

14

Timing Framework: Fundamentals

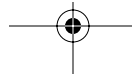
Introduction

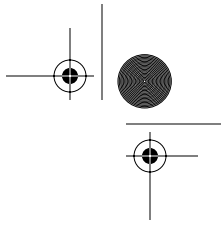
When you first start working with animations in Swing, you quickly realize two things about the built-in timers:

- Their simplicity makes building any kind of animation possible.
- Their simplicity makes building any kind of animation incredibly difficult.¹

That is, with a time-based callback mechanism like `javax.swing.Timer`, you can perform any time-based task, such as varying Swing component characteristics over time, and thus animate your GUI. But the details of implementing such animations are prohibitively tedious, so most developers skip this step and opt for the static component behavior that most GUI toolkits provide by default.

1. The timers illustrate the classical trade-off between simplicity and power. Sure, the ancient Egyptians could build the pyramids and Sphinx with just a bunch of rocks, but it took monumental efforts and lots of cheap labor to get it done. The English seem to have taken a more practical stance on the problem in constructing Stonehenge and similar monuments, where the finished product is just a small pile of large stones. The end result is not quite as impressive, but it must have been a far sight easier to build than the pyramids. The same trade-off is made in most “rich client” applications today: Applications either skip animations entirely or implement only rudimentary animations because the process of building more powerful ones is so tedious and time-consuming and the software industry lacks the cheap labor that abounded in ancient Egypt.





This problem became evident when I first started experimenting with animations in Swing and Java 2D. I just kept writing the same boilerplate code over and over again to get the basic functionality that all of my animations required. This experience was the inspiration for the Timing Framework.

**ONLINE
LIBRARY**

Note: The Timing Framework is a library that is being developed in a project at <http://timingframework.dev.java.net>. We have taken a specific version of the library and put it on the book's Web site so that all text in the book and all code in the book's demos match the version of the framework available there. So if you want to use the version of the framework that we discuss in the book, use the one on the book's Web site. If you want to use the latest/greatest version of the library, or you just want to see what's happening with ongoing development of the framework, check out the project site on java.net.

The Timing Framework is a set of utility classes that provides a much more capable animation system that handles many of the details that you would otherwise need to implement in your application. The purpose of the library is to enable you to create powerful animations in Swing without worrying about the low-level implementation details.

The motivations for all of the features in the Timing Framework were twofold:

- *Handle common tasks:* Much of the code that we write in animating graphics and GUIs is necessary for nearly all animations. For example, most time-based animations need to figure out what fraction of the animation has completed at any given time during the animation, so why not simplify things by calculating that fraction automatically?
- *Simple API:* In providing more capabilities for animations, we do not want to create an API that is prohibitively complex. It should be as easy to use as possible.

The framework has a few distinct levels of functionality. At the core of the framework is the `timing` package, with the fundamental building blocks that all of the other pieces use. This group of classes provides the equivalent of the built-in `Timers` but with significantly increased functionality. We cover this functionality in this chapter.

An additional level of functionality is provided in the `triggers` and `interpolation` packages. Triggers associate animations with specific events and automate starting animations on the basis of those events. Property setters in the `interpolation` package provide the ability to animate properties of Java objects and to define complex models of how those properties are interpolated between different val-



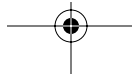
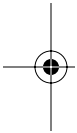


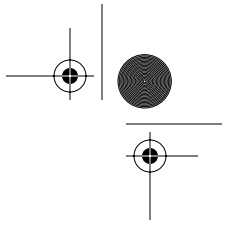
ues. We cover triggers and property setters in Chapter 15, “Timing Framework: Advanced Features.”

Core Concepts

Several key concepts and properties are embedded in the central classes used by the Timing Framework:

- *Animator*: This class encapsulates most of the functionality discussed in this chapter, but it is worth discussing it separately so that we can see how an *Animator*, and the animation it defines, is created and run.
- *Callbacks*: An application must have a means to be notified of events during the animation. In this way, an application can be involved in the animation to perform appropriate actions based on the state of the animation. Event notification is handled through callbacks to an interface that application code may implement. This mechanism is similar to what we saw earlier in our discussion of the built-in Java timers, except that the Timing Framework has more callbacks with more information to enable more flexibility in your animations.
- *Duration*: The duration value defines the length of time that the animation will last. An animation stops automatically when this duration has elapsed. You may also specify that an animation should run indefinitely.
- *Repetition*: Some animations are intended to run once and then finish. Others may run indefinitely. Still others may run with a finite duration and then repeat when they are done.
- *Resolution*: The resolution of an animation controls the frame rate of the animation. This concept was discussed at length in Chapter 12, “Animation Fundamentals.”
- *Starting behavior*: An animation may not want to start with the default behavior of moving forward from the beginning. It may instead want to run backwards or start from some other point than the beginning. It may also want to delay for some time before starting.
- *Ending behavior*: By default, an animation holds its final value when it is stopped. You might choose, instead, to have an animation reset to the start state when finished.
- *Interpolation*: The easiest kind of interpolation is linear interpolation, which we discussed in earlier chapters. But there are other kinds of interpolation that we can apply to give the animation nonlinear behavior.





Animator

`Animator` is the core class of the entire framework. Users of the Timing Framework create `Animators` with information that details the animations they want to run. The properties that define an animation are set through a combination of the constructors, which enable setting the most common properties, and other methods in the class. Animations are started and stopped by calling methods in this class. Finally, this class is responsible for issuing ongoing animation events while the animation is running, which is discussed in the next section, “Callbacks.”

