Fun with Java: Frame Animation

Baldwin teaches you how to do frame animation. Equally important, he also teaches you about event-driven programming, multi-threaded programming, ordinary inner classes, anonymous inner classes, exception handling, and image icons.

Published: February 4, 2003
By Richard G. Baldwin

Java Programming Notes # 1470

- Preface
- Preview
- Discussion and Sample Code
- Run the Program
- Summary
- Complete Program Listing

Preface

Computer programming doesn't have to be a gut-wrenching, high-pressure activity. Programming can also be fun. For me, programs that provide sensory feedback, such as animation and sound, are particularly enjoyable.

My previous series on animation, which began with the lesson entitled Fun with Java: Sprite Animation, Part 1, taught you how to combine sprite animation and frame animation.

My current series of lessons, beginning with the lesson entitled Java Sound, An Introduction, will teach you how to program using the Java Sound API.

A somewhat simpler approach

Sometimes, the combination of sprite animation and frame animation is overkill, and frame animation is all that you need. Frame animation is generally the simpler of the two.

Page-flip animation

When I was a young boy, one of my friends had a book that had a small image at the lower right-hand corner of every right-hand page. By grasping the pages between the thumb and forefinger, and flipping through the pages rapidly, it was possible to animate the images to create a movie. I later learned how to do the same things by drawing stick men in the right-hand margin of every right-hand page in the telephone book. Then by flipping through the pages very quickly, I could cause the stick men to dance, turn flips, run back and forth, etc.

Little did I know as a youngster that someday I would be doing essentially the same thing, *(and also teaching others how to do it)* using a computer.

> *(In fact, at that point in my life, I had never heard of a computer, nor had anyone that I knew).*

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

Preview

In this lesson, I will teach you how to do frame animation, independent of sprite animation. Equally important, I will also use this opportunity to teach you about:

- Image icons
- Event-driven programming
- Multi-threaded programming
- Ordinary inner classes
- Anonymous inner classes
- Exception handling

Not trivial topics

None of the topics in the above list are trivial topics. In fact, with the exception of image icons and exception handling, these topics constitute a large part of what I teach in my college-level Intermediate Java Programming class.

> *(I cover exception handling in the introductory OOP course, and cover Image Icons in one of the advanced courses.)*

Of course, I cover each of those topics in much more depth in the classroom than I will be able to do in this single lesson. Therefore, as we go along, I will refer you to more detailed tutorial lessons on some of the topics as background material.

Discussion and Sample Code

Will discuss a sample program

As is my usual practice, I will provide this instruction by discussing a sample program. Although this program is rather short and compact, it is by no means simple. In fact, it incorporates some of the most subtle and abstract concepts of Java programming, including:

- Event-driven programming
- Multi-threaded programming
- Ordinary inner classes
- Anonymous inner classes

Will discuss in fragments

Also, as is my usual practice, I will discuss the program in fragments. *(A complete listing of the program is shown in Listing 15 near the end of the lesson.)*

This program, named **Animate04**, illustrates frame animation. When you compile and run the program, the graphical user interface *(GUI)* shown in Figure 1 appears on the screen.

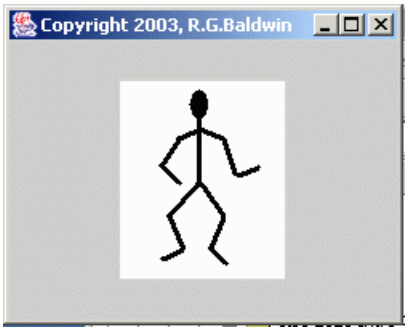


Figure 1 Program GUI

Each time the user points to the GUI with the mouse pointer, the stick man starts dancing and continues dancing until the mouse pointer exits the GUI. *(The program was tested using Sun's SDK version 1.4.1 under Win2000.)*

What is the animation process involved?

The animation process is essentially the same as the one I described above involving the telephone book. In this program, three images of the stick man are stored in the program. When the mouse pointer is inside the GUI, the program displays a series of images by cycling through the three images of the stick man. This creates the illusion that the stick man is moving.

(You could improve the quality of the animation considerably by creating additional image files showing intermediate positions for the stick man and causing the program to cycle through those images as well.)

The class definitions

This program consists of a single top-level outer class and two inner classes. Listing 1 shows the beginning of the top-level class, which is also the controlling class *(because it defines the **main** method).*

```
public class Animate04 extends JFrame{
  Thread animate;

  ImageIcon images[] = {//An array of
images
        new ImageIcon("java1470a.gif"),
    new ImageIcon("java1470b.gif"),
    new ImageIcon("java1470c.gif")};

  JLabel label = new
JLabel(images[0]);
```

Listing 1

Three instance variables are declared in Listing 1, with two of those instance variables being initialized when they are declared.

