Event-Driven Programming

Chapter Overview

- How do we design an entity to emphasize its responses to various events?

In previous chapters, we have seen how an animate object can use its explicit control loop as a dispatcher, calling appropriate methods depending on what input it receives. In this chapter, we discuss a style of programming that shifts the emphasis from the dispatcher to the various handler methods called by that control loop. Entities designed in this way highlight their responses to a variety of situations, now called events. An implicit -- behind-the-scenes -- control loop dispatches to these event handler methods.

This event-driven style of programming is very commonly used in graphical user interfaces (GUIs). In Java, AWT's paint methods are an example of this kind of event-driven programming. This chapter closes with an exploration of a portion of the `java.awt` package, including `java.awt.Component` and its subclasses, to illustrate the structure of programs written in an event-driven style.

Objectives of this Chapter

1. To recognize event-driven control
2. To understand that event handlers describe responses to events, not their causes
3. To be able to write event handlers for simple event-driven systems

Control Loops and Handler Methods

In chapter 11, we looked at mechanisms for explicit dispatch. In that chapter, the job of the central control loop was to decide what needs to be done and then to call a helper procedure to do it. In this way, a single control loop can handle a variety of different inputs or circumstances. We saw, for example, how a calculator might respond differently to a digit, an operation, or another button such as =. The calculator's central control loop acts as a manager, routing work to the appropriate procedures. The actual work is accomplished by these helpers, or handler methods.

In this chapter, we will look at the same kind of architecture from a different viewpoint. Instead of focusing on the central control loop's role as a dispatcher, we will take that function largely for granted and look instead at control from the perspective of the handler methods. In other words, we will explore how one writes handlers for special circumstances, assuming that these handler methods will be called when they are needed. By the end of this chapter, we will turn to a system in which this is true without programmer effort, i.e., in which Java takes responsibility for ensuring that the handler methods are called when they are needed.
The basic idea of **event-driven programming** is simply to create objects with methods that handle the appropriate events or circumstances, without explicit attention to how or when these methods will be called. These helper methods provide answers to questions of the form, "What should I do when xxx happens?" Because xxx is a "thing that happens", or an **event**, these methods are sometimes called **event handlers**. As the writer of event handler methods, you expect that the event handlers will somehow (automatically) be invoked whenever the appropriate thing needs dealing with, i.e., whenever the appropriate event arises.[Footnote: Ensuring that those event handler methods will be called is a precondition for event-driven programming, not a part of it. We will return to the question of precisely how this can be accomplished later in this chapter.]

The result of this transformation is that your code focuses on the occasions when something of interest happens -- instead of the times when nothing much is going on -- and on how it should respond to these circumstances. An event is, after all, simply something (significant) that happens. This style of programming is called event-driven because the methods that you write -- the event handlers -- are the instructions for how to respond to events. The dispatchers -- whether central control loop or otherwise -- is a part of the background; the event handlers drive the code.

**Dispatch Revisited**

Consider the case of an Alarm, such as might be part of an AlarmClock system. The Alarm receives two kinds of signals: SIGNAL_TIMEOUT, which indicates that it is time for the Alarm to start ringing, and SIGNAL_RESET, which indicates that it is time for the Alarm to stop. We might implement this using two methods, handleTimeout() and handleReset().

```java
public class Alarm
{
    Buzzer bzzz = new Buzzer();

    public void handleTimeout()
    {
        this.bzzz.startRinging();
    }

    public void handleReset()
    {
        this.bzzz.stopRinging();
    }
}
```

How do these methods get called? In a traditional control loop architecture, this might be accomplished using a dispatch loop. For example, we might make Alarm an Animate and give it its own AnimatorThread. The job of the dispatch loop would be to wait for and processes incoming

---

(timeout and reset) signals. This AnimateAlarm's act() method might say:

```java
public class AnimateAlarm extends AnimateObject {
    Buzzer bzzz = new Buzzer();
    public void handleTimeout()
    {
        this.bzzz.startRinging();
    }
    public void handleReset()
    {
        this.bzzz.stopRinging();
    }
    public void act()
    {
        int signal = getNextSignal();
        switch (signal)
        {
            case SIGNAL_TIMEOUT:
                this.handleTimeout();
                break;
            case SIGNAL_RESET:
                this.handleReset();
                break;
            // Maybe other signals, too....
        }
    }
}
```