Creation

`Animators` are created through one of the three constructor methods:

`Animator(int duration)`

This method takes only a duration parameter, discussed later, which controls how long the animation will run. Note that this constructor takes no `TimingTarget` parameter. Callers would typically add one or more `TimingTarget` objects later via the `addTarget()` method; otherwise the `Animator` will run but will issue no events.²

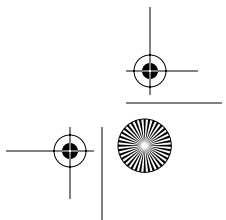
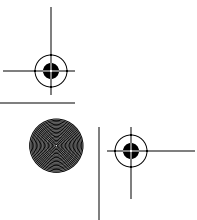
`Animator(int duration, TimingTarget target)`

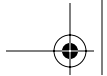
This variant, the most common, takes a duration and a `TimingTarget`. The `TimingTarget`, discussed in the section “Callbacks,” contains the methods that will be called with animation events as the animation runs.

`Animator(int duration, double repeatCount, Animator.RepeatBehavior repeatBehavior, TimingTarget target)`

This final variant takes the same duration and target parameters as before, but also takes two other parameters that control how the animation is repeated over time, as discussed in the section “Repetition.”

2. This is the animation equivalent of the great woody philosophical question: “If a tree falls in the forest and no one is around to hear it, does it make a sound?” The parallel question for `Animator` might be, “If an `Animator` runs and there is no `TimingTarget` around to receive the events, does it do anything?” We may never really know.





Control Flow

There are several methods that control the running and stopping of the animation. Note that, as discussed in the section “Duration,” animations may stop automatically. But animations may also be programmatically halted by some of the methods of `Animator` described here.

void start()

This method starts the animation, which results in callbacks to the `TimingTarget.begin()` and `timingEvent()` methods, as described in the section “Callbacks.”

void stop()

This method stops the animation, which results in a call to the `end()` method of `TimingTarget`, notifying any targets that the animation has completed.

void cancel()

This method is like `stop()` except that `TimingTarget.end()` will not be called for any targets. It’s like pulling the plug on the animation instead of letting it stop normally.

void pause()

This method pauses a running animation, which stops the animation in its current state until and unless a later call to `resume()` is issued.

void resume()

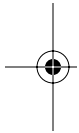
This method resumes an animation that has been paused. The animation will continue from its previously paused state, as if no time had passed between `pause()` and `resume()`. This method has no effect on an animation that has not been paused.

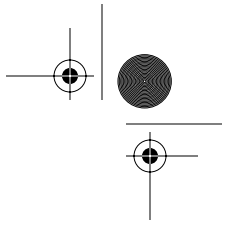
boolean isRunning()

This method queries whether the animation is currently running.

Controlling a Running Animation

It is worth noting that most of the parameters that control an animation, such as the duration and repetition parameters, make sense only on a non-running animation. Once an animation is running, it is not clear how changes to these





parameters should be interpreted. Therefore, most of the methods of `Animator` that set these parameters, except where noted, will throw an exception if called while an animation is running.

Callbacks

TimingTarget

Animations in the Timing Framework run by having the animation code in `Animator` call back into one or more `TimingTarget` implementations. A `TimingTarget` object exists to receive timing events from an `Animator`. The `TimingTarget` object is the connection between the animation running through `Animator` and the animation actually doing something. The callback methods in `TimingTarget` are given information about the current animation state and can set up object state, calculate new property values, or do anything else appropriate for the situation.

When you set up an `Animator`, you give it one or more `TimingTarget` objects through one of the constructors for `Animator` and through the `addTarget()` method of `Animator`. As the animation runs, the `Animator` object calls the methods in each of its `TimingTarget` objects.

The `TimingTarget` interface has four different event methods to implement:

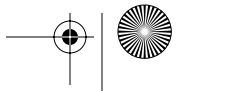
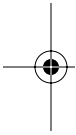
```
public interface TimingTarget {
    public void begin();
    public void end();
    public void repeat();
    public void timingEvent(float fraction);
}
```

void begin()

This method is called by `Animator` when the animation is first started. It allows the timing target to perform any necessary setup prior to running the animation.

void end()

This method is called when the animation is finished, either because the animation completed naturally by running for the specified duration and number of repetitions or because the `stop()` method was called on this target's `Animator`. This method allows the target to perform any necessary cleanup operations. The `end()` method can be used as a mechanism to help sequence animations





together. For example, a target can use the `end()` call to signal that some other dependent animation should start. Note, however, that triggers provide an even easier mechanism for this functionality, as we see in Chapter 15.

void repeat()

This method is called during a repeating animation, every time the animation begins another repetition. Repeating animations are discussed later.

void timingEvent(float fraction)

This method is the most important method in this interface and, in fact, in the entire framework.³ `timingEvent()` provides the target with the fraction, from 0 to 1, of the animation that has elapsed. The target can then use this information to change whatever properties need to be changed during the animation and to schedule a repaint if necessary.

The fraction value is directly related to the `duration` property. If `Animator` is given a duration of 2 seconds, then an animation that issues a `timingEvent()` one second after starting would call `timingEvent()` with a fraction value of `.5`.⁴

The fraction is a useful value to have. If you want to animate some variable linearly between start and end values, it is important to know what fraction of the animation has elapsed. If the animation is halfway through, then you know to set your variable to halfway between its start and end values.