The instance variable named *animate*

The first instance variable, named **animate**, is used later to store a reference to a **Thread** object. The **Thread** object is used to produce the animation. *(I will discuss the class from which this object is instantiated later.)*

The instance variable named *images*

The second instance variable, named **images**, is a reference to a three-element array object of type **ImageIcon**. This array object is initialized to contain references to three **ImageIcon** objects. The three **ImageIcon** objects are created, and the references to those objects are stored in the array when it is declared.

*(In Java, the elements of a new array object can be initialized by including a list of comma-separated expressions inside a matching pair of curly braces. Each expression corresponds to one element in the array. The size of the array is determined by the number of expressions. The contents of each array element are determined by evaluating the corresponding expression. Each expression must evaluate to a value that is assignment-compatible with the declared type of the array elements, which in this case is **ImageIcon**.)*

The ImageIcon class

Here is part of what Sun has to say about the **ImageIcon** class.

> *"An implementation of the **Icon** interface that paints Icons from Images. Images that are created from a URL or filename are preloaded using **MediaTracker** to monitor the loaded state of the image."*

Thus, the **ImageIcon** class implements the **Icon** interface. The class provides a large number of overloaded constructors.

The Icon interface

Here is part of what Sun has to say about the **Icon** interface:

> *"A small fixed size picture, typically used to decorate components."*

In this program, the objects of the interface type **Icon** *(objects of the class **ImageIcon**)* will be used to decorate a **JLabel** component.

ImageIcon constructors

The overloaded constructors of the **ImageIcon** class have the ability to create **ImageIcon** objects from a variety of sources. The overloaded version that I used in this program creates **ImageIcon** objects from gif files, as indicated by the code in Listing 1.

> *(Each of the ImageIcon objects created in Listing 1 is based on a different gif file. These gif files contain images of the stick man in different positions.)*

The MediaTracker class

In some situations, particularly when the file containing the image is large, or is not stored locally, the actions of a **MediaTracker** object can be important *(that is not the case in this program).* Here is part of what Sun has to say about the **MediaTracker** class.

> *"The **MediaTracker** class is a utility class to track the status of a number of media objects. Media objects could include audio clips as well as images, though currently only images are supported."*

Purpose of a MediaTracker object

Basically, the purpose of a **MediaTracker** object is to notify the program of the loading state of an image *(media object).* This can be important in situations where the amount of time required to load the image is significant. That is not the case in this program because:

- The images are small.

- The images are loaded from the local hard drive.
- The images are loaded and converted to **ImageIcon** objects when the object of the controlling class is created. *(The images are probably fully loaded before the user has time to pick up the mouse.)*

Therefore, I didn't need to use a **MediaTracker** object to track the load status of the images in this program.

The instance variable named *label*

Referring back to Listing 1, the instance variable named **label** is a reference to a **JLabel** object, which is initially decorated with one of the **ImageIcon** objects discussed above.

> *(The JLabel object is initially decorated with the **ImageIcon** object whose reference is stored at index 0 in the array. Later on, the value of the **Icon** property of the label changes with time to produce the animation.)*

The JLabel class

Here is part of what Sun has to say about the **JLabel** class:

> *"A JLabel object can display either text, an image, or both. You can specify where in the label's display area the label's contents are aligned ..."*

In this program, the JLabel object displays only an image. It does not display text.

JLabel constructors

The **JLabel** class provides several overloaded constructors, which allow you to specify the text, the image, and the alignment of the two when the object is constructed.

When the **JLabel** is intended to display an image, the incoming parameter to the constructor must be a reference to an object of the interface type **Icon**. This requirement is satisfied by the code in Listing 1, because the **ImageIcon** class implements the **Icon** interface.

The setIcon method of the JLabel class

The **JLabel** class also provides a method named **setIcon**, which requires an incoming parameter of the interface type **Icon**. This method makes it possible to change the image being displayed by the label at runtime. This method will be used to cycle through images of the stick man to produce the animation in this program.

The constructor for Animate04

The constructor for the **Animate04** class begins in Listing 2.

```
  public Animate04(){//constructor

    getContentPane().add(label);

Listing 2
```

As you saw earlier in Listing 1, the **Animate04** class extends the **JFrame** class. The code in Listing 2 places the **JLabel** object on the *content pane* of the **JFrame** object.

The getContentPane method

The **getContentPane** method returns a reference to the content pane. In case you are unfamiliar with the content pane, I discuss it extensively in my tutorial lesson entitled Swing, Understanding getContentPane() and other JFrame Layers.

To make a long story short, a **JFrame** object provides several thousand transparent layers on which you can place components, *(such as **JLabel** objects, **JButton** objects, etc.).* Each component is located horizontally and vertically by specifying the **x** and **y** coordinate position for the upper left-hand corner of the component. *(A layout manager may automatically perform that task for you.)*

In a three dimensional sense, the *z-order* or depth coordinate value of each components can be controlled by specifying the number of the layer on which you want to place each component. This causes some components to appear to be *in front of* other components.

What is the content pane?

The *content pane* is the default layer on which you place components if you don't need to control the z-order placement. All components placed on the content pane are at the same level in the z-order.

