent` as a parameter. A `BloodPressureEvent` object contains the following information:

- the event source, in this case the patient.
- `AWTEvents` include an integer event type. We will use this type field to keep track of whether we have a systolic, diastolic or warning event.

  These first two variables are inherited from `AWTEvent` and their values are set in the call to `super()`.
- the new value.

Generally speaking, AWTEvent objects are passive. They contain information used by handlers, but don't contain much active code. However, events are queued in the system event queue. To process the event, the system needs to go back to the source which contains the handlers. The easiest way to do this is to have the event class implement the ActiveEvent interface, which includes only one method, dispatch(). As discussed in the previous section, the EventQueue's dispatchEvent() method checks to see if an event is an ActiveEvent event. If so, it calls the event's dispatch() method. In our BloodPressureEvent class, we will use the event's dispatch() method to call back to the Patient's handleBPEvent() method.

```java
import java.awt.*;

/**
 * This class encapsulates all the information for
 * when the blood pressure changes
 *
 * @author Stuart Hansen
 */

public class BloodPressureEvent extends AWTEvent implements ActiveEvent {
    // We use these constants to sort out what type of Blood Pressure
    // event occurred
    public static final int SYSTOLIC_EVENT = 0;
    public static final int DIASTOLIC_EVENT = 1;
    public static final int WARNING_EVENT = 2;

    private int value;  // the new value

    // This is the only constructor.
    public BloodPressureEvent (Object source, int type, int newValue) {
        super(source, type);
        this.value = newValue;
    }

    // returns the value
    public int getValue()  {
        return value;
    }

    // Handle the event by dispatching it to the source.
    public void dispatch() {
        BloodPressureEvent event = (BloodPressureEvent) this;
        Patient pat = (Patient)getSource();
        pat.handleBPEvent(event);
    }
}
```

The dispatch() method is short enough to be very easy to read. It first finds the patient that is the source of this event. It then calls the patient's handleBPEvent() method.

The need for dispatch() and handleBPEvent() may seem confusing. Conceptually it isn't too hard to explain, however. Java's event infrastructure knows how to handle all the Java defined event types. ActionEvents, mouse clicks and the like are understood by the EventQueue's dispatchEvent() method. When dispatchEvent() comes to a BloodPressureEvent it doesn't know what to do with it, however. Making our events `ActiveEvent`s solves the problem.

4.5.4 The `Patient` class

The `Patient` class is our event source. The patient is a fairly typical class that stores data about the patient. In addition, because it is an event source, it has methods to add and remove listeners, mechanisms to fire events, and methods to help process events.

We declare and instantiate a vector of listeners to hold our handlers:

```java
    private Vector<BloodPressureListener> bpListeners =
                                        new Vector<BloodPressureListener>();
```

We declare two short methods to add and remove handlers from the vector:

```java
   // add a blood pressure listener to this patient
   public void addBloodPressureListener (BloodPressureListener bpListener) {
       bpListeners.add(bpListener);
   }

   // remove a blood pressure listener from this patient
   public void removeBloodPressureListener (BloodPressureListener bpListener) {
       bpListeners.remove(bpListener);
   }
```

Events are fired when values are blood pressure values are changes. `setDiastolic()` calls `fireBPEvent()` which enqueues the event in the system event queue.

```java
    // set the current diastolic blood pressure
    public void setDiastolic (int diastolic) {
        int olddiastolic = this.diastolic;
        this.diastolic = diastolic;
        if (olddiastolic != diastolic) {
            fireBPEvent(new BloodPressureEvent (
                           this,
                           BloodPressureEvent.DIASTOLIC_EVENT,
                           diastolic));
        }
    }

    // Firing the event requires enqueuing it
    public void fireBPEvent (BloodPressureEvent e) {
        Toolkit kit = java.awt.Toolkit.getDefaultToolkit();
        EventQueue queue = kit.getSystemEventQueue();
        queue.postEvent(e);
    }
```