Some of the parameters of `Animator`, such as the start direction and the reversing behavior, may make an animation run backwards. When this happens, the fraction values received in `timingEvent()` run in reverse, too. That is, the fraction always represents the elapsed fraction of the animation from the start to the end. An animation running in reverse starts at the end point and runs in reverse. So, for example, an animation that starts at the end and runs in reverse will issue values from 1 down to 0.

3. In fact, this method was the original inspiration for the entire framework. I just got so tired of recoding the same “how much of my animation has elapsed?” logic in every animation, it seemed like a much easier mechanism was called for—one that would have the timer give me the fraction instead of my having to compute it by querying the system time and calculating it from starting times, durations, and so on. So it doesn’t look like much, but the whole library grew from this one small method.

4. Note that some nondefault properties of `Animator`, such as a nonlinear `Interpolator` or a non-zero starting fraction, would change this simple example. We discuss these properties later.





TimingTargetAdapter

The `TimingTargetAdapter` class is a simple implementation of `TimingTarget`, providing empty methods for that interface. This class is provided as a utility for subclasses that want to receive only specific `Animator` events and do not want to implement all of the `TimingTarget` methods just to get the one or two that they really care about.

Duration

The discussion of `timingEvent()` relates directly to the `duration` property, because the fraction elapsed of an animation is calculated from the time elapsed so far and the total duration of that animation. The duration is specified in either one of `Animator`'s constructors, as seen previously, or in the following method:

```
setDuration(int duration)
```

Both the `setDuration()` method and `Animator`'s constructors set the duration for the animation in milliseconds. For example, an animation is assigned a duration of 2 seconds through a constructor, like this:

```
Animator myAnimation = new Animator(2000);
```

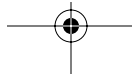
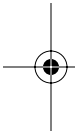
or through a later assignment to an existing animation:

```
myAnimation.setDuration(2000);
```

There is one additional, important value that a duration may have: `Animator.INFINITE`, which tells `Animator` that this animation should run indefinitely. Note that such animations will still call `timingEvent()` on a regular basis, but the fraction value in that call will be meaningless because there can be no elapsed percentage of an infinite amount of time.⁵

An important concept to note in relation to duration is that all animations are tracked in `Animator` in fractional time. That is, an animation, regardless of actual duration, may be thought of in terms of the percentage that the animation has elapsed at any time. So any animation, other than one of `INFINITE` duration, has a fractional duration of exactly 1. Calls to `timingEvent()` during the animation will use a fractional value instead of an actual duration. This mechanism tends to be easier to deal with for callers, which get more useful information

5. This is why, when you are in a meeting or lecture that seems to drag on forever, you keep looking at the clock and the minute hand has not moved at all. In fact, the meeting is of infinite duration and elapsed time has no meaning.





from knowing that an animation is one-quarter elapsed than that it is 500 ms into whatever its total duration may be. The concept of the elapsed fraction comes up often. We typically discuss animations in terms of this fraction instead of the total duration simply because that is what `Animator` keeps track of and reports to its targets, and because it is much more powerful and useful to `Animator`'s users.

Repetition

A repeated animation is a common pattern. Repetition can take the form of running the same animation over and over, like an indefinite progress bar whose status always crawls from the left to the right. A repeating animation can also be constantly reversing, like a pulsating button that has a glow effect in which the glow is constantly glowing toward full intensity and then dimming back down to some default state. Instead of constructing separate animations for each repetition or creating one large animation that handles all of the repetitions as an implementation detail, the framework provides the ability to define the core animation and then parameters for how that animation should be repeated.

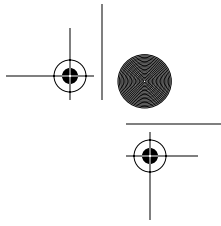
There are two properties of `Animator` that control repetition: the number of times the animation should be repeated and the behavior upon each repetition. These properties are controlled through the following constructor and methods:

```
Animator(int duration, double repeatCount,  
        Animator.RepeatBehavior repeatBehavior,  
        TimingTarget target)  
  
void setRepeatCount(double repeatCount)  
  
void setRepeatBehavior(Animator.RepeatBehavior repeatBehavior)
```

In this constructor and these methods, the repetition behavior is controlled through the `repeatCount` and `repeatBehavior` variables. `repeatCount` is simply the number of times that the animation should be repeated. This value can be fractional, such as 2.5, to indicate that the animation may stop partway through. `repeatCount` can also, like the `duration` value, take the value `Animator.INFINITE`, which indicates that the animation should repeat indefinitely.

`repeatBehavior` can have a value of either `RepeatBehavior.LOOP` or `RepeatBehavior.REVERSE`. `LOOP` repeats the animation in the same direction every time. When the animation reaches the end, it starts over from the beginning. So, for example, the animation fraction being passed into `timingEvent()` calls will go from 0 to 1, then 0 to 1, and so on, until `repeatCount` is reached or the animation is otherwise stopped. `REVERSE` creates an animation that reverses direction whenever it reaches the end of an animation. For example, the animation fraction passed into `timingEvent()` calls will go from 0 to 1, 1 to 0, 0 to 1, and so on.