In this program, we are placing only one component, *(a **JLabel** object),* in the **JFrame**. Therefore, we aren't concerned about z-order among components, and we simply place that component on the content pane *(default layer)* as shown in Listing 2.

An anonymous listener class

The code in Listing 3 creates an anonymous **MouseListener** object from an anonymous class and registers it to listen for mouse events on the content pane. The purpose is to detect when the mouse enters and exits the area of the content pane so that the animation can be started and stopped on that basis.

```
    getContentPane().addMouseListener(
      new MouseAdapter(){
        public void
mouseEntered(MouseEvent e){
          //Body code deleted for
```

```
brevity
        }//end mouseEntered

        public void
mouseExited(MouseEvent e){
          //Body code deleted for
brevity
        }//end MouseExited
      }//end new MouseAdapter
    );//end addMouseListener()

Listing 3
```

The code in Listing 3 is some of the most cryptic code in all of Java. Just in case you aren't already familiar with anonymous classes, I will take some time to explain the code in Listing 3.

The JavaBeans event model

The code in Listing 3, plus what I am about to explain to you, is based on the *JavaBeans* event model *(previously called the Delegation Event Model).*

> *(My series of tutorial lessons, beginning with Event Handling in JDK 1.1, A First Look, Delegation Event Model, discuss event-driven programming in detail.)*

Making a long story short

Once again, to make a long story short, the JavaBeans event model depends on sources and listeners. In this program, the content pane of the **JFrame** object will act as a source for mouse events. The anonymous **MouseListener** object registered on the content pane in Listing 3 will listen for, and handle those events when they occur.

What is a MouseListener object?

To be a mouse listener, an object must be instantiated from a class that implements the **MouseListener** interface. The **MouseListener** interface declares the following five methods:

- mouseClicked
- mouseEntered
- mouseExited
- mousePressed
- mouseReleased

As you probably already know, any class that implements an interface must provide a concrete definition for each method declared in the interface. Thus, any class that implements the **MouseListener** interface must provide a concrete definition for each of the five methods listed above.

A convenience class named MouseAdapter

The Java API provides a convenience class named **MouseAdapter**, which implements the **MouseListener** interface. This class defines concrete *(but empty)* versions of the five methods declared in the interface. A programmer needing to define a class from which **MouseListener** objects can be instantiated has two choices:

- Implement the **MouseListener** interface and define all five methods.
- Extend the **MouseAdapter** class and override only the methods of interest.

I elected the second option in this program. Furthermore, I elected to do it in a very common, but very cryptic way.

Overrides mouseEntered and mouseExited methods

To begin with, the code in Listing 3 overrides the **mouseEntered** and **mouseExited** methods, and ignores the other three methods.

The MouseListener registration method

The **addMouseListener** method in Listing 3 is used to *register* a **MouseListener** object on a source of mouse events. In this case, the source is the content pane.

> *(Once a **MouseListener** object is registered on a source of mouse events, the listener object will be notified each time a mouse event occurs on the source object. The notification methodology is to invoke the specific method, chosen from the five methods listed above, corresponding to the kind of mouse event that occurred. The behavior of the method will determine the response to the event.)*

The parameter required for registration

The **addMouseListener** registration method requires an incoming parameter of type **MouseListener**, which is a reference to a **MouseListener** object. This parameter is created in Listing 3 by instantiating an anonymous object inside the parenthesis of the call to the **addMouseListener** method.

What is an anonymous object?

It is very common in Java to instantiate a new anonymous object inside the parenthesis of a method call.

> *(An anonymous object is an object whose reference is not assigned to a named reference variable. Hence, it has no name and is "anonymous.")*

For example, the following statement instantiates a new anonymous object of the **Date** class and passes that object's reference to the **println** method. This will cause the current date and time to be displayed on the standard output:

System.out.println(new Date());

Not an ordinary anonymous object

However, the anonymous object instantiated in Listing 3 is not an ordinary anonymous object. An ordinary anonymous object would be instantiated using an expression of the form

new MouseListenerClass()

where **MouseListenerClass** is the name of a class that implements the **MouseListener** interface.

A non-existent MouseListener class

In the code in Listing 3, the anonymous object is instantiated from a non-existent **MouseListener** class. In other words, the class definition does not exist outside of the code in Listing 3.

The **MouseListener** object is instantiated anonymously in Listing 3. Furthermore, the class from which the object is instantiated is also defined anonymously within the parentheses that make up the call to the **addMouseListener** method.

A verbal interpretation

You can verbally interpret the first two lines of code in Listing 3 as follows:

> *Register, on the content pane, a **MouseListener** object, instantiated from a previously undefined class, which has no name, but which extends the class named **MouseAdapter**.*

Overridden interface methods

Beginning with the third line in Listing 3, the next several lines in Listing 3 override the **mouseEntered** and **mouseExited** methods declared in the **MouseListener** interface and defined in the **MouseAdapter** class.

