JavaFX

Oracle JavaFX Creating Transitions and Timeline Animation in JavaFX

Release 2.1

E20468-04

April 2012

ORACLE®

JavaFX Creating Transitions and Timeline Animation in JavaFX, Release 2.1

April 2012

Copyright © 2011,2012 Oracle and/or its affiliates. All rights reserved.

Primary Author: Dmitry Kostovaorv

Contributing Author: Andrey Nazarov

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Contents

1 Animation Basics

1.1 Transitions 1-1
1.1.1 Fade Transition 1-1
1.1.2 Path Transition 1-1
1.1.3 Parallel Transition 1-2
1.1.4 Sequential Transition 1-3
1.2 Timeline Animation 1-4
1.2.1 Basic Timeline Animation 1-4
1.2.2 Timeline Events 1-5
1.3 Interpolators 1-7
1.3.1 Built-in Interpolators 1-7
1.3.2 Custom Interpolators 1-7

2 Tree Animation Example

Figure 2–1 Project and Elements 2-1
n Grass 2-2
n Creating Grass 2-2
Example 2–1 Creating Timeline Animation for Grass Movement 2-3
Example 2–2 Tree 2-4
Figure 2–4 Branches 2-4
Example 2–4 Leaves and Flowers 2-6
Example 2–5 Animating Tree Elements 2-6
Example 2–6 Growing a Tree 2-6
Example 2–9 Creating Tree Crown Movement 2-7
Example 2–10 Animating Season Change 2-8

Part I

About This Document

This document contains information that you can use to create animation in JavaFX.

Animation Basics provides basic animation concepts and contains the following parts:

- Transitions
- Timeline Animation
- Interpolators

The Tree Animation Example chapter contains a description of the Tree Animation sample and provides some tips and tricks about animation in JavaFX.

1 Animation Basics

Animation in JavaFX can be divided into timeline animation and transitions. This chapter provides examples of each animation type.

- Transitions
- Timeline Animation
- Interpolators

`Timeline` and `Transition` are subclasses of the `javafx.animation.Animation` class. For more information about particular classes, methods, or additional features, see the API documentation.

1.1 Transitions

Transitions in JavaFX provide the means to incorporate animations in an internal timeline. Transitions can be composed to create multiple animations that are executed in parallel or sequentially. See the Parallel Transition and Sequential Transition sections for details. The following sections provide some transition animation examples.

1.1.1 Fade Transition

A fade transition changes the opacity of a node over a given time.

Example 1–1 shows a code snippet for a fade transition that is applied to a rectangle. First a rectangle with rounded corners is created, and then a fade transition is applied to it.

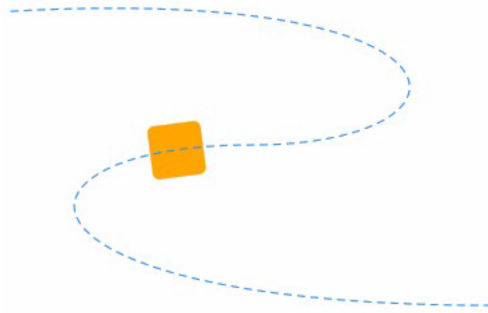
Example 1–1 Fade Transition

```
final Rectangle rect1 = new Rectangle(10, 10, 100, 100);
rect1.setArcHeight(20);
rect1.setArcWidth(20);
rect1.setFill(Color.RED);
...
FadeTransition ft = new FadeTransition(Duration.millis(3000), rect1);
ft.setFromValue(1.0);
ft.setToValue(0.1);
ft.setCycleCount(Timeline.INDEFINITE);
ft.setAutoReverse(true);
ft.play();
```

1.1.2 Path Transition

A path transition moves a node along a path from one end to the other over a given time.

Figure 1–1 Path Transition



Example 1–2 shows a code snippet for a path transition that is applied to a rectangle. The animation is reversed when the rectangle reaches the end of the path. In code, first a rectangle with rounded corners is created, and then a new path animation is created and applied to the rectangle.

