JavaFX Studies/ Aula09: Animations

Primeiro vamos começar com alguns tipos de animação básicos:

Antes disso vamos declarar algumas coisas: para esses métodos de animação estaremos utilizando um Node que recebe essa animação, então ele realiza a animação, e junto a isso o nosso disparador da animação está sendo o método Initialize, que dispara a animação quando a tela é iniciada.

O primeiro será o TranslateTransition:(Ela simula o movimento do seu Node)

```
@Override
public void initialize(URL location,
ResourceBundle resources) {
    //Translate:
    TranslateTransition translate = new
    TranslateTransition();
    translate.setNode(image);
    translate.setDuration(Duration.millis
    (ms:1000));
    translate.setCycleCount
    (TranslateTransition.INDEFINITE);
    translate.setByX(value:200);
    translate.setByY(-150);
    translate.setDelay(Duration.millis
    (ms:1000));
    translate.setAutoReverse(value:true);
    translate.play();
}
```

Vamos explicar o código linha por linha:

1. `TranslateTransition translate = new TranslateTransition();` - Isso cria uma nova instância da classe `TranslateTransition`, que é usada para realizar animações de translação (movimento) em JavaFX.

2. `translate.setNode(image);` - Esta linha define o nó (ou objeto) que será animado. No caso específico, o código está animando uma imagem representada pela variável `image`.

3. `translate.setDuration(Duration.millis(1000));` - Aqui, é definida a duração da animação. A duração é configurada em milissegundos e, neste caso, é definida como 1000 milissegundos, ou seja, 1 segundo.

4. `translate.setCycleCount(TranslateTransition.INDEFINITE);` - Define o número de vezes que a animação será repetida. Neste caso, está configurado como `TranslateTransition.INDEFINITE`, o que significa que a animação será repetida infinitamente.

5. `translate.setByX(200);` - Define a quantidade pela qual o nó será movido na direção horizontal (coordenada X). Aqui, o nó será movido 200 unidades para a direita.

6. `translate.setByY(-150);` - Define a quantidade pela qual o nó será movido na direção vertical (coordenada Y). Neste caso, o nó será movido 150 unidades para cima (a negativa indica uma direção para cima).

7. `translate.setDelay(Duration.millis(1000));` - Esta linha configura um atraso antes que a animação comece. É configurado com um atraso de 1000 milissegundos (ou 1 segundo).

8. `translate.setAutoReverse(true);` - Define se a animação deve ser reproduzida no sentido oposto após a conclusão de uma iteração. Neste caso, está configurado como verdadeiro, o que significa que a animação será reproduzida no sentido oposto.

9. `translate.play();` - Inicia a animação. Uma vez que todas as configurações foram feitas, este método inicia a animação conforme especificado pelas configurações anteriores.

Essas linhas juntas criam uma animação que move a imagem horizontal e verticalmente, repetidamente, com um atraso inicial e com a capacidade de ser invertida ao final de cada ciclo.

Agora vamos explicar o RotateTransition:

```
RotateTransition rotate = new
RotateTransition();
rotate.setNode(image);
rotate.setDuration(Duration.millis
(ms:500));
rotate.setCycleCount
(RotateTransition.INDEFINITE);
rotate.setByAngle(value:360);
rotate.setInterpolator(Interpolator.
LINEAR);
rotate.setDelay(Duration.millis
(ms:1000));
rotate.setAutoReverse(value:true);
rotate.setAxis(Rotate.Y_AXIS);//
Podendo ser X/Y/Z
rotate.play();
```

- setNode(image): Este método define o nó (Node) que será rotacionado pela transição. Você precisa passar o objeto que deseja rotacionar como argumento para este método. No exemplo dado, parece que image é o nó que será rotacionado.

- setDuration(Duration.millis(500)): Este método define a duração da transição de rotação. Você precisa passar um objeto Duration como argumento, especificando a quantidade de tempo que deseja que a rotação leve. Neste caso, Duration.millis(500) especifica que a rotação deve ocorrer ao longo de 500 milissegundos (ou 0.5 segundos).

- setCycleCount(RotateTransition.INDEFINITE): Este método define quantas vezes a transição de rotação será repetida. Você pode passar um valor numérico (inteiro) ou RotateTransition.INDEFINITE se quiser que a rotação seja repetida indefinidamente. No exemplo, está configurado para ser executado indefinidamente.

