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

Animate Objects

Chapter Overview

- How do I create an object that can act by itself?

This chapter builds on the previous ones to create an object capable of acting without an external request. Such an object has its own instruction follower, in Java called a Thread. In addition, an object with its own instruction-follower must specify what instructions are to be followed. This is accomplished by implementing a certain interface -- meeting a particular contract specification -- that indicates which instructions the Thread is to execute.

The remainder of this chapter deals with examples of how Threads and animate objects can be used to create communities of autonomously interacting entities.

Objectives of this Chapter

- To understand that Threads are Java's instruction-followers.
- To appreciate the relationship between a Thread and the instructions that it executes.
- To be able to construct an animate object using AnimatorThread and Animate.

Animate Objects

In previous chapters, we saw how objects group together state and behavior. Some objects exist primarily to hold together constituent pieces of a single complex state. Other objects exist to hold a static collection of primarily functional or system-specific resources. Most objects contain both local state and methods that rely on and interact with this state in complex ways. Many of these objects wait for something to happen or for someone else to ask them to act. That is, nothing happens until something outside the object invokes a method of the object. In this chapter, we look at objects that are capable of taking action on their own, without being asked to do so from outside. These objects have their own instruction-followers, making them full-blown entities.

Consider, for example, the Counter. This is a relatively traditional object. It has both state and methods that depend on that state. An individual counter object encapsulates this state-dependent behavior, wrapping it up into a neat package. But a counter doesn't do anything unless someone asks it to, using its increment() or reset() method. By itself, a counter can't do much.

Contrast this with a timer. A timer is very similar to a counter in having a method that advances it to the next state (paralleling the counter's increment() method) and one that sets the state back to its default

condition (such as reset()). A timer differs from a counter, however in that a timer counts merrily along whether someone asks it to or not. The timer's reset() method is a traditional (passive) method; the timer resets only when asked to. But the timer's increment() method is called by the timer itself on a regular basis.

This kind of object -- one that is capable of acting without being explicitly asked to do so -- is called an animate object. Such an object has its own instruction-follower, or actor, associated with it. While traditional objects are roles that an actor may take on and then leave, an animate object is a role that is almost always inhabited by an actor and tightly associated with it. Often, animate objects will use traditional objects (as well as data repositories, resource libraries, and other kinds of objects) to perform their tasks, temporarily executing instructions contained in these objects. But the animate object is where it begins and ends.

What makes an animate object different from other (passive) objects? Recall that on the first page of the first chapter of this book, we learned about the two prerequisites for a computation: The instructions for the computation must be present, and those instructions must be executed. Every method of every object is a set of instructions -- a rule -- that can be executed. When a method is invoked, its body is executed. (The method body is executed by the instruction-follower that invoked the method; this is how a method invocation expression is evaluated.)

An animate object differs from other objects because it also has its instruction follower. It does not need to wait for another instruction-follower to invoke one of its methods (although this may also happen). Instead, it has a way to start execution on its own.

In Java, an instruction-follower is called a Thread. No object can act except a Thread. A Thread is a special object that "breathes life" into other objects. It is the thing that causes other objects to move. An animate object is simply an object that is "born" with its own Thread. (Typically, this means that it creates its own Thread in its constructor and starts its Thread running either in its constructor or as soon as otherwise possible.)

Animacies are Execution Sequences

In every method of every object, execution of that method follows a well-defined set of rules. When the method is invoked, its formal parameters are associated with the arguments supplied to the method call. For example, recall the UpperCaser StringTransformer:

```
public class UpperCaser extends StringTransformer
{

     public String transform( String what )
     {
         return what.toUpperCase();
     }

}
```

If we have `UpperCaser cap = new UpperCaser();` then evaluating the expression `cap.transform( "Who's there?")` has the effect of associating the value of the String `"Who's there?"` with the name `what` during the execution of the body of the `transform` method.

Now, the first statement of the method body is executed. In the case of the method invocation expression `cap.transform( "Who's there?")`, there is only one statement in the method body. This is the return statement, which first evaluates the expression following the return, then exits the method invocation, returning the value of that expression. To evaluate the method invocation expression `what.toUpperCase()` involves first evaluating the name expression `what` and then invoking the `toUpperCase()` method of the object associated with the name `what`.