Example 1–2 Path Transition

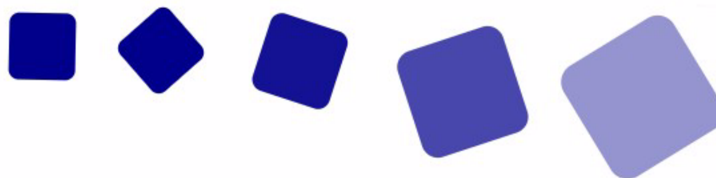
```
final Rectangle rectPath = new Rectangle (0, 0, 40, 40);
rectPath.setArcHeight(10);
rectPath.setArcWidth(10);
rectPath.setFill(Color.ORANGE);
...
Path path = new Path();
path.getElements().add(new MoveTo(20,20));
path.getElements().add(new CubicCurveTo(380, 0, 380, 120, 200, 120));
path.getElements().add(new CubicCurveTo(0, 120, 0, 240, 380, 240));
PathTransition pathTransition = new PathTransition();
pathTransition.setDuration(Duration.millis(4000));
pathTransition.setPath(path);
pathTransition.setNode(rectPath);
pathTransition.setOrientation(PathTransition.OrientationType.ORTHOGONAL_TO_
TANGENT);
pathTransition.setCycleCount(Timeline.INDEFINITE);
pathTransition.setAutoReverse(true);
pathTransition.play();
```

1.1.3 Parallel Transition

A parallel transition executes several transitions simultaneously.

Example 1–3 shows the code snippet for the parallel transition that executes fade, translate, rotate, and scale transitions applied to a rectangle.

Figure 1–2 Parallel Transition



Example 1–3 Parallel Transition

```
Rectangle rectParallel = new Rectangle(10,200,50, 50);
```

```
rectParallel.setArcHeight(15);
rectParallel.setArcWidth(15);
rectParallel.setFill(Color.DARKBLUE);
rectParallel.setTranslateX(50);
rectParallel.setTranslateY(75);
...
        FadeTransition fadeTransition =
            new FadeTransition(Duration.millis(3000), rectParallel);
        fadeTransition.setFromValue(1.0f);
        fadeTransition.setToValue(0.3f);
        fadeTransition.setCycleCount(2);
        fadeTransition.setAutoReverse(true);
        TranslateTransition translateTransition =
            new TranslateTransition(Duration.millis(2000), rectParallel);
        translateTransition.setFromX(50);
        translateTransition.setToX(350);
        translateTransition.setCycleCount(2);
        translateTransition.setAutoReverse(true);
        RotateTransition rotateTransition =
            new RotateTransition(Duration.millis(3000), rectParallel);
        rotateTransition.setByAngle(180f);
        rotateTransition.setCycleCount(4);
        rotateTransition.setAutoReverse(true);
        ScaleTransition scaleTransition =
            new ScaleTransition(Duration.millis(2000), rectParallel);
        scaleTransition.setToX(2f);
        scaleTransition.setToY(2f);
        scaleTransition.setCycleCount(2);
        scaleTransition.setAutoReverse(true);

        parallelTransition = new ParallelTransition();
        parallelTransition.getChildren().addAll(
                fadeTransition,
                translateTransition,
                rotateTransition,
                scaleTransition
        );
        parallelTransition.setCycleCount(Timeline.INDEFINITE);
        parallelTransition.play();
```

1.1.4 Sequential Transition

A sequential transition executes several transitions one after another.

Example 1–4 shows the code for the sequential transition that executes one after another. Fade, translate, rotate, and scale transitions that are applied to a rectangle.

Example 1–4 Sequential Transition

```
Rectangle rectSeq = new Rectangle(25,25,50,50);
rectSeq.setArcHeight(15);
rectSeq.setArcWidth(15);
rectSeq.setFill(Color.CRIMSON);
rectSeq.setTranslateX(50);
rectSeq.setTranslateY(50);

...

         FadeTransition fadeTransition =
            new FadeTransition(Duration.millis(1000), rectSeq);
```

```
        fadeTransition.setFromValue(1.0f);
        fadeTransition.setToValue(0.3f);
        fadeTransition.setCycleCount(1);
        fadeTransition.setAutoReverse(true);

        TranslateTransition translateTransition =
            new TranslateTransition(Duration.millis(2000), rectSeq);
        translateTransition.setFromX(50);
        translateTransition.setToX(375);
        translateTransition.setCycleCount(1);
        translateTransition.setAutoReverse(true);

        RotateTransition rotateTransition =
            new RotateTransition(Duration.millis(2000), rectSeq);
        rotateTransition.setByAngle(180f);
        rotateTransition.setCycleCount(4);
        rotateTransition.setAutoReverse(true);

        ScaleTransition scaleTransition =
            new ScaleTransition(Duration.millis(2000), rectSeq);
        scaleTransition.setFromX(1);
        scaleTransition.setFromY(1);
        scaleTransition.setToX(2);
        scaleTransition.setToY(2);
        scaleTransition.setCycleCount(1);
        scaleTransition.setAutoReverse(true);

sequentialTransition = new SequentialTransition();
sequentialTransition.getChildren().addAll(
        fadeTransition,
        translateTransition,
        rotateTransition,
        scaleTransition);
sequentialTransition.setCycleCount(Timeline.INDEFINITE);
sequentialTransition.setAutoReverse(true);

sequentialTransition.play();
```

For more information about animation and transitions, see the API documentation and the Animation section in the Ensemble project in the SDK.