- setByAngle(360): Este método define o ângulo pelo qual o nó será rotacionado. Neste caso, o valor passado é 360, o que significa uma rotação completa (360 graus) em torno do eixo especificado.

- setInterpolator(Interpolator.LINEAR): Este método define o interpolador usado para calcular os valores intermédios durante a transição. O interpolador afeta a velocidade da animação ao longo do tempo. Interpolator.LINEAR indica um movimento linear, ou seja, a rotação ocorre a uma velocidade constante.

- setDelay(Duration.millis(1000)): Este método define um atraso antes de iniciar a transição de rotação. Você passa um objeto Duration especificando a quantidade de tempo de atraso antes de começar a rotação. Neste caso, Duration.millis(1000) significa um atraso de 1000 milissegundos (ou 1 segundo).

- setAutoReverse(true): Este método define se a transição deve ser revertida automaticamente após a conclusão. Se configurado como true, após a rotação completa, o objeto irá inverter automaticamente a animação e retornar à sua posição inicial.

- setAxis(Rotate.Y_AXIS): Este método define o eixo em torno do qual o nó será rotacionado. Você pode passar um dos valores Rotate.X_AXIS, Rotate.Y_AXIS ou Rotate.Z_AXIS para especificar o eixo de rotação. Neste caso, parece que o eixo de rotação é o eixo Y.

- play(): Este método inicia a animação da transição de rotação. Depois de ter configurado todos os parâmetros, você chama este método para começar a animação de rotação.

Agora vamos entender sobre FadeTransition:

```
// Fade:
FadeTransition fade = new
FadeTransition();
fade.setNode(image);
fade.setDuration(Duration.millis
(ms:1000));
fade.setCycleCount(FadeTransition.
INDEFINITE);
fade.setInterpolator(Interpolator.
LINEAR);
fade.setFromValue(value:0);//Para
fade in use 0, fade out use 1
fade.setToValue(value:1);//Para fade
in use 1, fade out use 0
fade.play();
```

Aqui está a explicação detalhada dos métodos usados para configurar a transição de fade:

1. **`setNode(image)`**: Este método define o nó (Node) ao qual a transição de fade será aplicada. Você precisa passar o objeto que deseja aplicar o efeito de fade como argumento para este método. No exemplo dado, parece que `image` é o nó ao qual o efeito de fade será aplicado.

2. **`setDuration(Duration.millis(1000))`**: Este método define a duração da transição de fade. Você precisa passar um objeto `Duration` como argumento, especificando a quantidade de tempo que deseja que o fade leve. Neste caso, `Duration.millis(1000)` especifica que o fade deve ocorrer ao longo de 1000 milissegundos (ou 1 segundo).

3. **`setCycleCount(FadeTransition.INDEFINITE)`**: Este método define quantas vezes a transição de fade será repetida. Você pode passar um valor numérico (inteiro) ou `FadeTransition.INDEFINITE` se quiser que o fade seja repetido indefinidamente. No exemplo, está configurado para ser executado indefinidamente.

4. **`setInterpolator(Interpolator.LINEAR)`**: Este método define o interpolador usado para calcular os valores intermédios durante a transição. O interpolador afeta a velocidade da animação ao longo do tempo. `Interpolator.LINEAR` indica um movimento linear, ou seja, o fade ocorre a uma velocidade constante.

5. **`setFromValue(0)`**: Este método define o valor inicial do fade. O argumento passado indica a opacidade inicial do nó. No caso do valor `0`, significa que o nó estará completamente transparente no início da transição.

6. **`setToValue(1)`**: Este método define o valor final do fade. O argumento passado indica a opacidade final do nó. No caso do valor `1`, significa que o nó estará completamente visível (opaco) no final da transição.

7. **`play()`**: Este método inicia a animação da transição de fade. Depois de ter configurado todos os parâmetros, você chama este método para começar a animação de fade.