No matter how complex the method body, its execution is accomplished by following the instructions that constitute it. Each statement has an associated execution pattern. A simple statement like an assignment expression followed by a semicolon is executed by evaluating the assignment expression. Expressions have rules of evaluation; in the case of an assignment, the right-hand side expression is evaluated, then that value is assigned to the left-hand side (shoebox or label). Evaluating the right-hand side expression may itself be complicated, but by following the evaluation rules for each constituent expression, the value of the right-hand side is obtained and used in the assignment.

A more complex statement, such as a conditional, has execution rules that involve the evaluation of the test expression, then execution of one *but not both* of the following substatements (the "if-block" or the "else-block"). Loops and other more complex statements also have rules of execution. Declarations set up name-value associations; return statements exit the method currently being executed.

At any given time, execution of a particular method is at a particular point and in a particular context (i.e., with a particular set of name-value associations in force). If we could keep track of what we're in the middle of doing and what we know about while we're doing it, we could temporarily suspend and resume execution of this task at any time. Imagine that you're following an instruction booklet to assemble a complex mechanism. This problem is a lot like placing a bookmark into your instructions while you go off to do something else for a while. All you need to know is where you were, what you had around you, and what you were supposed to do next; the rest of the instructions will carry you forward.

Inside the computer, there are things that keep track of where you are in an execution sequence. These are special Java objects called Threads. The trick is that there can be more than one Thread in any program. In fact, there are exactly as many things going on at once as there are Threads executing in your program. A Thread keeps track of where it is in its own execution sequence. Each Thread works on its own assembly project using its own instruction booklet, just like multiple people can work side by side in a restaurant or a factory.

In this book, we will make extensive use of a special kind of Thread called an AnimatorThread. An AnimatorThread is an instruction follower that does the same thing over and over again. It also has some other nice properties: it can be started and stopped, suspended and resumed. These last two mean that it is possible to ask your instruction follower to take a break for a while, then ask it later to continue working. AnimatorThreads provide a nice abstraction for the kinds of activities commonly conducted by the animate objects that are often entities in our communities.

Being Animate-able

In order for a Thread to animate an object, the Thread needs to know where to begin. A Thread needs to know that it can rely on the object to have a suitable beginning place. There must be special contract between the Thread and the object whose instructions this Thread is to execute. The object promises to supply instructions; the Thread promises to execute them. (In the case of the AnimatorThread, it promises

to execute these instructions over and over again.) As we know, such a contract is specified using a Java interface. This interface defines a method containing the instructions that the Thread will execute. The Thread will begin its execution at the instructions defined by this method.

Implementing Animate

If we use an AnimatorThread to animate our object, our object must fulfill the specific contract on which AnimatorThread begins. This contract is specified by the interface Animate:

```
public interface Animate
{
    public abstract void act();
}
```

The Animate interface defines only a single method, void act(). A class implementing Animate will need to provide a body for its act() method, a set of instructions for how that particular kind of object act()s. An AnimatorThread will call this act() method over and over again, repeatedly asking the Animate object to act().

For example, the Timer that we described above could be implemented just as the Counter, but with the addition of an act() method:

```
public void act()
{
    this.increment();
}
```

Of course, we'd also have to declare that Timer implements the Animate interface. It isn't enough for Timer to have an act() method; we also have to specify that it does so as a commitment to the Animate interface. Here is a complete Timer implementation:

```
public class Timer implements Animate
{

    private int currentValue;

    public Timer()
    {
        this.reset();
    }

    public void increment()
    {
        this.currentValue = this.currentValue + 1;
    }

    public void reset()
    {
        this.currentValue = 0;
    }

    public int getValue()
```

```
        {
            return this.currentValue;
        }

        public void act()
        {
            this.increment();
        }

    }
```

Note that the implementation is entirely identical to the implementation of Counter except for the clause `implements Animate` and Timer's `act()` method. [Footnote: As we shall see in the next chapter, we could significantly abbreviate this class by writing it as

```
public class Timer extends Counter implements
Animate, Counting
{

    public void act()
    {
        this.increment();
    }

}
```

]

Now `Timer tick = new Timer();` defines a Timer ready to be animated.