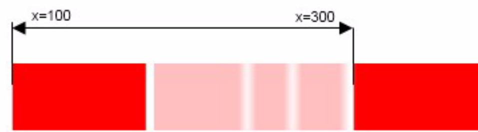
1.2 Timeline Animation

An animation is driven by its associated properties, such as size, location, and color etc. `Timeline` provides the capability to update the property values along the progression of time. JavaFX supports key frame animation. In key frame animation, the animated state transitions of the graphical scene are declared by start and end snapshots (key frames) of the state of the scene at certain times. The system can automatically perform the animation. It can stop, pause, resume, reverse, or repeat movement when requested.

1.2.1 Basic Timeline Animation

The code in Example 1–5 animates a rectangle horizontally and moves it from its original position `X=100` to `X=300` in 500 ms. To animate an object horizontally, alter the x-coordinates and leave the y-coordinates unchanged.

Figure 1–3 Horizontal Movement



Example 1–5 shows the code snippet for the basic timeline animation.

Example 1–5 Timeline Animation

```
final Rectangle rectBasicTimeline = new Rectangle(100, 50, 100, 50);
rectBasicTimeline.setFill(Color.RED);
...
final Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
final KeyValue kv = new KeyValue(rectBasicTimeline.xProperty(), 300);
final KeyFrame kf = new KeyFrame(Duration.millis(500), kv);
timeline.getKeyFrames().add(kf);
timeline.play();
```

1.2.2 Timeline Events

JavaFX provides the means to incorporate events that can be triggered during the timeline play. The code in Example 1–6 changes the radius of the circle in the specified range, and `KeyFrame` triggers the random transition of the circle in the x-coordinate of the scene.

Example 1–6 Timeline Events

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.animation.AnimationTimer;
import javafx.animation.KeyFrame;
import javafx.animation.KeyValue;
import javafx.animation.Timeline;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.effect.Lighting;
import javafx.scene.layout.StackPane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.scene.text.Text;
import javafx.util.Duration;

public class TimelineEvents extends Application {

    //main timeline
    private Timeline timeline;
    private AnimationTimer timer;

    //variable for storing actual frame
    private Integer i=0;

    @Override public void start(Stage stage) {
```

```
        Group p = new Group();
        Scene scene = new Scene(p);
        stage.setScene(scene);
        stage.setWidth(500);
        stage.setHeight(500);
        p.setTranslateX(80);
        p.setTranslateY(80);

        //create a circle with effect
        final Circle circle = new Circle(20,  Color.rgb(156,216,255));
        circle.setEffect(new Lighting());
        //create a text inside a circle
        final Text text = new Text (i.toString());
        text.setStroke(Color.BLACK);
        //create a layout for circle with text inside
        final StackPane stack = new StackPane();
        stack.getChildren().addAll(circle, text);
        stack.setLayoutX(30);
        stack.setLayoutY(30);

        p.getChildren().add(stack);
        stage.show();

        //create a timeline for moving the circle
        timeline = new Timeline();
        timeline.setCycleCount(Timeline.INDEFINITE);
        timeline.setAutoReverse(true);

//You can add a specific action when each frame is started.
        timer = new AnimationTimer() {
            @Override
            public void handle(long l) {
                text.setText(i.toString());
                i++;
            }

        };

        //create a keyValue with factory: scaling the circle 2times
        KeyValue keyValueX = new KeyValue(stack.scaleXProperty(), 2);
        KeyValue keyValueY = new KeyValue(stack.scaleYProperty(), 2);

        //create a keyFrame, the keyValue is reached at time 2s
        Duration duration = Duration.millis(2000);
        //one can add a specific action when the keyframe is reached
        EventHandler onFinished = new EventHandler<ActionEvent>() {
            public void handle(ActionEvent t) {
                 stack.setTranslateX(java.lang.Math.random()*200-100);
                 //reset counter
                 i = 0;
            }
        };

  KeyFrame keyFrame = new KeyFrame(duration, onFinished , keyValueX, keyValueY);

        //add the keyframe to the timeline
        timeline.getKeyFrames().add(keyFrame);

        timeline.play();
        timer.start();
```

```
    }

    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

1.3 Interpolators

Interpolation defines positions of the object between the start and end points of the movement. You can use various built-in implementations of the Interpolator class or you can implement your own Interpolator to achieve custom interpolation behavior.