Agora vamos entender sobre ScaleTransitions:

```
// Scale:
ScaleTransition scale = new
ScaleTransition();
scale.setNode(image);
scale.setDuration(Duration.millis
(ms:1000));
scale.setCycleCount(ScaleTransition.
INDEFINITE);
scale.setInterpolator(Interpolator.
LINEAR);
scale.setAutoReverse(value:true);
scale.setByX(value:1.3);//Para
diminuir use 0.5
scale.setByY(value:1.3);//Para
diminuir use 0.5
scale.play();
```

Aqui está a explicação detalhada dos métodos usados para configurar a transição de escala (Scale):

1. **`setNode(image)`**: Este método define o nó (Node) ao qual a transição de escala será aplicada. Você precisa passar o objeto que deseja aplicar a transformação de escala como argumento para este método. No exemplo dado, parece que `image` é o nó ao qual a transformação de escala será aplicada.

2. **`setDuration(Duration.millis(1000))`**: Este método define a duração da transição de escala. Você precisa passar um objeto `Duration` como argumento, especificando a quantidade de tempo que deseja que a escala leve. Neste caso, `Duration.millis(1000)` especifica que a escala deve ocorrer ao longo de 1000 milissegundos (ou 1 segundo).

3. **`setCycleCount(ScaleTransition.INDEFINITE)`**: Este método define quantas vezes a transição de escala será repetida. Você pode passar um valor numérico (inteiro) ou `ScaleTransition.INDEFINITE` se quiser que a escala seja repetida indefinidamente. No exemplo, está configurado para ser executado indefinidamente.

4. **`setInterpolator(Interpolator.LINEAR)`**: Este método define o interpolador usado para calcular os valores intermédios durante a transição. O interpolador afeta a velocidade da animação ao longo do tempo. `Interpolator.LINEAR` indica um movimento linear, ou seja, a escala ocorre a uma velocidade constante.

5. **`setAutoReverse(true)`**: Este método define se a transição deve ser revertida automaticamente após a conclusão. Se configurado como `true`, após a escala completa, a animação irá inverter automaticamente e retornar à sua escala original.

6. **`setByX(1.3)`**: Este método define a quantidade pela qual o nó será aumentado na direção horizontal (eixo X). O argumento passado indica o fator de escala. No caso de `1.3`, significa que o nó será aumentado em 30% na direção horizontal.

7. **`setByY(1.3)`**: Este método define a quantidade pela qual o nó será aumentado na direção vertical (eixo Y). O argumento passado indica o fator de escala. No caso de `1.3`, significa que o nó será aumentado em 30% na direção vertical.

8. **`play()`**: Este método inicia a animação da transição de escala. Depois de ter configurado todos os parâmetros, você chama este método para começar a animação de escala.

É importante citar que podemos usar mais de um tipo de animations desse tipo juntos, para assim realizar uma animação de rotação que aumenta e diminui misturando Rotate com Scale por exemplo.

Agora vamos usar animações mais complexas que servem para toda a sua aplicação e não apenas para um node, para isso vamos utilizar outros tipos de Animations, primeiramente vamos dar uma descrição geral de cada uma delas:

O JavaFX fornece várias classes para suportar animações e temporizadores. Algumas das principais classes relacionadas a animação e temporização em JavaFX são essas:

Timeline: Permite criar animações sequenciais com eventos (chamados KeyFrames) em momentos específicos durante a animação.
Pacote: javafx.animation.
Uso Típico: Criar sequências de animação com transições suaves.

KeyFrame: Representa um momento específico em uma Timeline onde um evento ocorre.
Pacote: javafx.animation.
Uso Típico: Especificar eventos em momentos específicos durante uma animação.

AnimationTimer: Fornece um mecanismo para executar código em intervalos regulares de quadros de animação.
Pacote: javafx.animation.
Uso Típico: Atualizar continuamente a interface gráfica ou realizar cálculos em cada quadro de animação.

PauseTransition: Implementa uma transição que pausa a execução por um determinado período de tempo.
Pacote: javafx.animation.
Uso Típico: Introduzir pausas entre diferentes estágios de uma animação.

PathTransition: Esta animação permite mover um nó ao longo de um caminho definido por um objeto Path. O nó segue o caminho especificado durante a duração da animação.

StrokeTransition: Esta animação permite alterar as propriedades de contorno (como cor ou largura da linha) de um nó ao longo do tempo.

FillTransition: Similar ao StrokeTransition, mas esta animação permite alterar as propriedades de preenchimento (como cor de fundo) de um nó ao longo do tempo.