AnimatorThread

On the other side of this contract is the instruction follower, an AnimatorThread. Like any other kind of Java object, a new AnimatorThread is created using an instance construction (`new`) expression and passing it the information required by AnimatorThread's constructor. The simplest form of AnimatorThread's constructor takes a single argument, an Animate whose act() method the new AnimatorThread should call repeatedly.

For example, we can animate a Timer by passing it to AnimatorThread's constructor expression:

```
Timer tick = new Timer();
AnimatorThread mover = new AnimatorThread( tick );
```

There is one more thing that we need to do before tick starts incrementing itself: tell the AnimatorThread to startExecution():

```
mover.startExecution();
```

An AnimatorThread's `startExecution()` is a very special method. It returns (almost) immediately. At the same time, the AnimatorThread comes to life and begins following its own instructions. That is, before the evaluation of the method invocation `mover.startExecution()`, there was only one

Thread running. At the end of the evaluation of the invocation, there are two Threads running, the one that followed the instruction `mover.startExecution()` and the one named `mover`, which begins following the instructions at `tick`'s `act()` method.

Once started, the AnimatorThread's job is to evaluate the expression `tick.act()` over and over again. Each time, this increments `tick`'s `currentValue` field. The AnimatorThread named `mover` calls `tick`'s `act()` method over and over again, repeatedly causing `tick` to `act`.

We can collapse the two AnimatorThread statements into one by writing

```
new AnimatorThread( tick ).startExecution();
```

However, this form does not leave us holding onto the AnimatorThread, so we couldn't later tell it to suspendExecution(), resumeExecution(), or stopExecution(). (See below.) If we anticipate needing to do any of these things, we should be sure to hold on to the AnimatorThread (using a label name).

Creating the AnimatorThread in the Constructor

If our Timers will always start ticking away as soon as they are created, we can include the Thread creation in the Timer constructor:

```
public class AnimatedTimer implements Animate
{

     private int currentValue;
     private AnimatorThread mover;

     public AnimatedTimer()
     {
         this.reset();
         this.mover = new AnimatorThread( this );
         this.mover.startExecution();
     }

     public void increment()
     {
     // ... rest of class is same as Timer
```

In this case, as soon as we say

```
Timer tock = new AnimatedTimer();
```

`tock` will begin counting away. If we invoke `tock.getValue()` at two different times -- even if no one (except its own AnimatorThread) asks `tock` to do anything at all in the intervening time -- the second value might not match the first. This is because `tock` (with its AnimatorThread) can act without needing anyone else to ask it.

Here is another class that could be used to monitor a Counting (such as a Counter or a Timer):

```
public class CountingMonitor implements Animate
{

     private Counting whoToMonitor;
     private AnimatorThread mover;

     public CountingMonitor( Counting whoToMonitor )
     {
         this.whoToMonitor = whoToMonitor;
         this.mover = new AnimatorThread( this );
         this.mover.startExecution();
     }

     public void act()
     {
         Console.println( "The timer says "
                          + this.whoToMonitor.getValue() );
     }

}
```

Note in the constructor that the first whoToMonitor (this.whoToMonitor) refers to the field, while the second refers to the parameter.

A Generic Animate Object

The way that AnimateTimer and CountingMonitor use an AnimatorThread is pretty useful. There is a cs101 class, AnimateObject, that embodies this behavior. It is probably the most generic kind of animate object that you can have; any other animate object would behave like a special case of this one. We present it here to reinforce the idea of an independent animate object. It generalizes both CountingMonitor and AnimateTimer.

At this point, you should regard this class as a template. Change its name and add a real act() method to get a real self-animating object. In the chapter on Inheritance, we will return to this class and see that there is a way to make this template quite useful directly.

```
public class AnimateObject implements Animate
{

     private AnimatorThread mover;

     public AnimateObject()
     {
          this.mover = new AnimatorThread( this );
          this.mover.startExecution();
     }

     public void act()
     {
         // what the Animate Object should do repeatedly
     }

}
```

It is worth noting that an Animate need not be animated by an AnimatorThread. For example, a group of Animates could all be animated by a single SequentialAnimator that asks each Animate to act(), one at a time, in turn. No Animate could act() while any other Animate was mid-act(). Each would have to wait for the previous Animate to finish. This SequentialAnimator would require only a single instruction follower (or Thread) to execute the sequential Animates' instructions, because it would execute them one act() method at a time. When one animate is acting, no one else can be.