1.3.1 Built-in Interpolators

JavaFX provides several built-in interpolators that you can use to create different effects in your animation. By default, JavaFX uses linear interpolation to calculate the coordinates.

Example 1–7 shows a code snippet where the EASE_BOTH interpolator instance is added to the KeyValue in the basic timeline animation. This interpolator creates a spring effect when the object reaches its start point and its end point.

Example 1–7 Built-in Interpolator

```
final Rectangle rectBasicTimeline = new Rectangle(100, 50, 100, 50);
rectBasicTimeline.setFill(Color.BROWN);
...
final Timeline timeline = new Timeline();
timeline.setCycleCount(Timeline.INDEFINITE);
timeline.setAutoReverse(true);
final KeyValue kv = new KeyValue(rectBasicTimeline.xProperty(), 300,
 Interpolator.EASE_BOTH);
final KeyFrame kf = new KeyFrame(Duration.millis(500), kv);
timeline.getKeyFrames().add(kf);
timeline.play();
```

1.3.2 Custom Interpolators

Apart from built-in interpolators, you can implement your own interpolator to achieve custom interpolation behavior. A custom interpolator example consists of two java files. Example 1–8 shows a custom interpolator that is used to calculate the y-coordinate for the animation. Example 1–9 shows the code snippet of the animation where the AnimationBooleanInterpolator is used.

Example 1–8 Custom Interpolator

```
public class AnimationBooleanInterpolator extends Interpolator {
    @Override
    protected double curve(double t) {
        return Math.abs(0.5-t)*2 ;
    }
}
```

Example 1–9 Animation with Custom Interpolator

```
 final KeyValue keyValue1 = new KeyValue(rect.xProperty(), 300);
 AnimationBooleanInterpolator yInterp = new AnimationBooleanInterpolator();
```

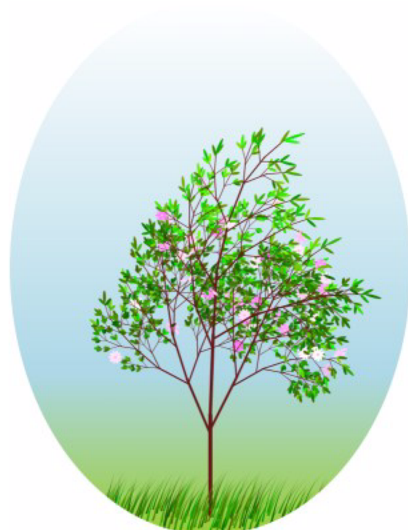
```
final KeyValue keyValue2 = new KeyValue(rect.yProperty(), 0., yInterp);
```

2

Tree Animation Example

This chapter provides details about the Tree Animation example. You will learn how all the elements on the scene were created and animated. Figure 2–1 shows the scene with a tree.

Figure 2–1 Tree Animation



Project and Elements

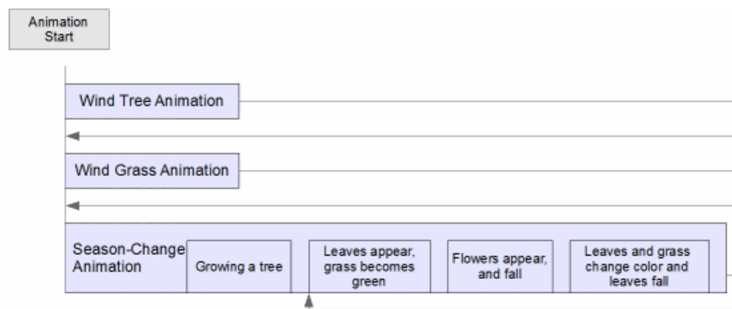
The Tree Animation project consists of several files. Each element, like leaves, grass blades, and others are created in separate classes. TreeGenerator class creates a tree from all the elements. Animator class contains all animation except grass animation that resides in the GrassWindAnimation class.

The scene in the example contains the following elements:

- Tree with branches, leaves, and flowers
- Grass

Each element is animated in its own fashion. Some animations run in parallel, and others run sequentially. The tree-growing animation is run only once, whereas the season-change animation is set to run infinitely.

Figure 2–2 Animation Timeline



The season-change animation includes the following parts:

- Leaves and flowers appear on the tree
- Flower petals fall and disappear
- Leaves and grass change color
- Leaves fall to the ground and disappear

Grass

This section describes how the grass is created and animated.

Creating Grass

In the Tree Animation example, the grass, shown in Figure 2–3 consists of separate grass blades, each of which is created using `Path` and added to the list. Each blade is then curved and colored. An algorithm is used to randomize the height, curve, and color of the blades, and to distribute the blades on the "ground." You can specify the number of blades and the size of the "ground" covered with grass.