ParallelTransition: Esta classe permite combinar várias transições em paralelo, de modo que elas ocorram simultaneamente.

SequentialTransition: Por outro lado, esta classe permite agrupar várias transições em sequência, de modo que elas ocorram uma após a outra.

Mas, quais são as mais utilizadas para as aplicações que nós iremos construir na matéria de Programação Concorrente:

Para animações em programação concorrente, onde várias animações ou processos precisam ser executados simultaneamente ou coordenados de forma assíncrona, os tipos de animação mais utilizados no JavaFX são:

1. **ParallelTransition**: Este tipo de animação é especialmente útil em programação concorrente, pois permite executar várias transições simultaneamente. Você pode agrupar várias transições em um `ParallelTransition` e todas elas serão executadas de forma paralela, o que é essencial quando você tem várias partes da interface do usuário que precisam ser animadas ao mesmo tempo.

2. **SequentialTransition**: Embora menos comum para animações em programação concorrente, `SequentialTransition` também pode ser útil quando você precisa coordenar várias transições ou eventos de forma sequencial. Por exemplo, você pode definir uma sequência de transições onde cada transição começa após a conclusão da anterior.

3. **Timeline e KeyFrame**: Embora não sejam diretamente voltados para programação concorrente, `Timeline` e `KeyFrame` podem ser utilizados para coordenar eventos em momentos específicos durante uma animação. Você pode definir vários keyframes em uma `Timeline`, cada um representando um momento específico durante a animação, e coordenar eventos ou mudanças de estado em diferentes partes da interface do usuário.

Esses tipos de animação oferecem maneiras eficazes de lidar com animações em programação concorrente no JavaFX, permitindo que você crie interfaces gráficas dinâmicas e responsivas mesmo quando múltiplos processos ou eventos estão ocorrendo simultaneamente.

Porém para controle extremamente específico das animações em concorrente nós teremos que usar primeiramente a classe AnimationTimer executando métodos que controlam o X e Y do nosso objeto que vai se movimentar, e futuramente utilizaremos as funções dos Threads juntamente com métodos de rotação de imagem com for para incrementar pixels a nossa imagem, dessa forma a execução das nossas animações ficará de forma extremamente precisa e principalmente com concorrência.

Porém agora vamos voltar aos nossos Tipos de animação utilizados no começo, primeiramente vamos estudar mais sobre TimeLine e KeyFrames:

Criando um Timer com o nosso TimeLine:

```java
@FXML
private Label label;

int i = 0;

@Override
public void initialize(URL location,
ResourceBundle resources) {
    label.setText(String.valueOf(i));

    Timeline timeline = new Timeline(new
    KeyFrame(Duration.seconds(s:1), e ->
    {
        i++;
        label.setText(String.valueOf(i));
        if (i == 10) {
            System.out.println
            (x:"Alarme");
        }
    }));

    timeline.setCycleCount(Timeline.
    INDEFINITE);
    timeline.play();
}
```

Aqui basicamente temos uma TimeLine com um KeyFrame, primeiro setamos no KeyFrame o seu Duration, que será o tempo entre suas execuções, então por esse meio teremos o controle sobre o tempo, pois o KeyFrame vai executar sua lambda function a cada 1 segundo.
Nessa lambda function eu incremento o valor da minha variável de controle e atualizo a minha label com o valor do i, dessa forma o TImeLine executa esse KeyFrame a cada 1 segundo e assim fazendo toda lógica do nosso timer.
Para que o Timer seja “Infinito” eu preciso declarar o .setCycleCount do meu time line como INDEFINITE.