> *(Note that the body code for each of the overridden methods in Listing 3 was deleted for brevity. I wanted to be able to explain the structure first without getting bogged down in detailed body code.)*

Only one semicolon

If you examine the code in Listing 3 carefully, you will see that it contains only one semicolon. Therefore, everything in Listing 3 is part of a single statement. The final two lines in Listing 3 provide the curly brace, parenthesis, and semicolon required to complete the statement.

An anonymous listener object of an anonymous class

Once the code in Listing 3 has been executed, a new anonymous object has been instantiated from an anonymous inner class. The new object has been registered on the content pane at that point.

The anonymous class is a class that implements the **MouseListener** interface. Whenever a mouse event occurs on the content pane, the appropriate method in the anonymous listener object will be invoked to handle the event.

Expanding Listing 3

Now I am going to provide two listings that expand on the omitted details from Listing 3. Listing 4 shows the complete **mouseEntered** method from Listing 3, with the body code restored. Similarly Listing 5, *(to be discussed later),* shows the complete **mouseExited** method from Listing 3 with the body code restored.

```
        public void
mouseEntered(MouseEvent e){
          animate = new Animate();
          animate.start();
        }//end mouseEntered

Listing 4
```

The mouseEntered method

After the **MouseListener** object is registered on the content pane, the **mouseEntered** method shown in Listing 4 will be invoked each time the mouse pointer enters the area of the screen occupied by the content pane.

The behavior of the **mouseEntered** method is:

- Instantiate a new object of the **Animate** class.
- Invoke the **start** method on that object.

The Animate class is a Thread class

The **Animate** class, which I will discuss in detail later, extends the **Thread** class, and defines the **run** method. Here is part of what Sun has to say in the documentation for the **Thread** class:

> *"A **thread** is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently."*

The start method

Here is part of what Sun has to say about the **start** method of the **Thread** class:

*"Causes this thread to begin execution; the Java Virtual Machine calls the **run** method of this thread.*

*The result is that two threads are running concurrently: the current thread (which returns from the call to the **start** method) and the other thread (which executes its **run** method)."*

Thus, when the **start** method is invoked on an object instantiated from the **Animate** class, the **run** method belonging to that object is executed in a separate *thread*, running concurrently with the thread that handles events. This makes it possible to produce the animation and continue to monitor for events at the same time.

The run method

Later, when we examine the **run** method of the **Animate** class, we will see that it causes the three images discussed earlier to be repetitively displayed in succession. That is what produces the illusion that the stick man is dancing. That is also what I refer to as frame animation.

The mouseExited method

The **mouseEntered** method shown in Listing 4 is invoked when the mouse pointer enters the area of the screen occupied by the content pane. This causes animation of the stick man to begin.

The **mouseExited** method shown in Listing 5 is invoked when the mouse pointer leaves the area of the screen occupied by the content pane. This causes the animation of the stick man to stop.

Stopping the animation

The process of stopping the animation is a little more complicated than the process of starting the animation. This is evidenced by the code in Listing 5, which shows the full body of the **mouseExited** method from Listing 3.

```
        public void
mouseExited(MouseEvent e){

          animate.interrupt();


while(animate.isAlive()){}//loop;

          animate = null;

          label.setIcon(images[0]);
          label.repaint();
        }//end MouseExited

Listing 5
```

Basically, here is what happens as a result of invoking the **mouseExited** method:

- Interrupt the animation thread to terminate the animation.
- Loop for a few cycles while the animation thread finishes everything that it needs to do and dies a natural death.
- Hand the animation object over to the garbage collector.
- Restore the default image to the label so that it will always be the same when not being animated.

The interrupt method

The result of invoking the **interrupt** method on a **Thread** object depends on the current state of the **Thread** object. As you will see when we examine the code in the **run** method of the **Thread** object, we are concerned with the Thread object being in one of two possible states:

- The thread is sleeping.
- The thread is not sleeping, but rather is involved in selecting a new icon and causing it to be painted on the screen.

Interrupting the Thread object

The **Thread** object cycles between these two states in order to:

- Display a new stick man image.
- Delay for an appropriate amount of time so that the new image will register in the brain of the viewer. If the delay is too short, the image won't be seen by the viewer and the animation will just be a blur. If the delay is too long, the animation will appear to be jerky.

If the **interrupt** method is invoked on the **Thread** object while it is sleeping, an **InterruptedException** will be thrown. We will stop the animation when the animation is thrown.

If the **interrupt** method is invoked on the **Thread** object while it is selecting a new icon and causing it to be painted on the screen, the *interrupt* status for the **Thread** object will be set to true. We will test this status in the **run** method before causing the **Thread** object to go to sleep. If the status is true at that point in time, we will purposely throw an **InterruptedException**. Again, we will stop the animation when the exception is thrown. However, in this case, a few extra machine cycles may occur before the exception is thrown.

The isAlive method

Shortly after the **InterruptedException** is thrown, the thread's **run** method will terminate normally, and the thread will die a natural death. Because this doesn't happen instantly, the code

in Listing 5 loops until the **isAlive** method returns false. This ensures that no further action is taken on the **Thread** object while it is still alive.