Figure 2–3 Grass

Example 2–1 Creating a Grass Blade

```
public class Blade extends Path {

    public final Color SPRING_COLOR = Color.color(random() * 0.5, random() * 0.5
 + 0.5, 0.).darker();
    public final Color AUTUMN_COLOR = Color.color(random() * 0.4 + 0.3, random()
 * 0.1 + 0.4, random() * 0.2);
    private final static double width = 3;
    private double x = RandomUtil.getRandom(170);
    private double y = RandomUtil.getRandom(20) + 20;
    private double h = (50 * 1.5 - y / 2) * RandomUtil.getRandom(0.3);
    public SimpleDoubleProperty phase = new SimpleDoubleProperty();

    public Blade() {
```

```
        getElements().add(new MoveTo(0, 0));
        final QuadCurveTo curve1;
        final QuadCurveTo curve2;
        getElements().add(curve1 = new QuadCurveTo(-10, h, h / 4, h));
        getElements().add(curve2 = new QuadCurveTo(-10, h, width, 0));

        setFill(AUTUMN_COLOR); //autumn color of blade
        setStroke(null);

        getTransforms().addAll(Transform.translate(x, y));

        curve1.yProperty().bind(new DoubleBinding() {

            {
                super.bind(curve1.xProperty());
            }

            @Override
            protected double computeValue() {

                final double xx0 = curve1.xProperty().get();
                return Math.sqrt(h * h - xx0 * xx0);
            }
        }); //path of top of blade is circle

        //code to bend blade
        curve1.controlYProperty().bind(curve1.yProperty().add(-h / 4));
        curve2.controlYProperty().bind(curve1.yProperty().add(-h / 4));

        curve1.xProperty().bind(new DoubleBinding() {

            final double rand = RandomUtil.getRandom(PI / 4);

            {
                super.bind(phase);
            }

            @Override
            protected double computeValue() {
                return (h / 4) + ((cos(phase.get() + (x + 400.) * PI / 1600 +
 rand) + 1) / 2.) * (-3. / 4) * h;
            }
        });
    }
}
```

Creating Timeline Animation for Grass Movement

Timeline animation that changes the x-coordinate of the top of the blade is used to create grass movement.

Several algorithms are used to make the movement look natural. For example, the top of each blade is moved in a circle instead of a straight line, and side curve of the blade make the blade look as if it bends under the wind. Random numbers are added to separate each blade movement.

Example 2–2 Grass Animation

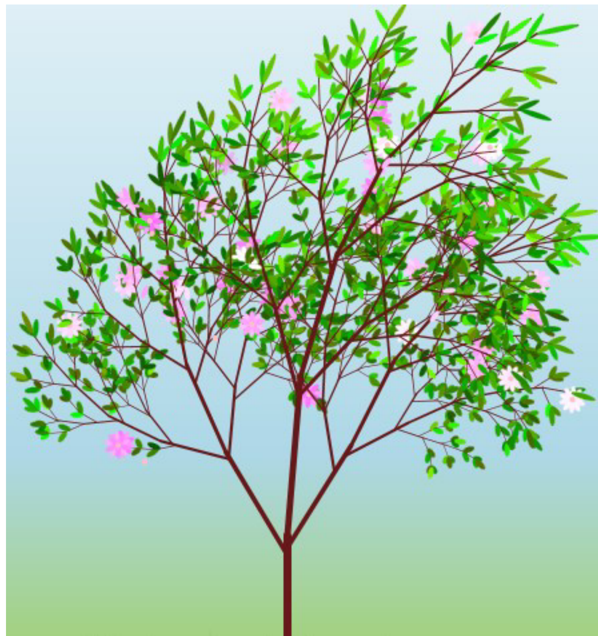
```
class GrassWindAnimation extends Transition {
```

```
    final private Duration animationTime = Duration.seconds(3);
    final private DoubleProperty phase = new SimpleDoubleProperty(0);
    final private Timeline tl = new Timeline(Animation.INDEFINITE);

    public GrassWindAnimation(List<Blade> blades) {

        setCycleCount(Animation.INDEFINITE);
        setInterpolator(Interpolator.LINEAR);
        setCycleDuration(animationTime);
        for (Blade blade : blades) {
            blade.phase.bind(phase);
        }
    }

    @Override
    protected void interpolate(double frac) {
        phase.set(frac * 2 * PI);
    }
}
```

Tree

This section explains how the tree shown in Figure 2–4 is created and animated.