Resolution

The resolution of `Animator` controls the amount of time between each call to `timingEvent()`. The default used by `Animator` is reasonable for most situations, so developers should not need to change the value in general, but changing it is a simple matter of calling `setResolution()`:

```
void setResolution(int resolution)
```

This method sets the number of milliseconds between each call to `timingEvent()`. Recall from our discussion of resolution in Chapter 12, “Animation Fundamentals,” that the actual resolution may be dependent on such factors as the internal timing mechanism being used and the runtime platform. The Timing Framework currently uses the Swing timer internally, and its resolution is thus constrained to the resolution of that timer for now.⁶

Start Behavior

There are three factors about the starting state of the animation that you can control: the start delay, the direction, and the initial fraction.

Start Delay

Some animations may wish to have an initial delay before commencing. The amount of this delay is controlled through the `setStartDelay()` method:

```
void setStartDelay(int startDelay)
```

where the `startDelay` value is in terms of milliseconds.

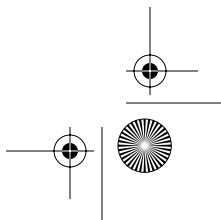
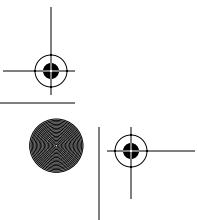
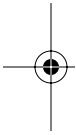
Start Direction

By default, an animation runs forward when it starts. The initial direction can be changed to run the animation in reverse instead. This setting is controlled through the `setStartDirection()` method:

```
void setStartDirection(Animator.Direction startDirection)
```

where `startDirection` can have the value of either `Direction.FORWARD`, which is the default behavior, or `Direction.BACKWARD`.

6. A late addition to the Timing Framework added the ability to use an external timer. So while the framework still uses the Swing timer by default, it is now possible to supply a timer with a different resolution. See the JavaDocs for `TimingSource` in the framework for more information, but note that most users should not need anything but the default timer.





Start Fraction

By default, an animation begins at fraction 0. This setting can be changed to start from any point during the animation by calling `setStartFraction()`:

```
void setStartFraction(float startFraction)
```

where `startFraction` is a value from 0 to 1, representing the fraction elapsed of the animation. Note that to run an animation from the end to the beginning, the caller should set the initial fraction to 1 and the direction to `BACKWARD`. An example of this behavior is shown in the `FadingButtonTF` demo later.

End Behavior

By default, an animation will hold its final value when it finishes. For example, an animation that finishes a normal forward cycle from 0 to 1 will hold the value 1 at the end. This can be changed to reset to 0 at the end instead by calling `setEndBehavior()`:

```
void setEndBehavior(Animator.EndBehavior endBehavior)
```

where `endBehavior` can have the value of either `EndBehavior.HOLD`, which is the default behavior, or `EndBehavior.RESET`, which sends out a final `timingEvent()` with a fraction of 0 at the end of the animation.

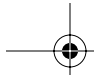
Demo: FadingButton Reprise

We're not quite done with the core `Animator` features. We still need to cover the important area of `Interpolator`. But it's time for a break to see some of the concepts in action.

Let's look at what we can do with just the classes that we have covered so far. We have many classes in the framework yet to cover, but the power and flexibility of just the basic `Animator` and `TimingTarget` classes provides enough to enable simple code that drives powerful animations. In particular, think of the things that we had to do with the built-in timers to animate our GUIs in previous chapters or the things that seemed unapproachably tedious, like cyclic, repeating animations.

For a simple example, let's revisit the `FadingButton` demo that we discussed Chapter 12. While the application is not terribly complex, it is a useful example for showing how using `Animator` helps make animations easier to program.





Recall in that example that we defined a custom JButton subclass with various methods for rendering the button translucently and animating the value of alpha. First, there were some instance variables to help track state:

```
float alpha = 1.0f;           // current opacity of button
Timer timer;                 // for later start/stop actions
int animationDuration = 2000; // animation will take 2 seconds
long animationStartTime;     // start time for each animation
```

In the constructor, we created the Timer object that ran the animation:

```
timer = new Timer(30, this);
```

Finally, we had an actionPerformed() method that served two purposes: It caught clicks on the button and used them to start and stop the animation, and it also received Timer events and animated the value of alpha with the following code:

```
public void actionPerformed(ActionEvent ae) {
    // ... code to handle button clicks not shown here ...

    long currentTime = System.nanoTime() / 1000000;
    long totalTime = currentTime - animationStartTime;
    if (totalTime > animationDuration) {
        animationStartTime = currentTime;
    }
    float fraction = (float)totalTime / animationDuration;
    fraction = Math.min(1.0f, fraction);
    // This calculation will cause alpha to go from 1 to 0
    // and back to 1 as the fraction goes from 0 to 1
    alpha = Math.abs(1 - (2 * fraction));
    repaint();
}
```

**ONLINE
DEMO**

Now that we have the power of Animator, let's see how the code changes. You can see and run the code for this version, called FadingButtonTF, on the book's Web site. First of all, we need fewer instance variables:

```
float alpha = 1.0f;           // current opacity of button
Animator animator;           // for later start/stop actions
int animationDuration = 2000; // each cycle takes 2 seconds
```

We do not need to track the animationStartTime because we no longer need to calculate the fraction of the cycle elapsed. Animator does this for us.