> *(I always get a little nervous about writing a loop in an event handler for fear that it may take too long to terminate. However, I decided to take a chance in this case, and it seems to work OK under Win2000. However, a more sophisticated approach may be required on other operating systems, particularly those that don't provide automatic time slicing to prevent the event-handling thread from hogging the CPU.)*

Eligible for garbage collection

Continuing with the code in Listing 5, once the thread is no longer alive, a *null* value is assigned to the reference variable that refers to the **Thread** object. This causes the object to become eligible for garbage collection.

The setIcon method

As mentioned earlier, the **setIcon** method can be used to define the icon a component will display. The code in Listing 5 causes the icon that will be displayed to be the same icon that is displayed when the program starts. Thus, the same icon is displayed whenever the stick man is not being animated.

The repaint method

The purpose of the **repaint** method is to cause the screen area occupied by a component to be repainted as soon as possible. In this case, that causes the label with the new icon to appear on the screen.

Invoking the **repaint** method on a **JLabel** component sends a request to the operating system to invoke the component's **paint** method to repaint the component. Since I didn't override the **paint** method for the **JLabel** object**,** the default version of the **paint** method for the label is actually invoked.

Not a blocking method

The **repaint** method is not a blocking method. This will become important later when we discuss the animation thread. Rather, the **repaint** method sends the *repaint* request to the operating system and returns immediately. In other words, control does not necessarily stay within the **repaint** method until the component has been repainted.

The operating system may not honor all repaint requests

In point of fact, the operating system is not required to honor all repaint requests. For example, if a series of repaint requests for the same component are made in short succession, the operating system has the option to ignore all the intermediate repaint requests and honor only the final

request in the series. This could happen if the time interval between repaint requests is less than the time required to perform the repaint operation on the screen.

The mouseExited method terminates

Once the repaint request is made, the **mouseExited** method shown in Listing 5 terminates.

> *(The actual repainting of the screen may not have been accomplished by that point in time. It may actually occur later.)*

The virtual machine is then free to honor the next event in the system event queue.

Still in the constructor

Recall that we are still discussing the code in the constructor. The final four statements in the constructor are shown in Listing 6. With the possible exception of the first statement in Listing 6, the final four statements in the constructor should be self-explanatory based on the names of the methods involved.

```
setDefaultCloseOperation(EXIT_ON_CLOSE);
    setTitle("Copyright 2003,
R.G.Baldwin");
    setSize(250,200);
    setVisible(true);

  }//end constructor
```

Listing 6

The setDefaultCloseOperation method

The **setDefaultCloseOperation** method specifies the action to be taken when the user clicks the *close* button on the frame.

> *(On a Windows system, this is the button with the X in the upper right corner of the **JFrame** object as shown in Figure 1).*

Several optional actions can be specified. When the value EXIT_ON_CLOSE is passed as a parameter to the method, this specifies that the program should terminate when the user clicks the close button.

The ordinary inner class

At this point, I am going to temporarily skip the **main** method and discuss the ordinary inner class named **Animate**. First, however, a few comments are in order regarding the use of inner classes.

Why use inner classes?

The primary benefit of using inner classes has to do with accessibility of instance variables and instance methods. Basically, the code in an inner class has direct access to all the instance variables and all the instance methods of all outer enclosing classes. In some cases, this can eliminate a great deal of parameter passing that is required when all classes are defined as top-level classes.

For example, the code in the anonymous inner class discussed above directly accesses the instance variable named **animate**. The code also accesses the instance variable named **images**, and the instance variable named **label**.

These are all instance variables of the outer enclosing class. If the event handling code had been placed in a top-level class instead, it would have been necessary to pass copies of those instance variables to the constructor for the top-level class and to save those copies for use within the **mouseEntered** and **mouseExited** methods of that class. Thus, it is often much more convenient to define a class an either an ordinary inner class or an anonymous inner class.

What happens at compile time?

Every class that is defined in a Java program results in a compiled file with an extension of *class* regardless of whether it is a top-level class, an ordinary inner class, or an anonymous inner class. For example, when compiled, this program produces the following three files:

- Animate04$1.class
- Animate04$Animate.class
- Animate04.class

The first file in the above list results from compiling the anonymous inner class discussed above. Obviously, the class file does have a name, which is automatically assigned by the virtual machine. However, that name is never used explicitly by the programmer. Hence the class is called an anonymous inner class.

Even though the anonymous class, in this case, is defined inside the constructor for the outer controlling class, it is actually compiled right along with the other classes that make up the program. In other words, the compilation of this class has nothing to do with the invocation of the constructor at runtime.

The second file in the above list results from compiling the ordinary inner class that we will discuss next. The third file in the above list results from compiling the controlling class for this program.

The inner class named Animate

An ordinary inner class is an inner class because its definition resides inside the definition of another class. However, it doesn't reside within the argument list of a method call, as is the case for an anonymous inner class.