`BloodPressureEvent`'s `dispatch()` method calls the `Patient`'s `handleBPEvent()` method, so this also needs to be defined. Since this method handles all types of blood pressure events, we use a switch statement to distinguish them.

```
    // Handle the blood pressure event when it comes out of the queue
    public void handleBPEvent (BloodPressureEvent e) {
        // clone the vector so registrations don't affect this event instance
        Vector<BloodPressureListener> temp =
                    (Vector<BloodPressureListener>) bpListeners.clone();

        // Walk through the vector firing events to each listener
        Iterator<BloodPressureListener> bplIterator = temp.iterator();
        while (bplIterator.hasNext()) {
            // The switch is needed to sort through the three types of
            // blood pressure events
            BloodPressureListener bpl = bplIterator.next();
            switch (e.getID()) {
                case BloodPressureEvent.DIASTOLIC_EVENT:
                    bpl.diastolicChange(e);
                    break;
                case BloodPressureEvent.SYSTOLIC_EVENT:
                    bpl.systolicChange(e);
                    break;
                case BloodPressureEvent.WARNING_EVENT:
                    bpl.warning(e);
            }
        }
    }
```

4.6 The Beans-Like Implementation

The same blood pressure example can be written using beans-like events that avoid the AWT event queue. This is a worthwhile exercise, if for no other reason to gain insight into the the way the two systems work.

The main difference is that the event source now takes responsibility for processing events rather than relying on the system's event queue.

The `BloodPressureListener` interface remains unchanged from the previous example.

4.6.1 Revamped Blood Pressure Events

The `BloodPressureEvent` class is simpler than in the previous section.

- We no longer need the event type field to distinguish which type of event we are processing. Since the patient will be calling the handlers directly, and the patient already knows the event type, the information is redundant.

- The `dispatch()` method has disappeared. The `dispatch()` method provided a route back from the event queue to the source. Since we are no longer using the event queue, the method is not needed.

```java
import java.util.*;

/**
 * This class encapsulates all the information for
 * when the blood pressure changes
 *
 * @author Stuart Hansen
 */

public class BloodPressureEvent extends EventObject  {
    private int value;  // the new value

    // This is the only constructor.
    public BloodPressureEvent (Object source,  int newValue) {
        super(source);
        this.value = newValue;
    }

    // returns the value
    public int getValue() {
        return value;
    }
}
```

4.6.2 The Revamped Patient Class

In this implementation, the `Patient` class takes full responsibility for invoking the event handlers. The `Patient` class includes methods to fire each of the different types of blood pressure events. The fire methods are called from within the `Patient`'s setter methods. Each setter checks to see if an event has occurred, and if so, it calls the appropriate `fire***Event()`. Below is `setDiastolic()`. `setSystolic()` is very similar, but may fire either warning or systolic change events.

```java
    // set the current diastolic blood pressure
    public void setDiastolic (int diastolic) {
        int olddiastolic = this.diastolic;
        this.diastolic = diastolic;
        if (olddiastolic != diastolic) {
            fireDiastolicEvent(new BloodPressureEvent (
                                this,
                                diastolic));
        }
    }
```

Note that the code calls `fireDiastolicEvent()` only if there is a change in the value. This makes things slightly more efficient, since it avoids events that have no affect.

Below is the `fireDiastolicEvent()` method. It iterates across the list of listeners calling `diastolicChange()` on each.

```
    // We call each of the handlers directly
    public void fireDiastolicEvent (BloodPressureEvent e) {
        // clone the vector so registrations don't affect this event instance
        Vector<BloodPressureListener> temp =
                   (Vector<BloodPressureListener>) bpListeners.clone();

        // Walk through the vector firing events to each listener
        Iterator<BloodPressureListener> bplIterator = temp.iterator();
        while (bplIterator.hasNext()) {
            BloodPressureListener bpl = bplIterator.next();
            bpl.diastolicChange(e);
        }
    }
```

`fireSystolicEvent()` and `fireWarning()` are almost identical. They only vary in their names and in the listener method called.

4.6.3 Infrastructure or Client Code

The previous two sections have emphasized the differences in two Java based approaches to event processing. Both were really dealing with event infrastructure issues, however.

It is important to realize that client code is oblivious to these differences. The GUI and the event handlers are the same in either case. A programmer can use either set of Blood Pressure Event classes to implement a nurse's station.

4.7 Summary

As we saw in earlier chapters, it is possible to write event based applications with only a naive understanding of how events are processed. In this chapter we showed that an in-depth understanding of the event processing infrastructure is necessary. Making simplifying assumptions about event processing is fraught with danger, as minor differences in the processing can yield significant differences in the results.

We also showed how to use our understanding of the event processing infrastructure to develop our own event classes.