Of course, the real work is still done by the handleTimeout() and handleReset() methods. The job of the dispatch loop (or other calling code) is simply to decide which helper (handler) method needs to be called. The dispatcher -- this act() method -- is only there to make sure that handleTimeout() and handleReset() are called appropriately.
Simple Event Handling

What would happen if we shifted the focus to the helper procedures? What if we made the dispatch code invisible? Imagine writing code (such as this Alarm) in which you could be sure that the helper methods would be called automatically whenever the appropriate condition arose. In the case of the Alarm, we would not have to write the act method or switch statement above at all. We would simply equip our Alarm with the appropriate helper methods -- handleTimeout() and handleReset() -- and then make sure that the notifier mechanism knew to call these methods when the appropriate circumstances arose. This is precisely what event-driven programming does.

A Handler Interface

We have said that event-driven programming is a style of programming in which your code provides event handlers and some (as yet unexplained) event dispatcher invokes these event handler methods at the appropriate time. This means that the event dispatcher and the object with the event handler methods will need a way to communicate. To specify the contract between the event dispatcher and the event handler, we generally use an interface specifying the signatures of the event handler methods. This way, the event dispatcher doesn't need to know anything about the event handlers except that they exist and satisfy the appropriate contract.

In the case of the alarm, this interface might specify the two methods we've described, handleTimeout() and handleReset():

```java
public interface TimeoutResettable {
    public abstract void handleTimeout();
    public abstract void handleReset();
}
```

Of course, we'll have to modify our definition of Alarm to say that it implements TimeoutResettable:

```java
public class Alarm implements TimeoutResettable {
    Buzzer bzzz = new Buzzer();
}
```

Figure #. An Alarm that handles two event types.
public void handleTimeout()
{
    this.bzzz.startRinging();
}

public void handleReset()
{
    this.bzzz.stopRinging();
}

Note that this is a modification of our original Alarm, not of the AnimateAlarm class. The
TimeoutResettable Alarm need not be Animate. In fact, if it is truly event-driven, it will not be.

This TimeoutResettable Alarm definition works as long as some mechanism -- which we will not
worry about just yet -- takes responsibility for dispatching handleTimeout() and handleReset()
calls as appropriate. That dispatcher mechanism can rely on the fact that our Alarm is a
TimeoutResettable, i.e., that it provides implementations for these methods. The dispatcher that
invokes handleTimeout() and handleReset() need not know anything about the Alarm other than
that it is a TimeoutResettable.

An Unrealistic Dispatcher

How might our TimeoutResettable Alarm be invoked? There are many answers, and we will see a
few later. For now, though, it is worth looking at one simple answer to get the sense that this really
can be done.

A simple -- and not very realistic -- event dispatcher might look a lot like the act method of
AnimateAlarm. To make it more generic, we will separate that method and encapsulate it inside its
own object. We will also give that object access to its event handler using the TimeoutResettable
interface. Major differences between this code and AnimateAlarm are highlighted. Of course, the
dispatcher doesn't have its own handler methods; its constructor requires a TimeoutResettable to
provide those.

    public class TimeoutResetDispatcher extends AnimateObject
    {
        private TimeoutResettable eventHandler;
        
        public TimeoutResetDispatcher( TimeoutResettable eventHandler )
        {
            this.eventHandler = eventHandler;
        }

        public void act()
        {
            int signal = getNextSignal();
            switch (signal)
            {
                case SIGNAL_TIMEOUT:
                    this.eventHandler.handleTimeout();
                    break;
                case SIGNAL_RESET:
                    this.eventHandler.handleReset();
                    break;
            }
The details of this dispatcher are rather unrealistic. For one thing, it is extremely specific to the type of event, and extremely general to its event handler dispatchees. More importantly, in event-driven programming it is quite common not to actually see the dispatcher.