The constructor is similar to what it was before, although the declaration for `Animator` is a bit different from that of `Timer`:

```
animator = new Animator(animationDuration/2, Animator.INFINITE,  
                        RepeatBehavior.REVERSE, this);
```

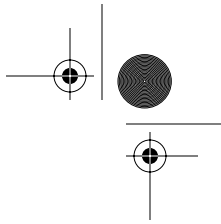
There are a few interesting bits about this call. First of all, we are using a duration of only half of `animationDuration`. This difference is because of how this new animation will be handled. Previously, each individual animation would consist of both the fade-out and fade-in portions, which we wanted to last for 2 seconds. With `Animator`, we can declare a more interesting reversing animation that reverses every second, which gives us the same result. Also, we see that we are going to be repeating infinitely, which is the same behavior as in the earlier `Timer` example. We declare a `REVERSE` behavior so that the animation reverses direction every time it repeats. And finally, we pass in `this` as the `TimingTarget` that will be called with timing events. Our object implements the `TimingTarget` interface in order to catch `timingEvent()` calls, just as the previous version of the demo received events from `Timer` in its `actionPerformed()` method.

Additionally, since we want to start at an opaque value and animate toward transparency, we need to make sure that we link up the animation fraction and our alpha value correctly. Both values vary between 0 and 1 over the course of the animation, so we're almost there. But since our animation starts at 0 by default and we want our alpha value to start at 1, or fully opaque, these values are going to run opposite of each other. We can either have alpha represent the inverse of the fraction, so that alpha would be 1 when fraction was at 0, or we can use additional facilities in `Animator` to start the animation at the end, playing backwards. This will ensure that the animation fraction starts at the same value as we want for our alpha. We add the following code to set the `Animator` properties accordingly:

```
animator.setStartFraction(1.0f);  
animator.setStartDirection(Direction.BACKWARD);
```

Finally, let's see the actual animation code for this new version of the demo. This time, the code is in the `timingEvent()` method, which is the target for `Animator`'s timing events, instead of the old `actionPerformed()` method. Compare the code in `actionPerformed()` to this approach for `timingEvent()`:

```
public void timingEvent(float fraction) {  
    alpha = fraction;  
    repaint();  
}
```



Note that we do not need to calculate the fraction elapsed of the animation, because it is given to us. Also, the flexibility in how we defined the animation, starting at the end and running backwards in the first animation, simplified our alpha calculation to simply equal the fraction itself. A screenshot of the application is seen in Figure 14-1.

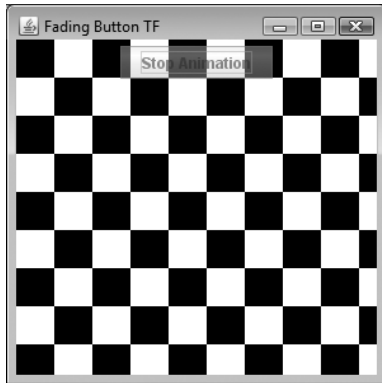


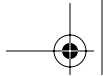
Figure 14-1 FadingButtonTF: same as the FadingButton demo, but with less code.

This new version of the demo behaves exactly like the old one, but with less code. The demo is, by design, very simple, so it does not really show off the power of the Timing Framework as much as the simpler code that is possible, even for very easy animations. But it would help for us to develop a more interesting demo that shows off more about the framework. We develop this demo, The Racetrack Demo, throughout this chapter and the next one so that you can see how the different elements of the framework work together to create more interesting and complex animations.

The Racetrack Demo

ONLINE
DEMO

This demo can be found on the book's Web site in the various projects ending with Race. There are several versions of the application that correspond to the different features of the framework that we discuss in this chapter and the next.



Background

First, we should explain how the application works in general. There are four main classes in the demo package in use in all of the versions of this application:

ControlPanel: This is the panel with the Go and Stop buttons that you can see at the bottom of the window in Figure 14-2. These buttons are added as listeners elsewhere in the application to start and stop the race appropriately.

TrackView: This is the part of the application that handles drawing the car in the current position and orientation on the track. Other code may call this class to set the position and rotation of the car, and the `paintComponent()` method of this class handles drawing the car appropriately.

RaceGUI: This class simply creates and manages the `TrackView` and `ControlPanel` objects inside a `JFrame`.

***Race:** Each variation of this demo that we will see is called `*Race`, according to what it demonstrates. For example, the first version we will see is called `BasicRace`. These classes handle the setup of the animation and the changing of the car properties during the animation.

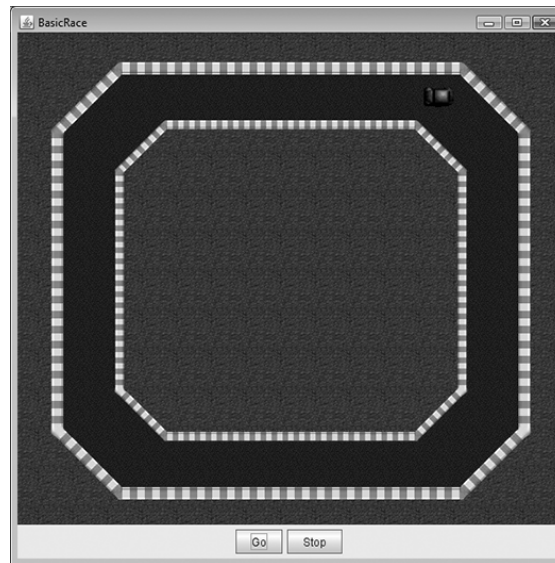
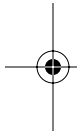
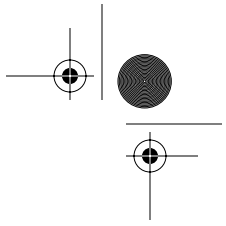


Figure 14-2 The Racetrack Demo.