Figure 2–4 Tree



Branches

The tree consists of branches, leaves, and flowers. Leaves and flowers are drawn on the top branches of the tree. Each branch generation consists of three branches (one top and two side branches) drawn from a parent branch. You can specify the number of generations in the code using the NUMBER_OF_BRANCH_GENERATIONS passed in the constructor of TreeGenerator in the Main class. Example 2–3 shows the code in the TreeGenerator class that creates the trunk of the tree (or the root branch) and adds three branches for the following generations.

Example 2–3 Root Branch

```
    private List<Branch> generateBranches(Branch parentBranch, int depth) {
        List<Branch> branches = new ArrayList<>();
        if (parentBranch == null) { // add root branch
            branches.add(new Branch());
        } else {
            if (parentBranch.length < 10) {
                return Collections.emptyList();
            }
            branches.add(new Branch(parentBranch, Type.LEFT, depth));
            branches.add(new Branch(parentBranch, Type.RIGHT, depth));
            branches.add(new Branch(parentBranch, Type.TOP, depth));
        }

        return branches;
    }
```

To make the tree look more natural, each child generation branch is grown at an angle to the parent branch, and each child branch is smaller than its parent. The child angle is calculated using random values. Example 2–4 provides a code for creating child branches.

Example 2–4 Child Branches

```
    public Branch(Branch parentBranch, Type type, int depth) {
        this();
        SimpleDoubleProperty locAngle = new SimpleDoubleProperty(0);
        globalAngle.bind(locAngle.add(parentBranch.globalAngle.get()));
        double transY = 0;
        switch (type) {
            case TOP:
                transY = parentBranch.length;
                length = parentBranch.length * 0.8;
                locAngle.set(getRandom(10));
                break;
            case LEFT:
            case RIGHT:
                transY = parentBranch.length - getGaussianRandom(0,
parentBranch.length, parentBranch.length / 10, parentBranch.length / 10);
                locAngle.set(getGaussianRandom(35, 10) * (Type.LEFT == type ? 1 :
-1));
                if ((0 > globalAngle.get() || globalAngle.get() > 180) && depth <
4) {
                    length = parentBranch.length * getGaussianRandom(0.3, 0.1);
                } else {
                    length = parentBranch.length * 0.6;
                }
                break;
        }
        setTranslateY(transY);
        getTransforms().add(new Rotate(locAngle.get(), 0, 0));
        globalH = getTranslateY() * cos(PI / 2 - parentBranch.globalAngle.get() *
PI / 180) + parentBranch.globalH;
        setBranchStyle(depth);
        addChildToParent(parentBranch, this);
    }
```

Leaves and Flowers

Leaves are created on top branches. Because the leaves are created at the same time as the branches of the tree, leaves are scaled to 0 by `leaf.setScaleX(0)` and `leaf.setScaleY(0)` to hide them before the tree is grown as shown in the Example 2–5. The same trick is used to hide the leaves when they fall. To create a more natural look, leaves have slightly different shades of green. Also, the leaf color changes depending on the location of the leaf; the darker shades are applied to the leaves located below the middle of the tree crown.

Example 2–5 Leaf Shape and Placement

```
public class Leaf extends Ellipse {

    public final Color AUTUMN_COLOR;
    private final int N = 5;
    private List<Ellipse> petals = new ArrayList<>(2 * N + 1);

    public Leaf(Branch parentBranch) {
        super(0, parentBranch.length / 2., 2, parentBranch.length / 2.);
        setScaleX(0);
        setScaleY(0);

        double rand = random() * 0.5 + 0.3;
        AUTUMN_COLOR = Color.color(random() * 0.1 + 0.8, rand, rand / 2);

        Color color = new Color(random() * 0.5, random() * 0.5 + 0.5, 0, 1);
        if (parentBranch.globalH < 400 && random() < 0.8) { //bottom leaf is
darker
            color = color.darker();
        }
        setFill(color);
    }
}
```

Flowers are created in the Flower class and then added to the top branches of the tree in the TreeGenerator class. You can specify the number of petals in a flower. Petals are ellipses distributed in a circle with some overlapping. Similar to grass and leaves, the flower petals are colored in different shades of pink.

Animating Tree Elements

This section explains techniques employed in the Tree Animation example to animate the tree and season change. Parallel transition is used to start all the animations in the scene as shown in Example 2–6.

Example 2–6 Main Animation

```
final Transition all = new ParallelTransition(new GrassWindAnimation(grass),
treeWindAnimation, new SequentialTransition(branchGrowingAnimation,
seasonsAnimation(tree, grass)));
        all.play();
```

Growing a Tree

Tree growing animation is run only once, at the beginning of the Tree Animation example. The application starts a sequential transition animation to grow branches one generation after another as shown in Example 2–7. Initially length is set to 0. The root branch size and angle are specified in the `TreeGenerator` class. Currently each generation is grown during two seconds.