But dispatchers in real event-driven programs play the same role that this piece of code does in many ways. For example, the dispatcher doesn't know much about the object that will actually be handling the events, beyond the fact that it implements the specified event-handling contract. This dispatcher can invoke `handleTimeout()` and `handleReset()` methods for any `TimeoutResettable`, provided that the appropriate `TimeoutResettable` is provided at construction time. Different dispatchers might dispatch to different `Alarms`. In fact, timeout and reset are sufficiently general events that other types of objects might rely on them.

**Sharing the Interface**

Another object that might be an event-driven user of timeouts and resets -- and be controlled by the `TimeoutResetDispatcher` -- is an image animation. An image animation is a series of images, displayed one after the other, that give the impression of motion. In this case, we use the timeout event to cause the next image to be displayed, while reset restores the image sequence to the beginning. `ImageAnimation` simply provides implementations of these methods without worrying about how or when they will be invoked.

```java
public class ImageAnimation implements TimeoutResettable {
    private Image[] frames;
    private int currentFrameIndex = 0;

    // To be continued...

    public void handleTimeout() {
        // change current image
        currentFrameIndex = (currentFrameIndex + 1) % frames.length;
        repaint();
    }

    public void handleReset() {
        currentFrameIndex = 0;
        repaint();
    }
}
```

The image array `frames` will hold the sequence of images to be displayed during the animation. When the `ImageAnimation` is asked to paint (or display) itself, it will draw the `Image` labeled by `this.frames[this.currentFrameIndex]`. By changing `this.currentFrameIndex`, we can change what is currently displayed. When we do change `this.currentFrameIndex`, we can make that change apparent by invoking the `ImageAnimation`'s `repaint()` method, which causes the `ImageAnimation` to display the image associated with `this.frames[this.currentFrameIndex]`.

We omit the setup code that loads the Images into `frames` and handles other construction details.

The next segment of code is the timeout event handler, the helper method that is called when a timeout occurs. What should the `ImageAnimation` do when a timeout is received? Note that the

Figure #. An ImageAnimation is a single component that displays a sequence of images, one at a time. For example, these frames, displayed in an ImageAnimation, would give the impression of a clock whose hands move.
question is not how to determine whether a timeout has occurred, but what to do when it has. This is the fundamental premise behind event-driven programming: the event handler method will be called when appropriate. The event handler simply provides the instructions for what to do when the event happens. When a timeout occurs, it is time to advance to the next frame of the animation:

```java
public void handleTimeout()
{
    if (this.currentFrameIndex < (this.frames.length - 1))
    {
        this.currentFrameIndex = this.currentFrameIndex + 1;
        this.repaint();
    }
}
```

This code checks to see whether there are any frames left. If the animation is already at the end of the sequence, the execution skips the if clause and -- since there is no else clause -- does nothing. Otherwise -- if there's a next frame -- the execution increments the current frame counter, setting up the next frame to be drawn. Then, it calls this.repaint(), the method that causes the ImageAnimation to be redrawn. Recall that the ImageAnimation paints itself using the image that is associated with this.frames[this.currentFrameIndex].

What about a reset? What should the ImageAnimation do when it receives the signal to reset? Handling a reset event is much like handling a timeout, but even simpler. The ImageAnimation simply returns to the first image in the sequence:

```java
public void handleReset()
{
    this.currentFrameIndex = 0;
    this.repaint();
}
```

No matter what, we reset the current frame index to 0, then repaint the image animation with the new frame. Note also that the next timeout will cause the frame to begin advancing again.

The code to actually repaint the image, which we have not shown here, makes this.frames [this.currentFrameIndex] appear. As a result, handleTimeout() works by changing the index to the next frame (until the end of the animation is reached); handleReset() restarts the image animation by restoring the index to the beginning index of this.frames once more.

Both Alarm and ImageAnimation are objects written in event-driven style. That is, they implement a contract that says "If you invoke my event handler method whenever the appropriate event arises, I will take care of responding to that event." Alternately, we think of the contract as saying "When the event in question happens, just let me know." When building both Alarm and ImageAnimation, the question to ask is, "What should I do when the specified event happens?"

