Оглавление

Введение

Развитие Всемирной паутины привело в 2004 году к рождению архитектуры Web 2.0 — набору рекомендаций и решений для создания и поддержки Web-ресурсов, при реализации которых распределенные системы Web 2.0 становятся тем более насыщенными контентом, чем больше пользователей ими пользуются, причем качество такой распределенной системы зависит от уровня взаимодействия и активности пользователей, в отличие от интернет-систем архитектуры Web 1.0, где за качество поставляемого контента целиком и полностью отвечает владелец Web-ресурса.

Web-приложения архитектуры Web 2.0 ориентированы на совместное использование информации, а также взаимодействие и сотрудничество участников Всемирной паутины. Примером систем Web 2.0 могут служить социальные сети, блоги, файловые и видеохостинги и т. д.

При миграции клиент-серверных систем от Web 1.0 к Web 2.0 клиентское приложение становится не просто "тонким" клиентом, отображающим формируемое сервером статическое содержимое в виде HTML-страниц, а насыщенным интернет-приложением RIA (Rich Internet Application) — Web-приложением, предоставляющим большую интерактивность для клиента.

Большая, по сравнению с традиционным "тонким" клиентом, интерактивность RIA-приложения обеспечивается за счет богатого графического GUI-интерфейса пользователя, содержащего, помимо разнообразных компонентов контроля, анимацию, векторную графику и аудио/видеоролики, при этом приемлемая скорость работы такого приложения достигается с помощью переноса части выполняемого кода с сервера на сторону клиента. Таким образом, RIA-приложение является промежуточным между "тонким" клиентом и "толстым" клиентом — настольным приложением.

Традиционный "тонкий" клиент использует для взаимодействия с сервером HTML-разметку и простой JavaScript-код и не требует загрузки и инсталляции дополнительного программного обеспечения. В отличие от "тонкого" клиента RIA-приложение создается на базе определенной платформы, предоставляющей язык программирования и набор библиотек программного интерфейса, и поэтому его работа на стороне клиента требует наличия среды выполнения соответствующей

платформы, которую необходимо предварительно загрузить и инсталлировать. Такой средой выполнения, как правило, служит плагин Web-браузера.

Примечание

RIA-приложения можно также создавать на основе языка разметки HTML5 и JavaScript-библиотек, таких как jQuery, Yahoo!, MochaUI и др.

Преимуществом RIA-приложения перед настольным приложением является отсутствие необходимости его специальной установки на клиентском компьютере — RIA-приложение предоставляется определенной областью на страничке Web-браузера, при этом RIA-приложение имеет автоматическое обновление версий и кроссплатформенность, т. к. его оберткой служит Web-браузер. Кроме того, работа RIA-приложения автоматически защищена "песочницей" Web-браузера.

Наиболее популярными технологиями для создания RIA-приложений на сегодняшний день являются платформы Adobe Flash, JavaFX и Microsoft Silverlight.

Все вышеупомянутые RIA-платформы позволяют разрабатывать не только Web-приложения, работающие в Web-браузере, но и настольные приложения, которые способны работать в режиме оффлайн. Для платформы Adobe Flash — это AIR-приложения или SWF-файлы, запускаемые автономно плеером Flash Player, для платформы Microsoft Silverlight — это Web-приложения, помещенные в специальный каталог со ссылками с рабочего стола или из меню **Пуск**.

Что касается технологии JavaFX, то один и тот же Java-код, созданный на базе платформы JavaFX, может запускаться как настольное приложение, которое разворачивается на клиентском компьютере автономно, или разворачиваться как Java Web Start-приложение, или отображаться в Web-браузере как JavaFX-апплет, встроенный в HTML-страничку.

Платформа JavaFX может использоваться совместно с технологией Swing, а также с другими языками — JRuby, Groovy и JavaScript для создания больших и комплексных приложений с насыщенным графическим интерфейсом пользователя.

Технология JavaFX обеспечивает создание мощного графического интерфейса пользователя (Graphical User Interface, GUI) для крупномасштабных приложений, ориентированных на обработку данных, насыщенных медиаприложений, поставляющих разнообразный медиаконтент пользователю, Mashup-приложений, объединяющих различные Web-ресурсы для пользователя, компонентов высококачественной графики и анимации для Web-сайтов, различного рода пользовательских программ с графикой, анимацией и интерактивными элементами.

Технологии создания RIA-приложений платформы Java берут свое начало от Java-апплетов, GUI-интерфейсы которых использовали графические системы AWT и Swing для организации взаимодействия с пользователем и отображения ему данных, текста, графики и анимации.