Now that we understand the overall architecture, let's see how the program actually works. We do not go into the details of the `ControlPanel` and `TrackView` classes because they are quite simple and somewhat irrelevant for our discussions. Instead, we cover the code in the `*Race` classes; that is where the animations occur.

BasicRace

ONLINE
DEMO

`BasicRace` is the simplest version of the demo: It runs the car down the first stretch of the track over some amount of time, showing how to use the basics of `Animator` to perform this task.

There are some constants and instance variables that `BasicRace` uses:

```
public static final int RACE_TIME = 2000;
Point start = TrackView.START_POS;
Point end = TrackView.FIRST_TURN_START;
Point current = new Point();
Animator animator;
```

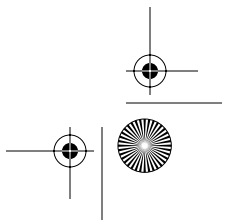
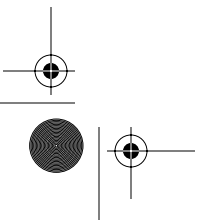
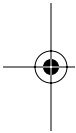
The `start` and `end` constants come from `TrackView`, which keeps track of the eight corners of the race track. The variable `current` is used for storing the position of the car. And the `animator` is, of course, the `Animator` that runs the show.

First of all, `BasicRace` is constructed. It creates a `RaceGUI` object that holds the track and the control panel. It adds itself as a listener to the buttons in the control panel so that it knows when to start and stop the race. And it creates the `Animator` object, which will call our `BasicRace` object with timing events during the animation:

```
public BasicRace(String appName) {
    RaceGUI basicGUI = new RaceGUI(appName);
    basicGUI.getControlPanel().addListener(this);
    track = basicGUI.getTrack();
    animator = new Animator(RACE_TIME, this);
}
```

When one of the buttons is clicked, the `actionPerformed()` method is called, which stops the current animation if `Stop` was clicked and starts a new one if `Go` was clicked:

```
public void actionPerformed(ActionEvent ae) {
    if (ae.getActionCommand().equals("Go")) {
        animator.stop();
        animator.start();
    } else if (ae.getActionCommand().equals("Stop")) {
        animator.stop();
    }
}
```





The heart of the animation is in the implementation of the `TimingTarget` methods. `BasicRace` extends `TimingTargetAdapter`, which implements all `TimingTarget` methods, and `BasicRace` chooses to override only this one method:

```
public void timingEvent(float fraction) {
    // Simple linear interpolation to find current position
    current.x = (int)(start.x + (end.x - start.x) * fraction);
    current.y = (int)(start.y + (end.y - start.y) * fraction);

    // set the new position; this will force a repaint in TrackView
    // and will display the car in the new position
    track.setCarPosition(current);
}
```

The `timingEvent()` method is where the meat of the demo is. This method simply calculates the current position of the car as a linear interpolation from the starting point to the end point, based on the fraction elapsed in the animation. Then it sends this car position to the `TrackView` object, which redraws the race track with the car in the new location.

That's it for the simplest version of the race. It doesn't do much, but you can watch the car run down the first stretch of the track, all powered by just a few lines of code. We will see additional functionality added to the demo as we go through the other sections in this chapter.

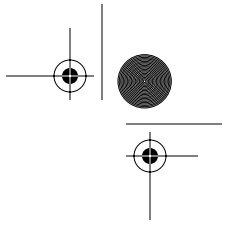
Interpolation

Haven't you sometimes wished you could change time, slow it down, reverse it, accelerate it, or just stop it altogether? That wasn't exactly the motivation for the interpolation features of the framework, but these are some of the things you can do with interpolation, at least in the context of your animations.

We saw in Chapter 13, "Smooth Moves," that nonlinear behavior of animations is a good thing; acceleration, deceleration, and other techniques create a more natural and smoother animation for the viewer.

By default, `Animator` reports linear fraction values in its calls to `timingEvent()`. That is, for any time elapsed t in an animation of length *duration*, the default value for the elapsed fraction will be $t/\textit{duration}$. `TimingTarget` objects are, of course, free to use the fraction value to do whatever they want; thus they can calculate nonlinear movement values given a linear time value. But wouldn't it be nice if the Timing Framework handled the pesky details?





There are actually two mechanisms for controlling the linearity of the timing fraction. The first mechanism, acceleration/deceleration, is quite easy to understand and may be the best to use in many situations. The second mechanism, Interpolator, is more involved, but also much more powerful.

Acceleration and Deceleration

The acceleration and deceleration parameters control whether there are periods of acceleration or deceleration in the animation. They control the value of the fraction passed into `timingEvent`. In a period of acceleration, which is always at the beginning of an animation, the fraction is increasing faster than the fraction based on the real elapsed time during the animation. In a period of deceleration, which is always at the end of an animation, the opposite is true; the fraction is increasing slower than the fraction based on real time elapsed.

Setting these values on an Animator is easy. Simply call the appropriate set methods before the animation begins:

```
setAcceleration(float acceleration)
setDeceleration(float deceleration)
```

Both methods take a value from 0 to 1 that represents the fraction of the animation that should be spent accelerating or decelerating. Note that the two periods are exclusive from each other. An animation cannot be accelerating and decelerating in the same time period of an animation, so $(acceleration + deceleration) \leq 1$ by necessity. These constraints are illustrated in Figure 14-3. Calling either method with values that disobey these constraints results in an `IllegalArgumentException`.