**Real Event-Driven Programming**

We have seen other examples of event-driven coding style. In this section, we briefly review these and recast them in light of event-driven programming's central question, "What should I do when xxx happens?" After reviewing these examples, we turn to look at the relationship of event providers to event handlers.

**Previous examples**

In chapter 9, we saw how an Animate's act() method is repeatedly invoked by an AnimatorThread.
This act() method is in effect an event handler. It answers the question, "What should I do when it is time for me to act?" The Animate doesn't know who is invoking its act() method or how that invoker decided that it was time to act. It simply knows that it is, and how to respond to that knowledge, i.e., how to act(). The act() method may be invoked by an AnimatorThread instruction follower, executing at the same time as other parts of the system. It might equally well be invoked by a TurnTakerAnimator that controls a group of Animates and gives one Animate at a time a turn to act(). This latter approach might make sense, for example, in a board game where each player could move only when it was that player's turn.

Similarly, we saw how a Runnable object has a run() method that can be invoked in an event-driven style. This is commonly done when the run() method is invoked by starting up a new Thread. In this case, the Runnable's run() method is invoked when the Thread is start()ed. From the perspective of the Runnable, its run() method is automatically invoked whenever it is time for the Runnable to "do its thing". In a self-animating object like a Clock, run() might be an event-handler-like method that is called by something "outside" (in this case, the Thread) when it is time for the Clock to begin execution.

The StringTransformer's transform methods of Interlude 1 were yet other examples of an event-driven style. These event handler methods simply answer the question, "What should I do when this StringTransformer is presented with a String to transform?" or "How do I respond to such a request?" These objects provide customized implementations for transforming strings. The decision of when to invoke these methods are outside the control of their owning objects.

In each of the cases described above, the event producer -- the thing that knows that it is time for a handler method to be invoked -- and the event handler -- which responds to the occurrence -- communicate fairly directly. For example, the TimeoutResetDispatcher polls (or explicitly asks) for signals and then directly invokes the event handler methods of its TimeoutResettable.

The Idea of an Event Queue

Event-driven programming by its very nature allows a more distant relationship between event producers and event consumers. Since the producer disavows responsibility for handling the event, it doesn't need to know or care who is taking on that responsibility; it merely needs to indicate that the event has arisen. The event handler doesn't really care where the event came from; it just needs to know that it will be invoked whenever the event has happened. This dissociation between event producers and event consumers is one of the potential benefits of programming in an event driven style.

Systems that take advantage of this opportunity to separate event producers from event handlers generally contain an additional component, called the event queue, that serves as an intermediary. It is important to understand how the event queue can be used and the role that it plays as an intermediary between event producers and event handlers. Unless you are building your own event-driven system from the ground up, it is not important that you be able to build it. Generally, an event queue is provided as a part of any event-based system, and the major event-based systems in Java are no exception.
The role of the event queue is to serve as a drop-off place for events that need to be handled, sort-of like a To Do list. When an object produces behavior that constitutes an event, it reports that event to the event queue, which holds on to the event. The report of the event may be as simple as an indication that something happened ("Timeout!") or as complex as a complete description of the state of the world at the time that the event happened (e.g., the complete Wall Street Journal report on the stock market crash). What is important is that the event queue stores (remembers) this event report.

In addition to receiving event reports, the event queue also has an active instruction-follower that removes an event (typically the oldest one) from the queue and notifies any interested event handler methods. This is the queue-checker/dispatcher. An event queue also needs some way to figure out who to notify when an event has happened. In the cases that we explore in this chapter, there is always a single event queue per hander object, so it is always that object to which events are reported. In the next chapter, we will discuss a system that allows finer-grained control.

Consider the TimeoutResettable event handlers described above. A timer might generate the timeout events and deposit them into the queue. It would then return to its own business, keeping time and paying no more attention to the event queue. A separate instruction follower, the event dispatcher, would discover the timeout event in the queue and invoke the handleTimeout() method of the relevant party. The structure of this "queue cleaner" would be very similar to the TimeoutResetDispatcher we saw above.