The purpose of this inner class is to provide the animation action for the program. The class definition begins in Listing 7.

```
  class Animate extends Thread{

Listing 7
```

A class that extends the Thread class

You will recall that the purpose of the **Animate** class is to behave as a thread, executing concurrently with other threads in the program. You can learn more about threads by reviewing my tutorial lesson entitled Threads of Control.

Two approaches to multi-threaded programming

Briefly, there are two different ways that you can define a class from which you can instantiate thread objects:

- Implement the **Runnable** interface
- Extend the convenience class named **Thread**, which in turn implements **Runnable**

Even though the second approach is sometimes more convenient than the first, because Java doesn't support multiple inheritance, it isn't always possible for your class to extend the **Thread** class. However, it is always possible for your class to implement the **Runnable** interface.

As you can see in Listing 7, I elected to define a new class named **Animate**, which extends the **Thread** class.

Defining the run method

Regardless of which approach you take, your new class must provide a concrete definition for the method named **run**, which is declared in the **Runnable** interface.

Here is what Sun has to say about the **run** method:

> *"When an object implementing interface **Runnable** is used to create a thread, starting the thread causes the object's **run** method to be called in that separately executing thread.*
>
> *The general contract of the method **run** is that it may take any action whatsoever."*

Somewhat analogous to the main method

I like to think of the **run** method of a thread as being somewhat analogous to the **main** method of an application. When you execute a Java application, its **main** method is always executed. When you invoke the **start** method on a **Runnable** object, its **run** method is always executed.

> *(The **start** method was invoked on an object of the **Animate** class in the **mouseEntered** method shown in Listing 4.)*

As mentioned earlier, when the **run** method of a thread terminates, the thread is deemed to no longer be alive. *(I made use of that fact in the **mouseExited** method discussed in Listing 5 earlier.)*

When the **main** method no longer has anything to do, and there are no live *non-daemon* threads, the application will terminate. *(I'm not going to get into a discussion of daemon threads here. See Threads of Control for a detailed discussion.)*

The run method of the Animate class

The **run** method of the **Animate** class contains two blocks of code:

- A **try** block
- A **catch** block

The try block is shown in Listing 8.

```
    public void run(){//begin run
method
      try{
        while(true){
          display(1,500);
          display(0,500);
          display(2,500);
          display(0,500);
        }//end while loop
      }//end try block
Listing 8
```

The try block

The code in the **try** block consists of an infinite loop. This loop continues to loop and invoke the method named **display** several times during each iteration until the **display** method throws an exception.

Display the stick man

The **display** method, *(which we will examine shortly)*, causes different images of the stick man to be displayed, depending on the value of the first parameter passed to the method.

Briefly, the two parameters to the **display** method specify which image to display, and how long to display that image.

> *(The first parameter is an index into the array containing the **ImageIcon** objects from Listing 1, and the second parameter is the display time in milliseconds.)*

This is the code that actually produces the animation. The four successive invocations of **display** cause four views of the stick man to be displayed. Then another iteration begins and the same four views appear again.

> *(If you need a more complex animation algorithm possibly involving more images and varying display times, this is where you would provide a suitable algorithm for that purpose.)*

An InterruptedException may be thrown

This animation process continues until the **display** method throws an exception.

> *(It is expected that an **InterruptedException** will be thrown when the **mouseExited** method invokes the **interrupt** method on the **Animate** thread object.)*

As is always the case, when an exception is thrown, execution of the code terminates and the virtual machine begins searching for a **catch** block whose type is compatible with the type of exception thrown. If such a match is found, control is transferred to that block, and the code in the catch block is executed.

The catch block

The **catch** block, which constitutes the second block of code in the **run** method, begins in Listing 9.

```
      catch(Exception ex){
        if(ex instanceof
InterruptedException){
          //Do nothing. This exception
is
          // expected on mouseExited.
        }//end if

Listing 9
```

This catch block is compatible with exceptions of type **Exception**, or any subclass of **Exception**. This includes **InterruptedException** and most other exceptions as well.

If an InterruptedException has occurred ...

The code in the **catch** block uses the **instanceof** operator to determine if the exception is of type **InterruptedException**. If so, no further action is taken. By the time control reaches the **catch** block as a result of an **InterruptedException**, animation has been terminated, which is the desired result of invoking the **interrupt** method in the **mouseExited** method. Therefore, no further action is required on the part of the thread object.

Some other type of exception

On the other hand, if the exception is of some type other than **InterruptedException**, this is an unexpected exception. In this case, the code in Listing 10 prints an error message and terminates the program.

```
        else{//Unexpected exception
occurred.
          System.out.println(ex);
          System.exit(1);//terminate
program
        }//end else
      }//end catch
    }//end run

Listing 10
```

And that is the end of the **run** method.

The display method

Recall that we are still discussing the code in the **Animate** class, from which **Thread** objects are instantiated to perform the actual animation of the stick man.