Графическая библиотека AWT была самой первой графической Java-системой набора JDK 1.0, дополненной затем библиотекой Java 2D двумерной графики и изображений. Библиотека AWT предоставляет разработчику возможность использования таких основных компонентов GUI-интерфейса, как кнопки, переключатели,

списки, метки, окна выбора файла, меню, компоненты визуализации и редактирования текста, функции drag and drop, возможность обработки событий UI-компонентов, компоновки компонентов в рабочей области, работы с цветом, шрифтом, графикой, рисования и печати. Библиотека AWT является тяжеловесной, т. к. она содержит собственную (родную, native) библиотеку java.awt.peer, через которую взаимодействует с операционной системой компьютера, поэтому отображение AWT GUI-интерфейса зависит от операционной системы, в которой приложение развернуто.

Ограниченность набора GUI-компонентов библиотеки AWT и ее тяжеловесность послужили причиной создания графической системы Swing, которая основывается на библиотеке AWT и поэтому является уже легковесной. Кроме того, библиотека Swing дополняет библиотеку AWT такими компонентами GUI-интерфейса, как панель выбора цвета, индикатор состояния, переключатель, слайдер и спиннер, панель с вкладками, таблицы и деревья, расширенными возможностями компоновки GUI-компонентов, таймером, возможностью изменения внешнего вида LookAndFeel GUI-интерфейса, отображения HTML-контента. Библиотека Swing реализует архитектуру MVC (Model-View-Controller) и потоковую модель Event Dispatch Thread (EDT).

Несмотря на богатые возможности графических систем AWT и Swing, они не удовлетворяют современным требованиям работы с медиаконтентом, что и послужило причиной создания платформы JavaFX, которая предоставляет современные GUI-компоненты, богатый набор библиотек графического и медиапрограммного API-интерфейса, а также высокопроизводительную среду выполнения приложений.

Первоначально, в 2007—2010 годах, версии 1.1, 1.2 и 1.3 платформы JavaFX содержали:

- декларативный язык программирования JavaFX Script создания UI-интерфейса;
- набор JavaFX SDK, обеспечивающий компилятор и среду выполнения;
- плагины для сред выполнения NetBeans IDE и Eclipse;
- плагины для Adobe Photoshop и Adobe Illustrator, позволяющие экспортировать графику в код JavaFX Script, инструменты конвертации графического формата SVG в код JavaFX Script.

Платформа JavaFX версии 2.0 выпуска 2011 года кардинально отличается от платформы JavaFX версии 1.*x*.

Платформа JavaFX 2.0 больше не поддерживает язык JavaFX Script, а вместо этого предлагает новый программный интерфейс JavaFX API для создания JavaFX-приложений полностью на языке Java. Для альтернативного декларативного описания графического интерфейса пользователя платформа JavaFX 2.0 предлагает новый язык FXML. Кроме того, платформа JavaFX 2.0 обеспечивает новые графический и медийный движки, улучшающие воспроизведение графического и мультимедийного контента, встраивание HTML-контента в приложение, новый плагин для Web-браузеров, широкий выбор UI-компонентов с поддержкой CSS3. При этом платформа JavaFX версии 2.0 содержит:

- набор JavaFX SDK, предоставляющий инструмент JavaFX Packager tool компиляции, упаковки и развертывания JavaFX-приложений, Ant-библиотеку для сборки JavaFX-приложений, библиотеки JavaFX API и документацию;
- среду выполнения JavaFX Runtime для работы настольных JavaFX-приложений и JavaFX-апплетов;
- поддержку платформы JavaFX 2.0 для среды выполнения NetBeans IDE 7;
- примеры JavaFX-приложений.

В дальнейшем платформа JavaFX будет интегрирована в платформу JDK 8 и не потребует отдельной инсталляции.

Сайт новой платформы JavaFX 2.0 находится по адресу: **http://javafx.com/**.

В данной книге рассматривается новая технология JavaFX 2.0 создания RIA-приложений платформы Java с насыщенным графическим GUI-интерфейсом и мощными возможностями по воспроизведению графического и мультимедийного контента, обработки данных и совмещению с другими Java-технологиями.

ГЛАВА 5

Трансформация и анимация

Платформа JavaFX 2.0 обеспечивает создание двух видов анимации — анимацию по ключевым кадрам и анимацию со встроенной временной шкалой.