Properties of Event Queues

This mechanism allows for a separation between the event producer and the event handler. The instruction-follower that puts an event into the queue -- the one who generates the event -- is not necessarily the instruction follower who performs the handler method (i.e., handles the event). Instead, one or more dedicated instruction followers have the task of processing events deposited into the queue, invoking the event handler method(s) as needed. Event suppliers need to know only about the event queue, not about the event handler methods.

Note that it is the event queue dispatcher's Thread (or instruction follower) that actually executes the steps of the event handler method. (Method invocation does not change which instruction follower is executing.) As a result, when you are writing event handlers, it is important that the event handler code complete and return (relatively) quickly; for example, it should not go into an infinite loop. [Footnote: A Runnable's run() method is an exception to this, because the Thread that executes run() has nothing to go back to doing. When run() completes, the execution of that Thread stops.] If the event dispatcher invoked an event handler that did not return, the dispatcher would be unable to process other events waiting in the queue.

You will almost never have to deal with an event queue explicitly unless you write your own event-driven system from scratch. Most programmers who write event-driven programs do not ever touch the event queue that underlies their systems. Instead, like many other aspects of event-driven programming, event queueing is generally a part of the hidden behavior of a system. However, there's nothing particularly mysterious about it. An event queue's contract provides an enqueue (add to the queue) operation and a dispatcher that actually invokes the event handler methods. [Footnote: In the next chapter, we will see that some event queues also provide an event listener registry service. This is not necessary in the event systems of this chapter, where there is a single event queue per handler object, but provides yet another layer of flexibility.]

In Java, the graphical user interface toolkit provides an event queue to handle screen events such as mouse clicking and button pressing. That event queue is fairly well hidden under the abstractions of the toolkit, so that you may not realize that it is an event queue at all. In the next chapter, we will explore that more complex system, which is used for most events in Java's windowing toolkit. That
system decouples the event handler from the object to whom the event happens, allowing one object to provide the handler for another's significant events. This is known as event delegation.

**Graphical User Interfaces: An Extended Example**

So far, we have left open the question of where and how events get generated. This is because in the most common kind of event system that you are likely to encounter -- a windowing system for a graphical user interface -- you do not deal with event generation directly. Instead, Java takes care of notifying the appropriate objects that an event of interest has occurred. When you are writing graphical user interfaces in Java, you will write event handlers without ever having to worry about when, where, and how the appropriate events are produced.

Before we can begin to talk about event handling in graphical user interfaces, we need to look briefly at what graphical user interfaces are and how they are built in Java. A graphical user interface -- sometimes called a GUI, pronounced "gooey" -- is a visual display containing windows, buttons, text boxes, and other "widgets". It is common to interact with a graphical user interface using a mouse, though a keyboard is often a useful adjunct. Graphical user interfaces became the standard interface for personal computers in the 1980s, though they were invented much earlier.

![A sample graphical user interface.](http://www.cs101.org/ipij/events.html)

**java.awt**

Java provides a few different ways of making graphical user interfaces. In this section, we will take a look at the package java.awt. This package contains three major kinds of classes that are useful for making GUIs. The first of these is `java.awt.Component` and its subclasses. These are things that appear on your screen, like windows and buttons. The immediately following subsection explores this component hierarchy. The second major GUI class is the class `java.awt.Graphics`, which is involved in special kinds of drawing. We will return to `java.awt.Graphics` at the end of this chapter. The final group of classes are the event classes: the `java.awt.Event` class together with the classes in the `java.awt.Event` package. We'll come back and look at AWT Events in the event delegation chapter. Here we'll deal only with one (pseudo-)event, painting. In the remainder of this section, we are going to focus on Components and, in a bit, Graphics.