The **display** method begins is listing 11. Before getting into the details of the code, however, consider the following overview. When the **display** method is invoked, it causes a specified image to be displayed. Then the thread goes to sleep for a prescribed period of time.

If the thread is interrupted *(by code in the **mouseExited** method)* while the **display** method is displaying the image, the code in the **display** method throws an **InterruptedException** and terminates.

If the thread wakes up prior to being interrupted, it returns normally and animation continues.

If the thread goes to sleep and is interrupted before it wakes up, it automatically throws an **InterruptedException**, causing the **display** method to terminate.

Transfer of control

In any case that an exception is thrown while control is within the **display** method, animation is terminated, and control is transferred to the **catch** block that begins in Listing 9.

The code in the display method

The beginning of the **display** method is shown in Listing 11.

```
    void display(int image,int delay)
                      throws
InterruptedException{
      //Select and display an image.
      label.setIcon(images[image]);
      label.repaint();

Listing 11
```

The **display** method receives two incoming parameters of type **int**. The first parameter is an index into the array containing the **ImageIcon** objects. The second parameter is the number of milliseconds to sleep after displaying the image specified by the first parameter.

The two statements in Listing 11 set and display an image. This code is identical to the code discussed earlier in Listing 5. Therefore, I won't discuss it further here.

Checking the interrupt status

If control reaches this point, and the **interrupt** method has been invoked on this **Thread** object since its instantiation *(meaning that the **mouseExited** method has been invoked)* the ***interrupt*** status of the thread will have been set to true.

The code in Listing 12 invokes the **interrupted** method to check the *interrupt* status of the thread. This code purposely throws an **InterruptedException** if the ***interrupt*** status is **true**.

```
if(Thread.currentThread().interrupted())
        throw(new
InterruptedException());

Listing 12
```

This will immediately terminate the **display** method and transfer control to the **catch** block that begins in Listing 9.

Go to sleep

If the ***interrupt*** status of the thread is **false**, the code in Listing 13 causes the **Thread** object to go to sleep for the prescribed number of milliseconds.

```
      //Delay specified number of
msec.
      //Terminate animation
automatically if
      // interrupted while asleep.

Thread.currentThread().sleep(delay);
    }//end display method

Listing 13
```

This will cause the current image to be displayed until the thread wakes up and returns normally, or until the thread is interrupted by the code in the **mouseExited** method.

Behavior when asleep

If the thread is not interrupted while it is asleep, it will wake up after the prescribed number of milliseconds and return normally to the code in Listing 8.

If the thread is interrupted while it is asleep, it will wake up immediately, throw an **InterruptedException**, and transfer control to the **catch** block that begins is Listing 9. As mentioned earlier, this will stop the animation, which is the desired result of invoking the **interrupt** method within the **mouseExited** method.

End of the Animate class definition

With the exception of one more curly brace, that ends the definition of the inner class named **Animate**. *(I won't waste your time by showing you that curly brace. You can view it in Listing 15 near the end of the lesson.)*

The main method

Finally, the **main** method for this program is shown in Listing 14.

```
  public static void main(String[]
args){
    new Animate04();
  }//end main

Listing 14
```

All that the **main** method does is invoke the constructor to instantiate a new object of the controlling class named **Animate04**.

Let's recap

To recap, here is what happens when the constructor for the **Animate04** class is invoked and the object is instantiated:

- The **JLabel** object, which was initialized to display the default image, is added to the *content pane* of the **JFrame** object.
- An object of the anonymous class *(stored in the class file named Animate04$1.class)* is instantiated and registered as a mouse listener on the content pane of the **JFrame** object. This object contains specified behavior for the **mouseEntered** and **mouseExited** methods. The other three methods declared in the **MouseListener** interface are defined with empty bodies. Thus, if invoked, these three methods will return immediately without performing any action.
- Property values are established for the following properties: *defaultCloseOperation*, *title*, and *size*.
- The *visible* property is set to true, which causes the **JFrame** object, and the **JLabel** object that it contains to appear on the screen.

A quiescent state

At this point, the program enters a quiescent state, waiting for the user to cause an event of some type. There are dozens of different types of event that the user can cause to occur. However, only two of those event types will be processed by the program. All the rest will be ignored.

> *(Actually, only one event type, **MouseEvent**, will be processed. The animation behavior of the program is based on two sub-categories of the **MouseEvent** type. To avoid unnecessary complexity, I will refer to them simply as two events.)*

Two events that will be processed

The two events that will be processed by the program are two sub-categories of the event type **MouseEvent**:

- **mouseEntered** - occurs when the mouse pointer enters the area of the screen occupied by the *content pane* of the **JFrame** object.
- **mouseExited** - occurs when the mouse pointer exits the area of the screen occupied by the *content pane* of the **JFrame** object.

The mouseEntered event

A **mouseEntered** event causes an object to be instantiated from the **Thread** class named **Animate**. Once the object is instantiated, the **run** method of the **Thread** object is started. The **run** method runs continuously, causing the stick man to be animated, until the **Thread** object is interrupted.