Example 2–7 Sequential Transition to Start Branch Growing Animation

```
SequentialTransition branchGrowingAnimation = new SequentialTransition();
```

The code in Example 2–8 creates the Tree growing animation:

Example 2–8 Branch Growing Animation

```
    private Animation animateBranchGrowing(List<Branch> branchGeneration, int
 depth, Duration duration) {

       ParallelTransition sameDepthBranchAnimation = new ParallelTransition();
       for (final Branch branch : branchGeneration) {
         Timeline branchGrowingAnimation = new Timeline(new KeyFrame(duration,
 new KeyValue(branch.base.endYProperty(), branch.length)));
            sameDepthBranchAnimation.getChildren().add(
                    new SequentialTransition(
PauseTransitionBuilder.create().duration(Duration.ONE).onFinished(new
 EventHandler<ActionEvent>() {

                @Override
                public void handle(ActionEvent t) {
                    branch.base.setStrokeWidth(branch.length / 25);
                }
            }).build(),
                    branchGrowingAnimation));

        }
        return sameDepthBranchAnimation;

    }
```

Because all the branch lines are calculated and created simultaneously, they could appear on the scene as dots. The code introduces a few tricks to hide the lines before they grow. In Example the code `duration.one millisecond` pauses transition for an unnoticeable time. In the Example 2–9, the `base.setStrokeWidth(0)` code sets branches width to 0 before the grow animation starts for each generation.

Example 2–9 Tree Growing Animation Optimization

```
    private void setBranchStyle(int depth) {
        base.setStroke(Color.color(0.4, 0.1, 0.1, 1));

        if (depth < 5) {
            base.setStrokeLineJoin(StrokeLineJoin.ROUND);
            base.setStrokeLineCap(StrokeLineCap.ROUND);
        }
        base.setStrokeWidth(0);
    }
}
```

Creating Tree Crown Movement

In parallel with growing a tree, wind animation starts. Tree branches, leaves, and flowers are moving together.

Tree wind animation is similar to grass movement animation, but it is simpler because only the angle of the branches changes. To make the tree movement look natural, the bend angle is different for different branch generations. The higher the generation of the branch (that is the smaller the branch), the more it bends. Example 2–10 provides code for wind animation.

Example 2–10 Wind Animation

```
    private Animation animateTreeWind(List<Branch> branchGeneration, int depth,
 Duration duration) {
     ParallelTransition wind = new ParallelTransition();
     for (final Branch brunch : branchGeneration) {
        final Rotate rotation = new Rotate(0);
        brunch.getTransforms().add(rotation);
wind.getChildren().add(TimelineBuilder.create().keyFrames(new KeyFrame(duration,
new KeyValue(rotation.angleProperty(), depth *
 2))).autoReverse(true).cycleCount(Animation.INDEFINITE).build());
        }
        return wind;
    }
```

Animating Season Change

Season-change animation actually starts after the tree has grown, and runû infinitely. The code in Example 2–11 calls all the season animations:

Example 2–11 Starting Season Animation

```
private Transition seasonsAnimation(final Tree tree, final List<Blade> grass) {

        Transition spring = animateSpring(tree.leafage, grass);
        Transition flowers = animateFlowers(tree.flowers);
        Transition autumn = animateAutumn(tree.leafage, grass);
        return SequentialTransitionBuilder.create().children(spring, flowers,
 autumn).cycleCount(Animation.INDEFINITE).build();
    }

    private Transition animateSpring(List<Leaf> leafage, List<Blade> grass) {
        ParallelTransition springAnimation = new ParallelTransition();
        for (final Blade blade : grass) {
springAnimation.getChildren().add(FillTransitionBuilder.create().shape(blade).
toValue(blade.SPRING_COLOR).duration(GRASS_BECOME_GREEN_DURATION).build());
        }
        for (Leaf leaf : leafage) {
springAnimation.getChildren().add(ScaleTransitionBuilder.create().toX(1).
toY(1).node(leaf).duration(LEAF_APPEARING_DURATION).build());
        }
        return springAnimation;
    }
```

Once all the tree branches are grown, leaves start to appear as directed in Example 2–12.