The event that we will be concerned with here is painting. That is, this is the event that occurs when a window or other user interface object becomes visible, is resized, or for other reasons needs to be redrawn. This event happens to a Component. In order to handle this event you need to know what the current state of the drawing is, including both its Coordinate system and what if anything is currently visible. That information is held by a Graphics. So when the event happens, it takes a form roughly paraphrased as "paint yourself on this screen". The event handler belongs to a Component -- the "self" to paint -- and it takes a single argument, a Graphics -- the "screen" on which to paint.
Components

A component is a thing that can appear on your screen, like a window or a button. The parent of all component classes is `java.awt.Component`. The `Component` class embodies a screen presence. You can't have a vanilla `Component`, though; you can only have an instance of one of its subclasses. (In fact, `java.awt.Component` is an abstract class. See the sidebar in Chapter 7 for further detail on abstract classes.)

Although you can't instantiate `Component` directly, `Component` has several useful subclasses. One group of these is the set of stand-alone widgets that let you interact with your screen in stereotyped ways. There are many GUI widgets built in to `java.awt`. These include `Checkbox`, `Choice`, `List`, `Button`, `Label`, and `Scrollbar`. In addition, there are several `Menu` variants that don't extend `Component` directly, but also provide useful widgets. Each of these widgets is pretty well able to handle its GUI behavior -- showing up, disappearing, allowing selections to be made, etc. In the event delegation chapter, we will see how to use these GUI components to allow the user to communicate with your application; for example, to have something smart happen when a selection is made. (This involves customizing these widgets' event handlers.)

Another set of components are called Containers. These `Components` extend `java.awt.Container` (which itself is an abstract class extending `java.awt.Component`). Containers are components that can have other components inside them. For example, a `java.awt.Window` (which is a kind of component) can have a `java.awt.Scrollbar`.

In this chapter, we will confine ourselves to one simple component behavior: painting itself. To do this, we will use a generic `Component`, called `Canvas`, that you can instantiate. The `java.awt.Canvas` class doesn't do anything special, but you can either use it as a generic component or extend it to get specialized behavior. We will make a `Canvas` that paints itself with a special picture.

![Figure #. Standard screen coordinates, showing the origin, directions of increasing horizontal (x) and vertical (y) coordinates, and two other sample points.](http://www.cs101.org/ipij/events.html)

Graphics

A `java.awt.Graphics` (sometimes called a "graphics context") is a special kind of object that knows how to make pictures appear. A `Graphics` uses a coordinate system to keep track of locations within it. The origin of this coordinate system -- the point (0,0) -- is in the upper left-hand corner. Moving right from this point involves increasing the first (x) coordinate, so (100, 0) is 100 pixels to the right of the origin, along the top edge of the `Graphics`. [Footnote: A pixel, short for picture element, is the smallest visible unit on your computer's screen. A higher resolution display is one that has more pixels in the same amount of space, i.e., one with smaller pixels. Java `Graphics` are delineated in pixels.] Moving down increases the second (y) coordinate, so (0,50) is 50 pixels below the top of the `Graphics`, along its left-hand side. (100,50) is a point that is not on either the top or left edge; it is 100 pixels to the right and 50 pixels down.
Each Graphics has methods such as drawLine, fillOval, and setColor that allow you to create pictures. For example, if you had a Graphics named g, g.fillOval(100,100,10,10) would make it display a 10-pixel by 10-pixel circle with its upper left-hand corner at position 100, 100. If you called g.setColor(Color.red) first, the circle would be red. A complete list of the methods of a java.awt.Graphics, together with a brief description of each, can be found in the java.awt.Graphics reference.

A Graphics is not the kind of object that you are likely to create or have hanging around. You will probably never run into the Graphics associated with GUI widgets or containers. However, each time that your Canvas needs to redisplay itself, it will be handed a Graphics context with which to do that redisplaying. So there will be times when your code will be given a Graphics to use.

The Story of paint

Painting (itself) is what a GUI component does when it becomes visible. For example, if a window is (partially) covering a component and then the window is moved, the component needs to make itself look right again. Java takes care of automatically determining that this should happen and asks the component to paint itself.

Every java.awt.Component has an event-driven paint method. This method does not say when the component should be painted, nor why, nor on what. This method has nothing to do with determining that painting is necessary. Instead, this method is the set of instructions that describe how to paint the Component. It is the answer to the question, "What should I do when it is time to paint myself (on the provided Graphics screen)?" It is the job of whatever calls the paint method to determine whether and when the Component needs to be painted.