JavaFX-анимацию представляет пакет javafx.animation, базовым классом которого является класс `Animation`. Класс `Animation` расширяется классами `Timeline` и `Transition`, при этом класс `Timeline` представляет *анимацию по ключевым кадрам*, а класс `Transition` — *анимацию со встроенной временной шкалой*.

Класс `Animation` имеет набор свойств, позволяющих управлять скоростью и направлением анимации, задержкой и количеством циклов анимации, устанавливать автореверс анимации, считывать статус анимации, обрабатывать завершение анимации и др.

Скорость и направление анимации можно установить с помощью метода `setRate(double value)`, задержку анимации — с помощью метода `setDelay(Duration value)`, количество циклов анимации — методом `setCycleCount(int value)`, автореверс анимации — методом `setAutoReverse(boolean value)`, считать статус `Animation.Status.PAUSED`, `Animation.Status.RUNNING` или `Animation.Status.STOPPED` анимации — методом `getStatus()`, установить обработчик завершения анимации — методом `setOnFinished(EventHandler<ActionEvent> value)`.

Также класс `Animation` предоставляет методы управления жизненным циклом анимации:

- `jumpTo(Duration time)` — переход анимации к указанной позиции на временной шкале;
- `playFrom(Duration time)` — запуск анимации, начиная с указанной позиции на временной шкале;
- `play()` — запуск анимации с текущей позиции на временной шкале;
- `playFromStart()` — запуск анимации с первоначальной позиции на временной шкале;
- `stop()` — остановка анимации;
- `pause()` — пауза анимации.

Анимация по ключевым кадрам позволяет создать видимое изменение значения любого JavaFX-свойства за определенный промежуток времени с помощью класса `Timeline`.

Экземпляр класса `Timeline` можно создать с помощью класса-фабрики `TimelineBuilder` или посредством одного из конструкторов, позволяющих установить частоту кадров и набор ключевых кадров анимации:

```
public Timeline(double targetFramerate)
public Timeline(double targetFramerate, KeyFrame... keyFrames)
public Timeline(KeyFrame... keyFrames) и public Timeline()
```

Набор ключевых кадров `Timeline`-анимации можно пополнить методом `getKeyFrames().addAll()`, а остановить `Timeline`-анимацию и вернуть ее в первоначальную позицию — методом `stop()`.

Ключевой кадр `Timeline`-анимации представлен классом `javafx.animation.KeyFrame` и определяет изменения значений JavaFX-свойств за определенный промежуток времени.

Экземпляр класса `KeyFrame` можно создать с помощью набора конструкторов, позволяющих установить время воспроизведения ключевого кадра, имя ключевого кадра, обработчик окончания ключевого кадра и набор изменений значений JavaFX-свойств:

```
public KeyFrame(Duration time, java.lang.String name,
                EventHandler<ActionEvent> onFinished,
                java.util.Collection<KeyValue<?>> values)
public KeyFrame(Duration time, java.lang.String name,
    EventHandler<ActionEvent> onFinished, KeyValue<?>... values)
public KeyFrame(Duration time, EventHandler<ActionEvent> onFinished,
    KeyValue<?>... values)
public KeyFrame(Duration time, java.lang.String name,
    KeyValue<?>... values)
public KeyFrame(Duration time, KeyValue<?>... values)
```

Изменение значения JavaFX-свойства представлено классом `javafx.animation.KeyValue`, экземпляр которого можно создать с помощью конструкторов, позволяющих установить изменяемое JavaFX-свойство, его конечное значение в результате анимации и способ его изменения в течение анимации:

```
public KeyValue(WritableValue<T> target, T endValue,
                Interpolator<? super T> interpolator)
public KeyValue(WritableValue<T> target, T endValue)
```

Способ изменения значения JavaFX-свойства в течение анимации представлен классом `javafx.animation.Interpolator`, имеющим статические поля:

- `Interpolator.DISCRETE` — дискретное изменение значения JavaFX-свойства, при котором значение остается начальным до окончания временного интервала, когда значение становится конечным;

- `Interpolator.LINEAR` (по умолчанию) — линейное изменение значения JavaFX-свойства, при котором значение определяется по формуле
  `startValue + (endValue − startValue) × fraction`;
- `Interpolator.EASE_BOTH` — используется величина `0.2` для прироста и уменьшения значения JavaFX-свойства;
- `Interpolator.EASE_IN` — используется величина `0.2` для прироста значения JavaFX-свойства;
- `Interpolator.EASE_OUT` — используется величина `0.2` для уменьшения значения JavaFX-свойства.