The nature of execution under such a synchronous assumption would be very different from executions in which each Animate had its own Thread and they were all acting simultaneously. Roughly it's the difference between a puppet show with one not-very-skillful puppeteer, who can only operate a single puppet at a time, and a whole crowd of puppeteers each operating a puppet. The potential for chaos is much greater in the second scenario, but so is the potential for exciting interaction. When each object has its own AnimatorThread -- as in the AnimateObject template -- any other Animate (or the methods it calls) can execute at the same time.

More Details

This section broadens the picture painted so far.

AnimatorThread Details

The AnimatorThread class and the Animate interface reside in the package cs101.lang. This means that any file that uses these classes should have the line

```
import cs101.lang.*;
```

before any class or interface definition.

The class AnimatorThread specifies behavior for a particular kind of instruction follower. Its constructor requires an object that implements the interface cs101.lang.Animate, the object whose act() method the AnimatorThread will repeatedly execute.

After constructing an AnimatorThread, you need to invoke its startExecution() method. [Footnote: AnimatorThread's instances also have a startExecution() method that is identical to the startExecution() method. This is for historical reasons.] This causes the AnimatorThread to begin following instructions. In particular, the instructions that it follows say to invoke its Animate's act() method, then wait a little while, then invoke the Animate's act() method again (and so on). To temporarily suspend execution, use the AniamtorThread's suspendExecution() method. Execution may be restarted using resumeExecution(). To permanently terminate execution, AnimatorThread has a stopExecution() method. Once stopped, an AnimatorThread's execution cannot be restarted. However, a new AnimatorThread can be created on the same Animate object.

An object -- like an Animate -- is a set of instructions -- or methods -- plus some state used by these instructions. There is nothing to prevent more than one Thread from following the same set of instructions at the same time. For example, it would be possible to start up two AnimatorThreads on the same Timer. If the two AnimatorThreads took turns fairly and evenly, one AnimatorThread would always move from an odd to an even numbered currentValue, while the other would always move from an even to an odd numbered value. Of course, there's nothing requiring that the two AnimatorThreads play fair. Like

children, one might take all of the turns -- incrementing the Timer again and again -- while the other might never (or rarely) get a turn. AnimatorThreads are designed to minimize this case, but it can happen. The problem is more prevalent with other kinds of Threads.

One of the ways in which AnimatorThread tries to "play fair" is in providing intervals between each attempt to follow the act() instructions of its Animate object. The AnimatorThread has two values that it uses to determine the minimum interval between invocations of the Animate's act() method and the maximum interval. Between these two values, the actual interval is selected at random each time the AnimatorThread completes an act(). You can adjust these parameters using setter methods of the AnimatorThread. Values for these intervals may also be supplied in the AnimatorThread's constructor. See the AnimatorThread sidebar for details.

class AnimatorThread

AnimatorThread is a cs101 class (specifically, cs101.lang.AnimatorThread) that serves as a special kind of instruction-follower. An AnimatorThread's constructor must be called with an instance of cs101.lang.Animate. The AnimatorThread repeatedly follows the instructions in the Animate's act() method.

An AnimatorThread is an object, so it can be referred to with an appropriate (label) name. It also provides several useful methods:

> `void startExecution()` causes the AnimatorThread to begin following the instructions at its Animate's act() method. Once started, the AnimatorThread will follow these instructions repeatedly at semi-random intervals until it is stopped or suspended
>
> `void stopExecution()` causes the AnimatorThread to terminate its execution. Once stopped, an AnimatorThread cannot be restarted. This method may terminate execution abruptly, even in the middle of the Animate's act() method.
>
> `void suspendExecution()` causes the AnimatorThread to temporarily suspend its execution. If the AnimatorThread is already suspended or stopped, nothing happens. If the AnimatorThread has not yet started and is started before an invocation of resumeExecution(), it will start in a suspended state, i.e., it will not immediately begin execution. This method will not interrupt an execution of the Animate's act() method; suspensions take effect only between act()s.
>
> `void resumeExecution()` causes the AnimatorThread, if suspended, to continue its repeated execution of its Animate's act() method. If the AnimatorThread is not suspended or already stopped, this method does nothing. If the AnimatorThread is suspended but not yet started, invoking resumeExecution() undoes the effect of any previous suspendExecution() but does not startExecution().