During the period of acceleration, the speed increases at a constant rate of acceleration. The opposite is true for the deceleration period. As the figure shows, all

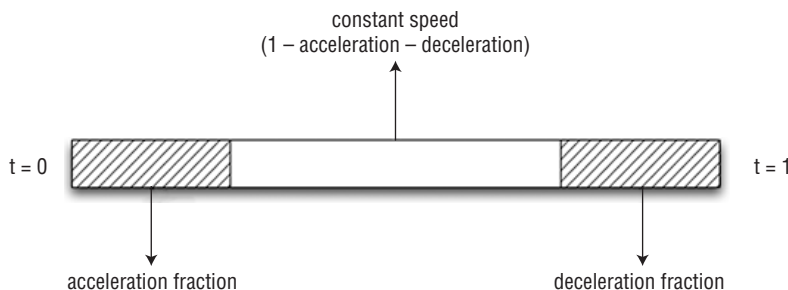
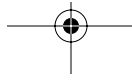
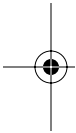


Figure 14-3 Acceleration, deceleration, and default (constant) fractions add up to 1, the total duration of an animation.





animations have an initial fractional period of acceleration, whose default length is 0, followed by some fractional period of constant speed, whose default length is 1, or the entire animation, and they end with a fractional period of deceleration, whose default length is also 0.

Think of the default situation, with no acceleration or deceleration, being an animation that goes from 0 to the full speed of the animation at the start and then from full speed back to 0 at the end. The graph of speed over time is shown in Figure 14-4.

For comparison, imagine an animator `anim` with an acceleration period of `.4` and a deceleration period of `.2`. We would set these parameters with the following statements:

```
anim.setAcceleration(.4f);
anim.setDeceleration(.2f);
```

This animation would ramp up smoothly from 0 to full speed over the first 40 percent of the animation, cruise at constant speed for another 40 percent of the animation, and then ramp down from full speed to 0 over the final 20 percent. The graph of speed over time for this animation would resemble Figure 14-5.

This acceleration/deceleration approach provides an experience of a smoother animation compared to the sudden on/off speed behavior of the default behavior.

If we map the interpolated value over time, we get the graph in Figure 14-6. The straight line is linear interpolation, for comparison.

If you recall from the beginning of this discussion, the value of the `timingEvent()` fraction is being altered by the acceleration and deceleration values. So the

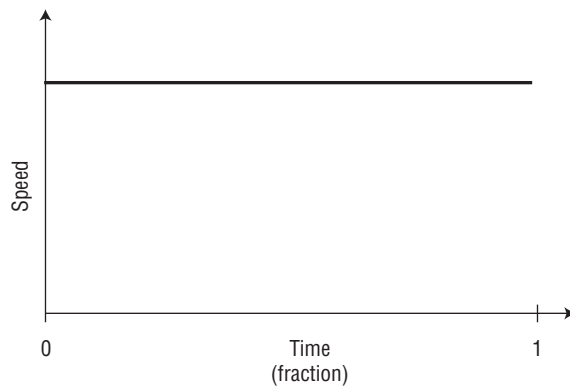


Figure 14-4 Default animation behavior, with no acceleration or deceleration: The speed of animation is constant.



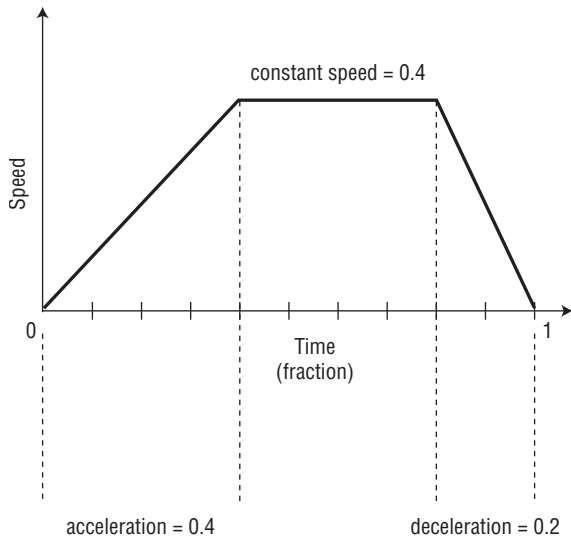


Figure 14-5 Speed of animation with acceleration (.4) and deceleration (.2) factors.



Linear vs. Nonlinear Interpolation

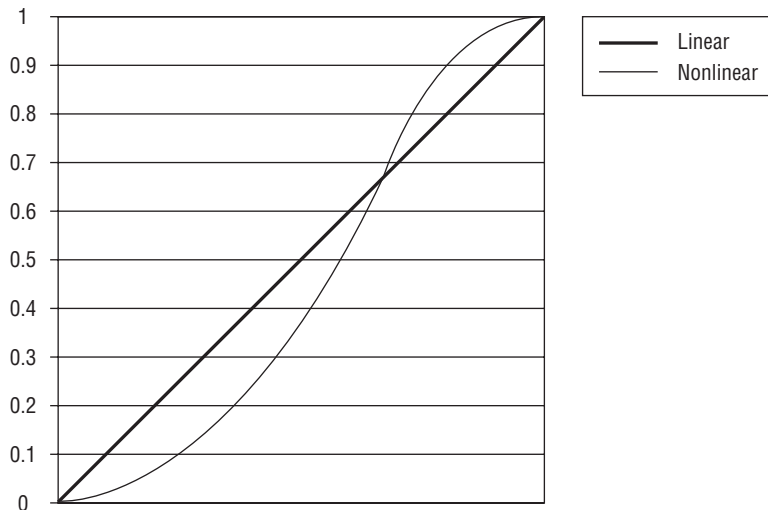
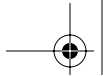