Кроме того, можно создать пользовательский класс `Interpolator` с переопределением его методов, описывающих изменение значения JavaFX-свойства.

`Transition`-анимация со встроенной временной шкалой также использует объект `Interpolator` в качестве значения свойства `interpolator` класса `Transition`.

Таким образом, анимацию изменения значений нескольких JavaFX-свойств можно создать двумя способами. Первый способ — это создание одного ключевого кадра `KeyFrame` и добавление в него нескольких объектов `KeyValue`. Другой способ — это создание отдельных ключевых кадров `KeyFrame` для каждого из объектов `KeyValue` и добавление их в `Timeline`-анимацию.

`Transition`-анимация со встроенной временной шкалой, в отличие от `Timeline`-анимации, описывает изменение во времени ограниченного набора JavaFX-свойств, таких как прозрачность, пространственное положение, вращение и масштабирование узла графа сцены, а также цвет заполнения и цвет контура формы `Shape`.

Анимация прозрачности узла графа сцены создается с помощью класса `FadeTransition`, имеющего свойства:

- `byValue` — шаг изменения свойства прозрачности;
- `duration` — продолжительность анимации;
- `fromValue` — начальное значение прозрачности;
- `node` — целевой узел графа сцены данной анимации;
- `toValue` — конечное значение прозрачности.

Экземпляр класса `FadeTransition` создается с помощью класса-фабрики `FadeTransitionBuilder` или посредством конструкторов

```
public FadeTransition(Duration duration, Node node)
public FadeTransition(Duration duration)
public FadeTransition()
```

Анимация пространственного положения узла графа сцены создается с помощью классов `PathTransition` и `TranslateTransition`.

Класс `PathTransition` позволяет создавать перемещение графического объекта вдоль кривой с помощью свойств:

- `duration` — продолжительность анимации;
- `orientation` — ориентация, если `PathTransition.OrientationType.NONE` — ориентация графического объекта не изменяется, если `PathTransition.OrientationType.ORTHOGONAL_TO_TANGENT` — графический объект перпендикулярен относительно кривой своего перемещения;
- `path` — объект `javafx.scene.shape.Shape`, представляющий кривую перемещения.

Экземпляр класса `PathTransition` создается с помощью класса-фабрики `PathTransitionBuilder` или посредством конструкторов:

```
public PathTransition()
public PathTransition(Duration duration, Shape path, Node node)
public PathTransition(Duration duration, Shape path)
```

Класс `TranslateTransition` позволяет создавать перемещение графического объекта из одной 3D-точки в другую 3D-точку с помощью свойств:

- `node` — целевой узел для анимации;
- `duration` — продолжительность анимации;
- `fromX` — начальная координата перемещения по оси *x*;
- `fromY` — начальная координата перемещения по оси *y*;
- `fromZ` — начальная координата перемещения по оси *z*;
- `toX` — конечная координата перемещения по оси *x*;
- `toY` — конечная координата перемещения по оси *y*;
- `toZ` — конечная координата перемещения по оси *z*;
- `byX` — шаг перемещения по оси *x*;
- `byY` — шаг перемещения по оси *y*;
- `byZ` — шаг перемещения по оси *z*.

Экземпляр класса `TranslateTransition` можно создать с помощью класса-фабрики `TranslateTransitionBuilder` или посредством конструкторов:

```
public TranslateTransition(Duration duration, Node node)
public TranslateTransition(Duration duration)
public TranslateTransition()
```

Анимация вращения узла графа сцены создается с помощью класса `RotateTransition`, имеющим свойства:

- `node` — целевой узел анимации;
- `duration` — продолжительность анимации;
- `axis` — ось вращения `javafx.geometry.Point3D`;
- `fromAngle` — начальный угол вращения;
- `toAngle` — конечный угол вращения;
- `byAngle` — шаг вращения.

Экземпляр класса `RotateTransition` создается с помощью класса-фабрики `RotateTransitionBuilder` или посредством конструкторов:

```
public RotateTransition(Duration duration, Node node)
public RotateTransition(Duration duration)
public RotateTransition()
```

Анимация масштабирования узла графа сцены создается с помощью класса `ScaleTransition`, имеющего свойства:

- `node` — целевой узел анимации;
- `duration` — продолжительность анимации;
- `fromX` — начальное значение масштабирования по оси *x*;
- `fromY` — начальное значение масштабирования по оси *y*;
- `fromZ` — начальное значение масштабирования по оси *z*;
- `toX` — конечное значение масштабирования по оси *x*;
- `toY` — конечное значение масштабирования по оси *y*;
- `toZ` — конечное значение масштабирования по оси *z*;
- `byX` — шаг масштабирования по оси *x*;
- `byY` — шаг масштабирования по оси *y*;
- `byZ` — шаг масштабирования по оси *z*.