Between calls to the Animate's act() method, the AnimatorThread sleeps, i.e., remains inactive. The duration of each of these sleep intervals is randomly chosen to be at least `sleepMinInterval` and no more than `sleepMinInterval + sleepRange`. These values are by default set to a range that allows for variability and slows activity to a rate that is humanly perceptible. If you wish to change these defaults, they may be set either explicitly using setter methods or in the AnimatorThread constructor.

`void setSleepRange( long howLong )` sets the desired variance in sleep times above and beyond sleepMinInterval

`void setSleepMinInterval( long howLong )` sets the range of variation in the randomization

By setting sleepRange to 0, you can make your AnimatorThread's activity somewhat more predictable as it will sleep for approximately the same amount of time between each execution of the Animate's act() method. Setting sleepMinInterval to a smaller value speeds up the execution rate of the AnimatorThread. Setting it to 0 can be dangerous and should be avoided. If sleepRange is 0, it is possible that this AnimatorThread will interfere with other Threads' ability to run.

AnimatorThread supplies a number of constructors. The first requires only the Animate whose act method supplies this AnimatorThread's instructions:

```
AnimatorThread( Animate who )
```

The next two constructors incorporate the same functions as setRange and setMinInterval:

```
AnimatorThread( Animate who, long sleepRange )

AnimatorThread( Animate who, long sleepRange,
long sleepMinInterval )
```

It is also possible to specify explicitly whether the AnimatorThread should start executing immediately. By default, it does so. The following constructor allows you to override this explicitly using the boolean constants `AnimatorThread.START_IMMEDIATELY` and `AnimatorThread.DONT_START_YET`.

```
AnimatorThread( Animate who, boolean startImmediately )
```

Finally, there are two additional constructors that incorporate both startup and timing information:

```
AnimatorThread( Animate who, boolean startImmediately,
long sleepRange )

AnimatorThread( Animate who, boolean startImmediately,
long sleepRange, long sleepMinInterval )
```

Delayed Start and the init() Trick

It is awfully convenient to be able to define an animate object as an Animate that creates and starts its own AnimatorThread. This hides the Thread creation and manipulation inside the Animate (as in the example of AnimateTimer), making it appear to be a fully self-animating object from the outside. However, sometimes we need to separate the construction of the Animate and its AnimatorThread from the initiation of the AnimatorThread instruction follower. That is, we want the AnimatorThread set up, but not yet actually running. For example, we might need a part that isn't yet available at Animate/AnimatorThread creation time. On these occasions, it would be awkward to start the execution of an AnimatorThread in the

constructor of its Animate. For example, if the Animate's act() method relies on other objects and these other objects may not yet be available, you wouldn't want the AnimatorThread to start executing the act() method yet.

An example of this might be in the StringTransformer class in the first interlude, in which you can't read or transform a String until after you've accepted an input connection. Since the input connection might not be available at StringTransformer construction time, one solution to this problem is to delay the starting of the execution of the act() method until after the input connection has been accepted. Once the constructor completes, the newly constructed object's acceptInputConnection method can be invoked. At this point -- and not before -- the AnimatorThread's startExecution() method can be invoked. This means that the call to the AnimatorThread's startExecution() method can't appear in the constructor. But it can't be invoked by any object other than the Animate, because the AnimatorThread is held by a private field of the Animate.

This situation -- that there are things that need to be done that are logically part of the setup of the object, but that cannot be done in the constructor itself -- is a common one. To get around it, there is a convention that says that such objects should have init() methods. Whoever is responsible for setting up the object should invoke its init() method after this setup is complete. The object can rely on the fact that its init() method will be invoked after the object is completely constructed and -- in this case -- connected. We could then put the call to the AnimatorThread's startExecution() method inside this init() method.