Dessa forma temos um maior entendimento sobre o controle de tempo do TimeLine e seus KeyFrames:

Agora um exemplo de como adicionar vários KeyFrames na nossa TimeLine:

```
// Criar uma linha do tempo (Timeline)
Timeline timeline = new Timeline();

// Adicionar vários KeyFrames à linha do tempo
timeline.getKeyFrames().addAll(
        new KeyFrame(Duration.ZERO, e -> { //
        KeyFrame inicial
            circle.setFill(Color.BLUE);
            circle.setTranslateX(value:0);
        }),
        new KeyFrame(Duration.seconds(s:1), e ->
        { // KeyFrame após 1 segundo
            circle.setFill(Color.RED);
            circle.setTranslateX(value:100);
        }),
        new KeyFrame(Duration.seconds(s:2), e ->
        { // KeyFrame após 2 segundos
            circle.setFill(Color.GREEN);
            circle.setTranslateX(value:200);
        }),
        new KeyFrame(Duration.seconds(s:3), e ->
        { // KeyFrame após 3 segundos
            circle.setFill(Color.YELLOW);
            circle.setTranslateX(value:300);
        })
);
```

É importante ressaltar que assim cada KeyFrame é realizado no seu tempo Duration especificado, logo quando se passa 1 segundo o primeiro executa, em 2 segundos o segundo e assim sucessivamente, mas como decretamos o TimeLine para se repetir, essa diferença de segundo continua, então em 4 segundos ele volta ao KeyFrame Incial e assim segue o ciclo.

O timeLine tem algumas especificações quando utilizados, podemos dar a preferência ao timeLine para criar animações quando podemos controlar o número de vezes que aquela execução será feita ou a ordem de execução de uma animação que se repete porém de forma controlada baseada no tempo. Um exemplo é o próprio código acima, em um AnimationTimer

sem KeyFrames isso aconteceria tão rápido que o usuário não ia conseguir ver porém em uma TimeLine temos o intervalo de 1 segundo até cada KeyFrame executar nela.

Outro exemplo de uso de uma TimeLine é que podemos determinar o número de ciclos que ela se repete, isso pode ser controlado por variáveis externas, por exemplo se temos um jogo que precisa executar uma timeLine para cada elemento em um ArrayList, podemos utilizar: timeline.setCycleCount(arraylist.size());, dessa forma ele irá executar para o número de elementos dentro do arrayList.

Uma classe geralmente utilizada com as TimeLines é a PauseTransition ### Classe `PauseTransition`:

Principais Usos:

1. **Pausa na Execução:** Utilizada para introduzir uma pausa ou atraso antes da execução de determinadas ações.

2. **Temporização de Eventos:** Útil para temporizar a execução de eventos em um programa.

Principais Métodos:

1. **Construção:**

   PauseTransition pause = new PauseTransition(Duration.seconds(1));

   - Cria uma transição de pausa com a duração especificada.

2. **Adição de Evento de Conclusão:**

   pause.setOnFinished(e -> {
       // Código a ser executado quando a pausa terminar
   });

   - Adiciona um evento que será executado quando a pausa terminar.

3. **Iniciar a `PauseTransition`:**

   pause.play();

   - Inicia a transição de pausa.

Ela resumidamente serve para criar uma pausa entre as execuções e após essa pausa um código é executado, isso serve por exemplo para dar um tempo entre cada execução gerando um certo controle de segurança de código, um exemplo seria: se o usuário clica em algo que muda de cor e o código é controlado principalmente por cores, logo o código necessita de uma pausa entre cada execução para não obter a cor errada no momento do clique.

De forma resumida a TimeLine é um meio de se criar animações com duração mais específica entre cada execução dos seus KeyFrames, como também pode ser usado junto a pausas para transições corretas em um código, o TimeLine será usado quando o tempo e ordem de execução for de suma importância:

Animation Timer:

A classe AnimationTimer é uma parte da API de animação do JavaFX e é utilizada para criar animações baseadas em quadros (frames) em uma aplicação gráfica. Ela é especialmente útil quando você precisa realizar ações contínuas e repetitivas, como a atualização da interface

gráfica, execução de lógica de jogo ou qualquer atividade que precise ser repetida em cada quadro da animação.

Principais Aspectos da Classe AnimationTimer**:**

- **Método** handle(long timestamp)**:**
  - O método handle é onde você coloca o código que será executado em cada quadro da animação. Ele recebe um argumento timestamp, que representa o tempo atual em nanossegundos. Esse timestamp pode ser utilizado para calcular variações de tempo e controlar a lógica da animação.

- **Método** start()**:**
  - Inicia a execução do AnimationTimer. Quando você chama timer.start(), o método handle é chamado repetidamente em intervalos regulares.

- **Método** stop()**:**
  - Para a execução do AnimationTimer. Isso impede a chamada contínua do método handle. Quando você chama timer.stop(), a animação é interrompida.