Экземпляр класса `ScaleTransition` можно создать с помощью класса-фабрики `ScaleTransitionBuilder` или посредством конструкторов:

```
public ScaleTransition(Duration duration, Node node)
public ScaleTransition(Duration duration)
public ScaleTransition()
```

Анимация цвета заполнения формы `Shape` создается с помощью класса `FillTransition`, имеющего свойства:

- `duration` — продолжительность анимации;
- `fromValue` — начальное значение цвета;
- `shape` — целевой объект `javafx.scene.shape.Shape` анимации;
- `toValue` — конечное значение цвета.

Экземпляр класса `FillTransition` создается с помощью класса-фабрики `FillTransitionBuilder` или посредством конструкторов:

```
public FillTransition(Duration duration, Shape shape, Color fromValue,
                      Color toValue)
public FillTransition(Duration duration, Color fromValue, Color toValue)
public FillTransition(Duration duration, Shape shape)
public FillTransition(Duration duration)
public FillTransition()
```

Анимация цвета контура формы `Shape` создается с помощью класса `StrokeTransition`, имеющего свойства:

- `shape` — целевой объект `javafx.scene.shape.Shape` для анимации;
- `duration` — продолжительность анимации;
- `fromValue` — начальный цвет контура;
- `toValue` — конечный цвет контура.

Экземпляр класса `StrokeTransition` создается с помощью класса-фабрики `StrokeTransitionBuilder` или посредством конструкторов:

```
public StrokeTransition(Duration duration, Shape shape, Color fromValue,
                        Color toValue)
public StrokeTransition(Duration duration, Color fromValue, Color toValue)
public StrokeTransition(Duration duration, Shape shape)
public StrokeTransition(Duration duration)
public StrokeTransition()
```

Классы `ParallelTransition` и `SequentialTransition` дают возможность группировать анимации в параллельное и последовательное выполнение.

Класс `ParallelTransition` имеет свойство `node` (целевой узел графа сцены для анимации) и конструкторы:

```
public ParallelTransition()
public ParallelTransition(Node node, Animation... children)
public ParallelTransition(Animation... children)
public ParallelTransition(Node node)
```

а также класс-фабрику `ParallelTransitionBuilder`. Метод `getChildren().addAll` позволяет пополнить список дочерних параллельных анимаций.

Класс `SequentialTransition` имеет свойство `node` (целевой узел графа сцены для анимации) и конструкторы:

```
public SequentialTransition()
public SequentialTransition(Node node, Animation... children)
public SequentialTransition(Animation... children)
public SequentialTransition(Node node)
```

а также класс-фабрику `SequentialTransitionBuilder`. Метод `getChildren().addAll` позволяет пополнить список дочерних последовательных анимаций.

Класс `PauseTransition` позволяет сделать паузу в последовательности анимаций. Класс `PauseTransition` имеет свойство `duration` (продолжительность паузы) и конструкторы `public PauseTransition(Duration duration)` и `public PauseTransition()`, а также класс-фабрику `PauseTransitionBuilder`.

Класс `AnimationTimer` позволяет создавать таймер, вызываемый в каждом кадре анимации. Создать таймер можно расширив абстрактный класс `AnimationTimer` с переопределением его метода `handle(long now)`, вызываемом в каждом кадре. Для управления таймером класс `AnimationTimer` предлагает методы `start()` и `stop()`.