Here is a delayed-start version of the AnimateObject template.

```
public class InitAnimateObject implements Animate
{

     private AnimatorThread mover;

     public InitAnimateObject()
     {
          this.mover = new AnimatorThread( this );
     }

     public void init()
     {
          this.mover.startExecution();
     }

     public void act()
     {
         // what the Animate Object should do repeatedly
     }

}
```

A concrete example of this issue arises if we look at CountingMonitor and don't assume that the Counting will be supplied to the constructor. Here is another version of CountingMonitor without the constructor parameter:

```
public class InitCountingMonitor implements Animate
{
```

```
        private Counting whoToMonitor;
        private AnimatorThread mover = new AnimatorThread( this );

        public void setCounting( Counting whoToMonitor )
        {
            this.whoToMonitor = whoToMonitor;
        }

        public void init()
        {
            this.mover.startExecution();
        }

        public void act()
        {
            Console.println( "The timer says "
                             + this.whoToMonitor.getValue() );
        }

    }
```

The use of a method named init() here is completely arbitrary. You are free to define your own method and call it whatever you want. However, you will see that many people follow this convention and provide an init() method for their objects when there is initialization that must take place after the constructor and setup process is complete.

Threads and Runnables

The Animate/AnimatorThread story that we've just seen is not a standard part of Java, though it is only a minor variant on something that is. There are two reasons why we've used AnimatorThreads here. The first is that most of the self-animating object types in this book are objects whose act method is executed over and over again. AnimatorThread is a special kind of Thread designed to do just that. The second is that AnimatorThread contains some special mechanisms to facilitate its use in applications where you might want to suspend and resume its execution or even to stop it entirely. AnimatorThread provides methods supporting this behavior.

There is, however, in Java a more primitive type of Thread, called simply Thread. Like an AnimatorThread, a simple Java Thread can be given an object to animate when the Thread is created. (Its constructor takes an argument representing the object whose instructions the Thread is to follow once it has been started.) However, the Thread does not execute this method repeatedly; it executes it once, then stops. The contract that a Thread requires of the object providing its instructions is not Animate, meaning it can be called on to act repeatedly. Instead, it is Runnable, meaning it can be executed once.

Thread (as of Java 1.1) does not provide suspension, resumption, or cessation methods. In this book, we avoid the use of plain Java Threads.

In addition, it is technically possible in Java to extend a Thread object rather than passing it an independent Runnable. Except in code that creates special kinds of Threads (such as AnimatorThread) capable of animating other objects, the extending of Thread is highly discouraged in this book. Extending Thread to

create an executing object (whose own run() method is the set of instructions to be followed) confounds the notion of an executor with the executed.

Thread Methods

Thread methods

Threads are Java's instruction followers. In this book, we will most often make use of AnimatorThreads. However, it is useful to understand how Java's built-in Thread class works as well.

Like an AnimatorThread, each Thread provides a few methods for its management.

`void start()` Like AnimatorThread's startExecution(), this method causes the target Thread to begin following instructions. If the Thread's constructor was supplied a Runnable, the Thread begins execution at this Runnable's run() method. When the run() method terminates, the Thread's execution is finished.

`boolean isAlive()` tells you whether the target Thread is alive, i.e., has been started and has not completed its execution.

`void interrupt()` sends the target Thread an InterruptedException. Useful if that Thread is sleeping, waiting, or joining.

`void join()` causes the invoking Thread to suspend its execution until the target Thread completes. Variants allow time limits on this suspension: `void join( long millis )` and `void join( long millis, long nanos )`.

Unlike AnimatorThread, a Thread cannot safely be stopped, suspended, or resumed.

In addition to its role as the type of Java's instruction followers, the Thread class provices useful static (i.e., class-wide) functionality. These methods are static methods of the class Thread:

`static void sleep( long millis )` causes the currently active Thread to stop executing for millis milliseconds. This method throws InterruptedException, so it cannot be used without some additional machinery (introduced in the chapter on Exceptions). There is a variant method, `sleep( long millis, long nanos )` that allows more precision in controlling the duration of the Thread's sleep.

`static void yield()` is intended to pause the currently executing Thread and to allow another Thread to run. However, not all versions of Java implement Thread.yield in a way that ensures this behavior.

Other Thread features are outside the scope of this course.

Where do Threads come from?