The paint method of a Component is passed a Graphics object. This is the Graphics which contains, among other things, the coordinate frame within which drawing on this Component should take place. It also contains a variety of utilities that will make things actually appear within the Component. Just as you don't have to determine when or whether paint should be invoked, you don't need to provide the Graphics object. Like magic, when paint is invoked, the Graphics object will be there.

Each paint method contains the specific instructions that that component needs to make itself appear. For example, a Button's paint method makes the button label appear on the button. A Window's paint method not only makes the Window appear, it also makes sure that the paint method of each of the components contained in the Window gets called as well.

When the paint method is invoked, it is equipped with a single argument, a Graphics. If the Component does to display itself is, for example, to draw shapes, this Graphics (the argument passed in to the Component's paint method) is what actually does the drawing.

Your job, when implementing a paint method, is to make use of this provided Graphics (and any other information that the object may have) in order to make the correct picture appear. You supply the instructions to be executed. To paint me, make a big red dot. Or, to paint me, print my name. Or, to paint me, paint each of the Components that appear inside me.

Suppose that you want to have your Component contain a rectangle in the upper left-hand corner. A Graphics has a drawRect method which does just that. When your component's paint method is called, it should ask whatever Graphics object is supplied to it to drawRect(int x, int y, int width, int height). [Footnote: drawRect takes four arguments: the upper left hand coordinates and the size coordinates. All measurements are in pixels -- tiny boxes that make up your screen -- and the origin -- the point (0,0) -- is in the upper left-hand corner of the component. These are called "screen coordinates". Graphics objects have lots of other drawing methods, too. See the
For example, if paint were called with a Graphics named g, the instructions might read

```java
   g.fillRect(0,0,20,20);
```

to draw a square in the upper left-hand corner of the Component. The whole method would read

```java
   public void paint( Graphics g )
   {
      g.fillRect(0,0,20,20);
   }
```

A Component's paint method is an event handler. This means that the Component's paint method is the set of instructions describing the Component's response to a request to redisplay itself. It triggers whenever Java finds that something has happened that requires the component to redisplay itself.

**Painting on Demand**

When we say that paint is an event handler method, what we mean in part is that your code doesn't call paint directly. Instead, paint is called automatically by the Java runtime system any time the Component needs to redisplay itself. This could happen, for example, if a window were covered up and then uncovered: when the uncovering event occurs, the window needs to repaint itself. Each of the components, containers, and widgets in java.awt has an event-driven paint method. Note, however, that there's no Paintable interface; paint is a method of Component and is inherited by every class that extends Component.

The `paint` method takes a `Graphics` context as an argument. You cannot, in general, supply the appropriate `Graphics` context to a `Component`; but since you don't call paint, you don't need to supply the `Graphics`. Instead, Java's behind the scenes bookkeeping takes care of this. (Remember, `paint(Graphics g)` is used in **event-driven** style; that is, it is called by Java, not by your program.)

Your code cannot call paint directly. It is an event handler method and it uses an event queue; only the queue manager can call paint. But sometimes you will know that it is necessary for a GUI object to repaint itself. For example, in the code above the image animation needed to repaint itself each time the currentFrameIndex changed. Since you can't call the component's event handler directly, each Component provides another method, called `repaint()`, that you can call. If you call the component's `repaint()` method, it will ask Java to send it a new paint event.

If you do ever need to tell the system that you want your component to be painted, you need to arrange for Java to provide the appropriate information to your class. You can do this by calling the component's `repaint()` method. Unlike `paint`, which takes a `Graphics` as an argument, `repaint` takes no parameters. (This is good, because you don't generally have a `Graphics` around to give `paint`. This is another thing that Java keeps track of automatically.) You don't have to implement `repaint(); java.awt.Component.repaint();`, which you will inherit, queues up a new paint request (even supplying the appropriate `Graphics`) behind the scenes. Remember: **You never call paint, and you never implement repaint.** To cause a painting to happen, call `repaint()`; to explain how to paint your component, implement (override) the `paint(Graphics g)` method -- and don't worry about the Graphics, it will be automatically supplied to you!