Код далее демонстрирует `Transition`- и `Timeline`-анимацию букв текста. `Transition`-анимация состоит из параллельных анимаций перемещения, вращения и масштабирования, а `Timeline`-анимация изменяет расположение источника света для эффекта подсветки текста:

```
Group root = new Group();
Scene scene = new Scene(root, 500, 500, Color.BLACK);
//Создание эффекта подсветки текста
final Light.Point lightPoint = new Light.Point();
  lightPoint.setColor(Color.WHITE);
  lightPoint.setX(0.0);
  lightPoint.setY(0.0);
  lightPoint.setZ(100.0);
final Lighting effect = new Lighting();
  effect.setLight(lightPoint);
  effect.setDiffuseConstant(1.5);
  effect.setSpecularConstant(1.5);
  effect.setSurfaceScale(8);
//Создание букв текста
final Text tJ = new Text();
  tJ.setEffect(effect);
  tJ.setX(80);
  tJ.setY(250);
  tJ.setText("J");
  tJ.setFill(Color.RED);
  tJ.setFont(Font.font(null, FontWeight.BOLD, 80));
final Text ta1 = new Text();
  ta1.setEffect(effect);
  ta1.setX(120);
  ta1.setY(250);
  ta1.setText("a");
  ta1.setFill(Color.RED);
  ta1.setFont(Font.font(null, FontWeight.BOLD, 80));
final Text tv = new Text();
  tv.setEffect(effect);
  tv.setX(170);
  tv.setY(250);
  tv.setText("v");
  tv.setFill(Color.RED);
  tv.setFont(Font.font(null, FontWeight.BOLD, 80));
final Text ta2 = new Text();
  ta2.setEffect(effect);
  ta2.setX(220);
  ta2.setY(250);
  ta2.setText("a");
  ta2.setFill(Color.RED);
  ta2.setFont(Font.font(null, FontWeight.BOLD, 80));
```

```
final Text tF = new Text();
  tF.setEffect(effect);
  tF.setX(270);
  tF.setY(250);
  tF.setText("F");
  tF.setFill(Color.RED);
  tF.setFont(Font.font(null, FontWeight.BOLD, 80));
final Text tX = new Text();
  tX.setEffect(effect);
  tX.setX(320);
  tX.setY(250);
  tX.setText("X");
  tX.setFill(Color.RED);
  tX.setFont(Font.font(null, FontWeight.BOLD, 80));
//Создание анимаций перемещения, вращения и масштабирования букв текста
final TranslateTransition ttJ =
      new TranslateTransition(Duration.millis(1000),tJ);
  ttJ.setCycleCount(2);
  ttJ.setByX(-200.0);
  ttJ.setToY(-270.0);
  ttJ.setAutoReverse(true);
final RotateTransition rtJ =
      new RotateTransition(Duration.millis(500),tJ);
  rtJ.setByAngle(360);
  rtJ.setAxis(new Point3D(10.0, 10.0, 10.0));
  rtJ.setCycleCount(4);
final ScaleTransition stJ = new ScaleTransition(Duration.millis(1000),tJ);
  stJ.setByX(-1.5);
  stJ.setByY(-1.5);
  stJ.setCycleCount(2);
  stJ.setAutoReverse(true);
final TranslateTransition tta1 =
      new TranslateTransition(Duration.millis(1000),ta1);
  tta1.setCycleCount(2);
  tta1.setByX(-100.0);
  tta1.setToY(-270.0);
  tta1.setAutoReverse(true);
final RotateTransition rta1 =
      new RotateTransition(Duration.millis(500),ta1);
  rta1.setByAngle(360);
  rta1.setAxis(new Point3D(10.0, 10.0, 10.0));
  rta1.setCycleCount(4);
final ScaleTransition sta1 =
      new ScaleTransition(Duration.millis(1000),ta1);
  sta1.setByX(-1.5);
  sta1.setByY(-1.5);
  sta1.setCycleCount(2);
  sta1.setAutoReverse(true);
```

```
final TranslateTransition ttv =
      new TranslateTransition(Duration.millis(1000),tv);
  ttv.setCycleCount(2);
  ttv.setByX(0.0);
  ttv.setToY(-270.0);
  ttv.setAutoReverse(true);
final RotateTransition rtv =
      new RotateTransition(Duration.millis(500),tv);
  rtv.setByAngle(360);
  rtv.setAxis(new Point3D(10.0, 10.0, 10.0));
  rtv.setCycleCount(4);
final ScaleTransition stv = new ScaleTransition(Duration.millis(1000),tv);
  stv.setByX(-1.5);
  stv.setByY(-1.5);
  stv.setCycleCount(2);
  stv.setAutoReverse(true);
final TranslateTransition tta2 =
      new TranslateTransition(Duration.millis(1000),ta2);
  tta2.setCycleCount(2);
  tta2.setByX(100.0);
  tta2.setToY(-270.0);
  tta2.setAutoReverse(true);
final RotateTransition rta2 =
      new RotateTransition(Duration.millis(500),ta2);
  rta2.setByAngle(360);
  rta2.setAxis(new Point3D(10.0, 10.0, 10.0));
  rta2.setCycleCount(4);
final ScaleTransition sta2 =
      new ScaleTransition(Duration.millis(1000),ta2);
  sta2.setByX(-1.5);
  sta2.setByY(-1.5);
  sta2.setCycleCount(2);
  sta2.setAutoReverse(true);
final TranslateTransition ttF =
      new TranslateTransition(Duration.millis(1000),tF);
  ttF.setCycleCount(2);
  ttF.setByX(200.0);
  ttF.setToY(-270.0);
  ttF.setAutoReverse(true);
final RotateTransition rtF =
      new RotateTransition(Duration.millis(500),tF);
  rtF.setByAngle(360);
  rtF.setAxis(new Point3D(10.0, 10.0, 10.0));
  rtF.setCycleCount(4);
final ScaleTransition stF = new ScaleTransition(Duration.millis(1000),tF);
  stF.setByX(-1.5);
  stF.setByY(-1.5);
```

```
  stF.setCycleCount(2);
  stF.setAutoReverse(true);
final TranslateTransition ttX =
      new TranslateTransition(Duration.millis(1000),tX);
  ttX.setCycleCount(2);
  ttX.setByX(300.0);
  ttX.setToY(-270.0);
  ttX.setAutoReverse(true);
final RotateTransition rtX =
      new RotateTransition(Duration.millis(500),tX);
  rtX.setByAngle(360);
  rtX.setAxis(new Point3D(10.0, 10.0, 10.0));
  rtX.setCycleCount(4);
final ScaleTransition stX = new ScaleTransition(Duration.millis(1000),tX);
  stX.setByX(-1.5);
  stX.setByY(-1.5);
  stX.setCycleCount(2);
  stX.setAutoReverse(true);
//Группировка анимаций в параллельное выполнение
final ParallelTransition ptJ = new ParallelTransition(tJ,ttJ,rtJ,stJ);
final ParallelTransition pta1 =
      new ParallelTransition(ta1,tta1,rta1,sta1);
final ParallelTransition ptv = new ParallelTransition(tv,ttv,rtv,stv);
final ParallelTransition pta2 =
      new ParallelTransition(ta2,tta2,rta2,sta2);
final ParallelTransition ptF = new ParallelTransition(tF,ttF,rtF,stF);
final ParallelTransition ptX = new ParallelTransition(tX,ttX,rtX,stX);
//Создание анимации перемещения источника света эффекта подсветки текста
final Timeline timeline = new Timeline();
timeline.setCycleCount(2);
timeline.setAutoReverse(true);
KeyValue kv =
  new KeyValue(lightPoint.xProperty(),500.0, Interpolator.EASE_BOTH);
KeyFrame kf = new KeyFrame(Duration.millis(1000), kv);
timeline.getKeyFrames().addAll(kf);
timeline.setDelay(Duration.millis(2000));
//Создание кнопки запуска анимации
Button btn = new Button();
  btn.setLayoutX(10);
  btn.setLayoutY(450);
  btn.setText("Анимация");
  btn.setStyle("-fx-font: bold italic 14pt Georgia;-fx-text-fill: white;-fx-
background-color: black;-fx-border-width: 1px; -fx-border-color:white");
  btn.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
      if (!timeline.getStatus().equals(Animation.Status.RUNNING)){
        ptJ.play();
```

```
        pta1.play();
        ptv.play();
        pta2.play();
        ptF.play();
        ptX.play();
        timeline.play();  }
  }});
root.getChildren().addAll(btn, tJ,ta1,tv,ta2,tF,tX);
```