The mouseExited event

A **mouseExited** event causes the **interrupt** method to be invoked on the **Thread** object. This causes the **run** method of the **Thread** object to terminate, which in turn causes the animation of the stick man to stop.

This also causes the **Thread** object to die a natural death. Once the **Thread** object is no longer alive, the **mouseExited** method makes the **Thread** object eligible for garbage collection, and displays the default image on the label on the **JFrame** object.

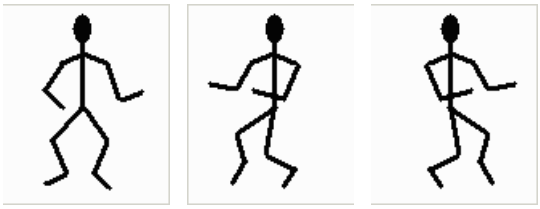
Run the Program

At this point, you may find it interesting to compile and run the **Animate04** program shown in Listing 15 near the end of the lesson. Because I tested the program using SDK version 1.4.1, I recommend that you compile and run it using that version or a later version of the SDK.

You should be able to copy and paste the program into your editor. Then store it in a file named **Animate04.java**.

The stick-man images

In order to compile and run the program, you will also need to download the three image files shown below. You should be able to download them individually by right-clicking on them and saving them on your local disk.



Save them under the following file names in the same directory containing your source code file and compiled files:

- java1470a.gif
- java1470b.gif
- java1470c.gif

Summary

In this lesson, I taught you how to do frame animation. Equally important, I also taught you about event-driven programming, multi-threaded programming, ordinary inner classes, anonymous inner classes, exception handling, and image icons.

Complete Program Listing

A complete listing of the program is shown in Listing 11.

```
/*File Animate04.java
Revised 01/07/03

Illustrates frame animation.  Stick figure
dances when user point to the image with the
mouse.  Stick figure stops dancing when mouse
pointer exits the image.

Also illustrates:
  Event-driven programming
  Multi-threaded programming
  Ordinary inner classes
  Anonymous inner classes
  Image icons

Tested using SDK 1.4.1 under Win 2000.
**********************************************/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Animate04 extends JFrame{
  Thread animate;//Store ref to animation thread
  ImageIcon images[] = {//An array of images
        new ImageIcon("java1470a.gif"),
    new ImageIcon("java1470b.gif"),
    new ImageIcon("java1470c.gif")};
  JLabel label = new JLabel(images[0]);

  //---------------------------------------//
  public Animate04(){//constructor

    getContentPane().add(label);

    //Use an anonymous inner class to register a
    // mouse listener
    getContentPane().addMouseListener(
      new MouseAdapter(){
        public void mouseEntered(MouseEvent e){
          //Get a new animation thread and start
          // the animation on it.
          animate = new Animate();
          animate.start();
        }//end mouseEntered

        public void mouseExited(MouseEvent e){
          //Terminate the animation.
          animate.interrupt();
          //Let the thread die a natural death.
          // Then make it eligible for garbage
          // collection.
          while (animate.isAlive()){}//loop;
          animate = null;
          //Restore default image.
```

```
          label.setIcon(images[0]);
          label.repaint();
        }//end MouseExited
      }//end new MouseAdapter
    );//end addMouseListener()
    //End definition of anonymous inner class

    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setTitle("Copyright 2003, R.G.Baldwin");
    setSize(250,200);
    setVisible(true);

  }//end constructor
  //---------------------------------------//

  public static void main(String[] args){
    new Animate04();
  }//end main
  //---------------------------------------//

  //Ordinary inner class to animate the image
  class Animate extends Thread{

    public void run(){//begin run method
      try{
        //The following code will continue to
        // loop until the animation thread is
        // interrupted by the mouseExited
        // method.
        while(true){
          //Display several images in succession.
          display(1,500);
          display(0,500);
          display(2,500);
          display(0,500);
        }//end while loop
      }catch(Exception ex){
        if(ex instanceof InterruptedException){
          //Do nothing. This exception is
          // expected on mouseExited.
        }else{//Unexpected exception occurred.
          System.out.println(ex);
          System.exit(1);//terminate program
        }//end else
      }//end catch
    }//end run
    //---------------------------------------//

    //This method displays an image and sleeps
    // for a prescribed period of time.  It
    // terminates and throws an
    // InterruptedException when interrupted
    // by the mouseExited method.
    void display(int image,int delay)
                    throws InterruptedException{
      //Select and display an image.
```

```
      label.setIcon(images[image]);
      label.repaint();
      //Check interrupt status.  If interrupted
      // while not asleep, force animation to
      // terminate.
      if(Thread.currentThread().interrupted())
        throw(new InterruptedException());
      //Delay specified number of msec.
      //Terminate animation automatically if
      // interrupted while asleep.
      Thread.currentThread().sleep(delay);
    }//end display method
    //-------------------------------------//
  }//end inner class named Animate

}//end class Animate04
```

Listing 15

About the author

Richard Baldwin is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which has gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

-end-