Figure 14-6 Interpolated fraction with acceleration = .4 and deceleration = .2, compared to linear time (straight line).





elapsed fraction values that you receive in your `timingEvent()` method will be nonlinear and will enable you to easily calculate nonlinear values accordingly. For example, if you look at the graph in Figure 14-6, you can imagine what impact this might have on object movement calculations. If the interpolated fraction, represented by the curved line, is being interpolated in this way while linear time, represented by the straight line, marches on, then your calculations on object movement will be affected the same way. If you do a parametric, linear calculation on your object based on the incoming pre-interpolated fraction, then your object will have slowly accelerating movement to begin with, will eventually be going faster than linear movement, and will then slow toward the end as it reaches the final destination. This holds not just for acceleration and deceleration but for the more general methods of interpolation as well.



Tip: Nonlinear motion can be created by performing simple linear interpolation calculations using nonlinear timing values.



Tip: Acceleration/deceleration is an easy way to tap into nonlinear movement for your animations; just tell `Animator` to accelerate and decelerate the timing fraction, and your simple linear calculations take on this advanced, realistic nonlinear motion.

Interpolator Race

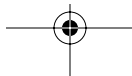
ONLINE
DEMO

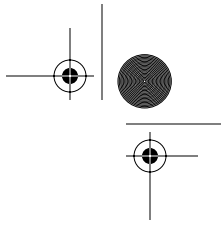
Now let's see how to apply our new knowledge of acceleration and deceleration to our `Racetrack` demo to get a little more realistic motion out of that car.

In the previous version of the demo, `BasicRace`, the car looked pretty good rolling down the track, but it seemed fairly unrealistic.⁷ In particular, it was strange how the car went from 0 to full speed immediately and then simply halted suddenly at the end of the first stretch. Wouldn't it be more realistic to accelerate up to full speed and then have some period of slowing down at the end?

For this demo, we want to reuse as much of `BasicRace` as possible. Therefore, we simply create `NonLinearRace` as a subclass of `BasicRace`. This new class has no functionality in it at all apart from a `main()` method to launch the application

7. Except for the rendering style of the track and car, which I think you'll agree are very realistic.





and a constructor. The constructor defers to the superclass, `BasicRace`, to do its work and then makes two minor adjustments:

```
public NonLinearRace(String appName) {
    super(appName);
    animator.setAcceleration(.5f);
    animator.setDeceleration(.1f);
}
```

These calls into the `animator` object set the acceleration period to half of the animation duration and the deceleration to the last 10 percent. The result is that the car speeds up to full speed over the first half and then slows down to 0 right at the end.

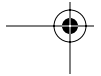
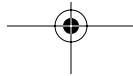
The reason we can make this change to have nonlinear movement so easily is, as we explained earlier, setting the acceleration and deceleration values changes how the fraction is interpolated against real time. During the first half of the animation, the animation fraction is accelerating, or increasing faster than the real elapsed fraction of time in our animation. When we take this fraction and compute the new location of the car, in our existing `BasicRace.timingEvent()` method, our calculation results in the new location also being interpolated at that accelerating rate. Similarly, use of the decelerating fraction toward the end results in decelerating movement in our standard position calculations. So even though we use simple, linear, parametric calculations for the car position in `BasicRace.timingEvent()`, our use of acceleration and deceleration changed the results of those calculations into more realistic nonlinear results by using a nonlinear timing fraction.

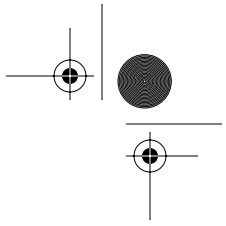
Run the demo. I think you will agree that the car's movement looks a lot better than it did in `BasicRace`.⁸ And with only these two lines of code to make it work, it was well worth the effort.

Interpolator

`Interpolator` is a general and powerful mechanism for interpolating timing values. `Interpolator` is, as we mentioned earlier, somewhat more involved than acceleration and deceleration. Or at least it can be. It is actually quite easy to get complex behavior from `Interpolators` without doing much work. And there is

8. I would put a picture here to show you, but it would look exactly the same as the earlier picture of `BasicRace`. It is so difficult to get across time-based animation nuances when we have only individual pictures to work with. We could spend the rest of the book on a flip-book animation of the result. Maybe if we hadn't gotten around to writing anything more, that's what we would have done here. But now you'll have to satisfy your curiosity by going to the book's Web site and running the demo directly.





Of course, there are other ways to achieve this simple inverse effect, such as setting the direction of REVERSE for an Animator, but this will do for a simple example of custom interpolation.

For a slightly more interesting example, you could provide simple sine-curve behavior with the following class:

```
public class SineInterpolator implements Interpolator {  
    public float interpolate(float fraction) {  
        return (float)Math.sin(fraction * Math.PI);  
    }  
}
```

You can see that it's really up to you and your mathematical imagination here; what kind of effect do you want? Something cyclic? discrete? gravity-based? erratic? random? You can implement your own Interpolator implementation to suit your needs.

Summary

The core functionality covered in this chapter is powerful enough to create great animations. Just the ability to create animations of finite durations with repeating behavior and have your code called with the elapsed animation fraction is a big step up from the built-in timers. With Interpolators thrown in, there are plenty of great animated effects that you can create. But read on; the next chapter details more functionality that makes animations even easier and more powerful.