Примечание

Проект JavaFXApplicationAnimation с примером создания `Transition`- и `Timeline`-анимации находится в папке Примеры\Глава5 в электронном архиве *(см. приложение в конце книги)*.

Пакет javafx.scene.transform платформы JavaFX 2.0 обеспечивает трансформации узлов графа сцены, состоящие из аффинных преобразований: вращений, перемещений, масштабирования и сдвига.

В отличие от анимации, трансформации графических объектов не имеют плавного видимого перехода от начальной точки к конечной точке в течение определенного промежутка времени, а выполняются сразу.

Базовым классом JavaFX-трансформаций является класс `javafx.scene.transform.Transform`, имеющий реализации в виде классов `Affine`, `Rotate`, `Scale`, `Shear` и `Translate`.

Применить JavaFX-трансформации к узлу графа сцены можно двумя способами. Первый способ — это использовать метод `getTransforms()` класса `javafx.scene.Node`, возвращающий список `ObservableList<Transform>` объектов `Transform`, пополнить который можно методом `addAll()`. Другой способ — это применение методов `setRotate()`, `setRotationAxis()`, `setScaleX()`, `setScaleY()`, `setScaleZ()`, `setTranslateX()`, `setTranslateY()` и `setTranslateZ()` класса `javafx.scene.Node`, обеспечивающих трансформации вращения, масштабирования и перемещения для узла графа сцены.