We have discussed the idea of AnimatorThreads above, showing how to create self-animating objects by having an AnimatorThread created in an object's constructor. Such an object is born running; it continually acts, over and over, until its Thread is suspended or stopped.

In fact, no execution in Java can take place without a Thread. But something must call the AnimatorThread constructor; this instruction must be executed by a Thread! So where does the first Thread come from? This depends on the particular kind of Java program that you are running. In this book, we look primarily at Java applications. In the appendix, we also answer these questions for Java applets.

Starting a Program

What does it mean for a Java program to run? It means that there is an instruction follower that executes the instructions that make up this program. In Java, there is no execution without a Thread, or instruction-follower, to execute it. So when a program is run, some Thread must be executing its instructions. Where does this Thread come from, and how does it know what instructions to execute?

Let's answer the first of these questions first. When a Java program is run, a single Thread is created and started. This is not a Thread that your program creates; it is the Thread that Java creates to run your program. Depending on whether your Java program is an application (as we're discussing in this book) or an applet (as you may have encountered on the world-wide web) or some other kind of Java program, there are different conventions as to where this Thread begins its execution. But running a program *by definition* means creating a Thread -- an instruction follower -- to execute that program.

How does the Thread know where to begin? By convention. What do we mean by a convention? AnimatorThread's use of Animate is a convention. This convention is, in some sense, completely arbitrary. That is, a different interface name or other method might have been used. For example, the raw Thread class uses a different convention, that of Runnable/run(). If you were to design your own type of Thread, you could create a different convention for it to follow. However, once these names and contracts have been selected by the designers of AnimatorThread and Thread, they are absolute rules that cannot be violated.

Similarly, there must be some arbitrary convention as to how a Java program begins. In a standalone application, the convention is that running a Java program means supplying a class to the executable, and by convention a particular method of the class is always the place that execution begins. This default execution does not create an instance of the class, so the method must be a static one. Again by convention, the name of this method is main, it takes as argument an array of Strings, and it returns nothing. That is, the arbitrary but unvarying start point for the execution of a standalone Java application is the

```
public static void main ( String[] args )
```

method of the class whose name is supplied to the executable.[Footnote: Typically, this means the class you select before choosing run from the IDE menu or the class whose name follows the command `java` on the command line.]

So if you want to write a program, you simply need to create a class with a method whose signature matches the line above. The body of that main method will be executed by the single Thread that is created at the beginning of a Java execution. When execution of main terminates, the program ends. If you do not want the program to end, you need to do something during the course of executing main that causes things

to keep going. Typically, this means that you use the body of main to create one or more objects that themselves may execute. For example, if the body of main creates an animate object (with its own AnimatorThread), then that object will continue executing even if the body of main is completed. This is called "spawning a new Thread".

Here is a very simple class that exists solely to create a new instance of the AnimateTimer class:

```
public class Main
{

     public static void main ( String[] args )
     {
         Counting theTimer = new AnimateTimer();
     }

}
```

This program simply counts. The instruction follower that begins when this program starts up (e.g., using java Main) executes the main() method, invoking new AnimateTimer() and assigning the result to theTimer. This Thread is now done executing and stops. However, the constructor for AnimateTimer has created a new AnimatorThread and then called that AnimatorThread's startExecution() method. This starts up the new Thread which repeatedly calls AnimateTimer's act() method. The program as a whole will not terminate until the AnimatorThread stops executing, which it will not do by itself. If you run this program, you will need to forcibly terminate it from outside the program!

Since we didn't give this program any way to monitor or indicate what's going on, running it wouldn't be very interesting. But we can use the CountingMonitor above to improve this program:

```
public class Main
{

     public static void main ( String[] args )
     {
         Counting theTimer = new AnimateTimer();
         Animate theMonitor = CountingMonitor( theTimer );
     }

}
```

Q. Can you find a more succinct way to express the body of the main method?

Q. What will be printed by this program? On what does it depend? (Hint: fairness.)

The instruction follower executing the Main class's main method exits. However, before it completes it executes the instructions to create and start two separate AnimatorThreads. These AnimatorThreads continue after the execution of the main Thread exits. Again, this program must be forcibly terminated from outside.