**Events and Polymorphism**

One advantage of using an event-driven style is that your code can focus on how to respond to things that happen. It does not have to spend a lot of time figuring out whether things happen or deciding...
what has happened and who should deal with it. (Of course, event-driven code relies on an event dispatcher, which does have to deal with these things, but often either one is available -- as in the GUI case -- or a fairly simple and generic one will do.)

A second advantage of the event-driven style is that, when used in concert with an event queue (as in Java's AWT), it separates the generator of the event (e.g., the window motion) from the handler of the event (the component that is uncovered). This means that these two pieces of the system can be designed independently. All they have to do is to agree on the event protocol that they will use (in this case, repaint() and paint(Graphics g)). How each one fulfills its side of the contract -- how the component decides to paint itself, for example -- is something that the rest of the system doesn't have to worry about.

A corollary benefit, then, is that different kinds of components can handle the same event in very different ways. We saw this early in this chapter where the same pair of events -- timeout and reset -- were used to run both an alarm and an image animation. In these two objects, the timeout event meant very different things. The alarm handled a timeout by turning on its buzzer; the image animation switched to the next image each time a timeout occurred.

The GUI painting system that we have described uses this polymorphism to great advantage. When a component like a Canvas is asked to paint itself in a Graphics, it may draw a simple picture using the Graphics supplied. When a widget like a Button is asked to paint itself, it creates labeled region of the screen appropriate for clicking into. A Checkbox may paint itself as a square, with or without an X in it depending on whether the Checkbox is checked. A container such as a Window not only paints itself, it also asks each of the components contained inside it to paint themselves. The Window doesn't need to know anything about how these components appear; it simply asks them to paint themselves in the way that they know best.

---

**Chapter Summary**

- By hiding the central control loop, we shift emphasis from explicit dispatch to event handler methods.
- Event driven programming separates things that happen from how they're handled.
- Each object is free to implement the same event handler in a different, customized way.
- In Java's AWT, certain GUI events are automatically dispatched by the Java runtime.
- The root of the GUI component hierarchy is `java.awt.Component`. Although `java.awt.Component` is an abstract class, it has many useful subclasses, including
  - widgets such as Checkbox, Choice, List, Button, Label, and Scrollbar.
  - Containers, Components that can hold other Components.
  - Canvas, a generic Component that you can customize.
- Every `java.awt.Component` has a `paint(Graphics g)` method that is called by Java when Java needs to make the Component (re)appear on the screen.
  - By overriding or implementing a `paint(Graphics g)` method, you are describing how your custom component should handle requests to paint itself.
  - You don't generally call a component's `paint` method. (Among other things, you don't have a Graphics to pass it.) If you want to redraw a GUI component, you can call its `repaint()` method.

---

**Exercises**
1. Define a TimeoutResettable that simply prints to the Console whenever an event happens. The message printed should differ depending on which event occurs. Implement it in a purely event driven style, i.e., assuming that something else will manage the event dispatching.
2. Describe a scenario in which an event occurs to the object in the previous exercise. Explain the sequence of action.
3. Define a class that extends `java.awt.Canvas` and has an unfilled circle with its upper left hand corner at 100, 100.

   (Bonus) What happens if you make the Canvas very small? Can you modify your class to keep the circle centered on the Canvas? You can use Canvas's `getSize()` method, which returns a `java.awt.Dimension` with directly manipulable height and width fields.

4. Define a class that extends `java.awt.Canvas` and paints itself like a black-and-white checkerboard. You may assume that the dimensions of the Canvas are 400x400.

   (Bonus) Make the checkerboard red and black.

5. Define a class that extends `java.awt.Canvas` and has two different painting behaviors. (For example, it could paint a black circle or a red square.) This class should also have a changeMe method. Each time its changeMe method is called, it should redisplay itself using the other behavior (e.g., it should switch between a black circle and a red square). Hook this up to a Timer (from Chapter 8).

You can test your Canvases using the cs101.awt.DefaultFrame class included in the code supplement to this book.