Example 2–12 Parallel Transition to Start Spring Animation and Show Leaves

```
    private Transition animateSpring(List<Leaf> leafage, List<Blade> grass) {
        ParallelTransition springAnimation = new ParallelTransition();
        for (final Blade blade : grass) {
springAnimation.getChildren().add(FillTransitionBuilder.create().shape(blade).
toValue(blade.SPRING_COLOR).duration(GRASS_BECOME_GREEN_DURATION).build());
        }
        for (Leaf leaf : leafage) {
springAnimation.getChildren().add(ScaleTransitionBuilder.create().toX(1).toY(1).
node(leaf).duration(LEAF_APPEARING_DURATION).build());
        }
        return springAnimation;
    }
```

When all leaves are visible, flowers start to appear as shown in Example 2–13. The sequential transition is used to show flowers gradually. The delay in flower appearance is set in the sequential transition code of Example 2–13. Flowers appear only in the tree crown.

Example 2–13 Showing Flowers

```
    private Transition animateFlowers(List<Flower> flowers) {

        ParallelTransition flowersAppearAndFallDown = new ParallelTransition();

        for (int i = 0; i < flowers.size(); i++) {
            final Flower flower = flowers.get(i);
            for (Ellipse pental : flower.getPetals()) {
          flowersAppearAndFallDown.getChildren().add(new SequentialTransition(
                        FadeTransitionBuilder.create().delay(FLOWER_APPEARING_
DURATION.divide(3).multiply(i + 1)).duration(FLOWER_APPEARING_
DURATION).node(pental).toValue(1).build(),
                        fakeFallDownAnimation(pental)));
            }
        }
        return flowersAppearAndFallDown;
    }
```

Once all the flowers appear on the screen, their petals start to fall. In the code in Example 2–14 the flowers are duplicated and the first set of them is hidden to show it later.

Example 2–14 Duplicating Petals

```
    private Ellipse copyEllipse(Ellipse petalOld, Color color) {
        Ellipse ellipse = new Ellipse();
        ellipse.setRadiusX(petalOld.getRadiusX());
        ellipse.setRadiusY(petalOld.getRadiusY());
        if (color == null) {
            ellipse.setFill(petalOld.getFill());
        } else {
            ellipse.setFill(color);
        }
        ellipse.setRotate(petalOld.getRotate());
        ellipse.setOpacity(0);
        return ellipse;
    }
```

Copied flower petals start to fall to the ground one by one as shown in Example 2–15. The petals disappear after five seconds on the ground. The fall trajectory of a petal is not a straight line, but rather a calculated sine curve, so that petals seem to be whirling as they fall.

Example 2–15 Shedding Flowers

```
           Animation fakeLeafageDown = fakeFallDownEllipseAnimation((Ellipse)
 leaf, leaf.AUTUMN_COLOR, new HideMethod() {

                @Override
                public void hide(Node node) {
                    node.setScaleX(0);
                    node.setScaleY(0);
                }
            });
```

The next season change starts when all the flowers disappear from the scene. The leaves and grass become yellow, and the leaves fall and disappear. The same algorithm used in Example 2–15 to make the flower petals fall is used to show falling leaves. The code in Example 2–16 enables autumn animation.

Example 2–16 Animating Autumn Changes

```
    private Transition animateAutumn(List<Leaf> leafage, List<Blade> grass) {
        ParallelTransition autumn = new ParallelTransition();

        ParallelTransition yellowLeafage = new ParallelTransition();
        ParallelTransition dissappearLeafage = new ParallelTransition();

        for (final Leaf leaf : leafage) {

            final FillTransition toYellow =
 FillTransitionBuilder.create().shape(leaf).toValue(leaf.AUTUMN
_COLOR).duration(LEAF_BECOME_YELLOW_DURATION).build();
            yellowLeafage.getChildren().add(toYellow);

            Animation fakeLeafageDown = fakeFallDownEllipseAnimation((Ellipse)
 leaf, leaf.AUTUMN_COLOR, new HideMethod() {

                @Override
                public void hide(Node node) {
                    node.setScaleX(0);
                    node.setScaleY(0);
                }
            });
            dissappearLeafage.getChildren().add(new SequentialTransition(
                    fakeLeafageDown,
                    FillTransitionBuilder.create().shape(leaf).toValue((Color)
 leaf.getFill()).duration(Duration.ONE).build()));
        }

    ParallelTransition grassBecomeYellowAnimation = new ParallelTransition();
        for (final Blade blade : grass) {
            final FillTransition toYellow =
 FillTransitionBuilder.create().shape(blade).toValue(blade.AUTUMN_
COLOR).delay(Duration.seconds(1 * random())).duration(GRASS_BECOME_YELLOW_
DURATION).build();
            grassBecomeYellowAnimation.getChildren().add(toYellow);
        }

        autumn.getChildren().addAll(grassBecomeYellowAnimation, new
 SequentialTransition(yellowLeafage, dissappearLeafage));
        return autumn;
    }
```

After all leaves disappear from the ground, spring animation starts by coloring grass in green and showing leaves.