Q. Can you cause this program to stop by itself sometime after it has counted to 100? (This is a bit tricky.)

The two versions of the Main class above each contain just the instructions to create an instance or two. In the cs101 libraries, we have provided a Main that does this for you. This allows you to write applications without needing to write public static void main(String[]) methods yourself.

class Main

The cs101 libraries include a class, cs101.util.Main, that can be run from the java command line to create an instance of a single class with a no-args constructor. For example, we could implement the unmonitored Timer example using the following command:

```
java  cs101.util.Main  AnimateTimer
```

This causes code much like the first Main class to execute, creating a single instance of AnimateTimer (using its no-args constructor).

The class cs101.util.Main contains nothing but the single static method main (taking a String[] argument). The command above tells Java to start its initial instruction follower at this method -- the static main(String[]) method of the class cs101.util.Main. The remainder of the information on the command line (in this case, AnimateTester) is supplied to the main method using its parameter.[Footnote: For more detail on arrays ([]), see the chapter on Dispatch.]

Style Sidebar

Using main()

If you do decide to write your own main() method, you should do so in a class separate from your other classes, generally one called Main and containing only the single public static void main() method requiring a String[] (i.e., an array of Strings). This method may have some complexity, creating several objects and gluing them together, for example.

Alternately, you can create an extremely simple main method in any (or even every) class that you write. In this case, however, the main method should do nothing more than to create a single instance of the class within which it is defined, using that class's no-args constructor. Of course, the signature of each main method is the same: public static void main(String[] args) The main that will actually be executed is the one belonging to the (first) class whose name is supplied to the java execution command. So, for example, in the sidebar on class Main, we said

```
java  cs101.util.Main  AnimateTimer
```

causing cs101.util.Main's main method to be run.

The logic behind these restrictions on the use of main() is as follows. In the second case -- main in many instantiable class's files -- the presence of main allows that object to be tested independently. However, this test is extremely straightforward and predictable. If the main method takes on any additional complexity, it should be separated from the other (instantiable) classes and form its own resource library, one that exists solely to run the program in all its complexity.

Why Constructors Need to Return

In the code above, each Animate's constructor calls the startExecution() method of a new Thread. This in turn repeatedly calls the act() method of the Animate. Why doesn't the constructor just repeatedly call the Animate's act() method itself (e.g., in a while loop)?

This is a fundamental issue. If the Animate's constructor called the act() method itself, the instruction follower -- or Thread -- executing the constructor would be trapped forever in a loop calling act() over and over. The constructor invocation -- the new expression -- would never complete. In the monitored counting example, the invocation of AnimateTimer's constructor would cause the instruction follower to execute the act() method of AnimateTimer over and over again. This instruction follower -- the only instruction follower to be running so far -- would never complete the repeated execution of the act() method. This means that it would never get around to creating the CounterMonitor.

This is why AnimatorThread.startExecution() must be a very special kind of method. The Thread, or instruction follower, that executes startExecution() must return (almost) immediately. It is the *new* Thread, the one just started, that goes off to execute the act() method. The original Thread returns from this invocation and goes about its business just as if nothing ever happened. In personal terms, this is the difference between doing the job yourself and assigning someone else to do it. True, when someone else does it you have less control over how or when the job gets done; but while someone else is working on it, you can be doing something else.

Chapter Summary

- In Java, activity is performed by instruction followers called Threads.
- An animate object is simply one that has its very own Thread.
- An AnimatorThread is a useful kind of Thread that repeatedly follows the instructions provided by some object's act() method.
    - This object must implement the Animate interface.
    - It must be supplied to the AnimatorThread's constructor.
- An AnimatorThread can also be asked to start, stop, suspend, or resume execution.
- Java programs may involve other Threads.
    - One Thread begins execution at public static void main(String[] args) when a Java application is begun.
    - GUI objects involve their own Threads.
    - Other Threads may be explicitly created.

Exercises

1. Define a class whose instances each have an internal value that doubles periodically. Each time that the value doubles, the instance should print this new value to the Console.
2. Define a class that periodically reads from the Console and writes the value back to the Console.
3. Define a main class that creates three instances of your doubler.

4. Using the timing parameters of AnimatorThread, demonstrate that not all doublers have to run at the same rate.

Questions or comments:
<webmaster@cs101.org>