Класс `Affine` представляет аффинные преобразования матрицы:

$$\begin{bmatrix} mxx & mxy & mxz & tx \\ myx & myy & myz & ty \\ mzx & mzy & mzz & tz \end{bmatrix}$$

с помощью свойств:

- `mxx` — *X*-множитель матрицы;
- `mxy` — *XY*-множитель матрицы;
- `mxz` — *XZ*-множитель матрицы;
- `tx` — сдвиг по оси *x*;
- `myx` — *YX*-множитель матрицы;
- `myy` — *Y*-множитель матрицы;

- `myz` — *YZ*-множитель матрицы;
- `ty` — сдвиг по оси *y*;
- `mzx` — *ZX*-множитель матрицы;
- `mzy` — *ZY*-множитель матрицы;
- `mzz` — *Z*-множитель матрицы;
- `tz` — сдвиг по оси *z*.

Экземпляр класса `Affine` можно создать с помощью класса-фабрики `AffineBuilder`, с помощью конструктора `public Affine()` или с помощью статического метода `affine()` класса `Transform`, возвращающего `Affine`-объект.

Аффинные преобразования отображают *n*-мерный объект в *n*-мерный, сохраняют параллельность линий и плоскостей, а также пропорции параллельных объектов. С помощью аффинных преобразований можно создавать трансформации вращения, перемещения, масштабирования и сдвига.

Класс `Rotate` обеспечивает вращение узла графа сцены с помощью свойств:

- `angle` — угол вращения;
- `pivotX` — горизонтальная координата опорной точки вращения;
- `pivotY` — вертикальная координата опорной точки вращения;
- `pivotZ` — *Z*-координата опорной точки вращения.

Экземпляр класса `Rotate` можно создать с помощью класса-фабрики `RotateBuilder`, посредством набора конструкторов:

```
public Rotate(), public Rotate(double angle)
public Rotate(double angle, Point3D axis)
public Rotate(double angle, double pivotX, double pivotY)
public Rotate(double angle, double pivotX, double pivotY, double pivotZ)
public Rotate(double angle, double pivotX, double pivotY,
              double pivotZ, Point3D axis)
```

или с помощью статического метода `rotate()` класса `Transform`, возвращающего `Rotate`-объект.

Класс `Scale` обеспечивает масштабирование узла графа сцены с помощью свойств:

- `x` — множитель масштабирования по оси *x*;
- `y` — множитель масштабирования по оси *y*;
- `z` — множитель масштабирования по оси *z*;
- `pivotX` — горизонтальная координата опорной точки масштабирования;
- `pivotY` — вертикальная координата опорной точки масштабирования;
- `pivotZ` — *Z*-координата опорной точки масштабирования.

Экземпляр класса `Scale` можно создать с помощью класса-фабрики `ScaleBuilder`, посредством набора конструкторов:

```
public Scale()
public Scale(double x, double y)
public Scale(double x, double y, double pivotX, double pivotY)
public Scale(double x, double y, double z)
public Scale(double x, double y, double z, double pivotX,
             double pivotY, double pivotZ)
```

или с помощью статического метода `scale()` класса `Transform`, возвращающего `Scale`-объект.

Класс `Shear` обеспечивает сдвиг узла графа сцены с помощью свойств:

- `x` — множитель по оси *x* от `-1` до `1`;
- `y` — множитель по оси *y* от `-1` до `1`;
- `pivotX` — горизонтальная координата опорной точки сдвига;
- `pivotY` — вертикальная координата опорной точки сдвига.

Экземпляр класса `Shear` можно создать с помощью класса-фабрики `ShearBuilder`, посредством набора конструкторов:

```
public Shear()
public Shear(double x, double y)
public Shear(double x, double y, double pivotX, double pivotY)
```

или с помощью статического метода `shear()` класса `Transform`, возвращающего `Shear`-объект.

Класс `Translate` обеспечивает перемещение узла графа сцены с помощью свойств:

- `x` — смещение по оси *x*;
- `y` — смещение по оси *y*;
- `z` — смещение по оси *z*.

Экземпляр класса `Translate` можно создать с помощью класса-фабрики `TranslateBuilder`, посредством набора конструкторов

```
public Translate()
public Translate(double x, double y)
public Translate(double x, double y, double z)
```

или с помощью статического метода `translate()` класса `Transform`, возвращающего `Translate`-объект.