Uso Típico:

- **Atualização Contínua:**
  - O AnimationTimer é frequentemente usado para atualizar continuamente a interface gráfica em cada quadro. Por exemplo, você pode usar um AnimationTimer para mover objetos, alterar cores ou realizar outras alterações visuais.

- **Lógica de Jogo:**
  - É comum usar um AnimationTimer em jogos para lidar com a lógica de jogo que precisa ser atualizada em cada quadro. Isso pode incluir movimento de personagens, detecção de colisões, processamento de entrada, entre outras coisas.

- **Simulações e Gráficos em Tempo Real:**
  - Em aplicações que envolvem simulações físicas ou gráficos em tempo real, o AnimationTimer pode ser usado para calcular e exibir alterações dinâmicas.

Vamos esclarecer algumas dúvidas que geralmente aparecem:

1. **O que é o timestamp?**

   - O `timestamp` é um valor representando o tempo atual em nanossegundos no momento em que o método `handle` do `AnimationTimer` é chamado. Ele é uma medida do tempo desde um ponto de referência específico (geralmente o início da execução do programa) até o momento atual.

2. **O AnimationTimer realiza as linhas de código dele a cada nanossegundo?**

   - Não exatamente. O `AnimationTimer` chama o método `handle` a uma taxa específica, mas a frequência exata pode variar dependendo das capacidades do hardware e das condições do sistema. O objetivo é realizar a atualização em cada quadro de animação, mas o tempo entre os quadros pode variar. O JavaFX tenta ajustar automaticamente a taxa de chamada do `handle` para proporcionar uma animação suave e eficiente.

3. **Posso escolher de quanto em quanto tempo o AnimationTimer irá realizar seu loop?**

   - Não diretamente. O `AnimationTimer` é projetado para ser sincronizado com a taxa de quadros do sistema. Em vez de configurar explicitamente a taxa de chamada, o JavaFX tenta ajustar dinamicamente para coincidir com a taxa de atualização da tela. Isso proporciona uma experiência de animação suave e eficiente.

Se você precisar de um temporizador que execute a lógica em intervalos específicos e regulares, talvez seja mais apropriado usar uma classe como `Timeline` ou `ScheduledExecutorService`. Essas classes permitem que você defina intervalos específicos para a execução de tarefas.

Para um controle mais preciso sobre o tempo entre quadros no `AnimationTimer`, você pode calcular a diferença de tempo (delta) entre quadros sucessivos usando o `timestamp`. Essa abordagem é comumente usada para ajustar movimentos ou animações com base no tempo decorrido entre os quadros. Porém a AnimationTImer é geralmente utilizada para o controle de animação dos objetos em um ciclo de cadeia de frames, dessa forma ela geralmente não é usada para controle sequencial baseado em tempo, ela é usada para uma animação fluida, que a cada segundo pode realizar lógicas em códigos, como lógica de colisão entre objetos ou leituras de entrada de teclado para movimento de algum Node.

Além disso geralmente o AnimationTimer é mais utilizado para animações em concorrência, afinal podemos criar um timer específico para diferentes objetos na nossa aplicação e fazer com que eles sejam independentes um do outro, ou se comunicarem, controlando um a condição para a execução do outro, agora vamos ver alguns exemplos diversos e principais formas de controles entre AnimationTimers:

Primeiramente vamos entender como usar AnimationTImers para controle de movimento de Nodes juntamente com cliques de teclado:

Para isso primeiro precisamos entender BooleanBindins e BooleanPropertys:

- **(BooleanProperty):** Propriedades booleanas que indicam valores de verdadeiro ou falso para suas variáveis, elas devem ser declaradas como BooleanProperty “nome” = SimpleValueProperty(), dessa forma apenas recebendo true ou false.
- **(BooleanBinding):** Uma ligação booleana que é uma propriedade que pode ser observada, nesse caso você cria uma ligação entre BooleanPropertys, e essa ligação tem diferentes aspectos lógicos, você pode relacionar 2 BolleanPropertys de forma a se uma for verdadeira o BooleanBinding também será, ou pode relacionar para apenas uma ser verdadeiro, além disso o booleanBinding pode receber um Listener, sendo assim ele pode ser analisado a cada mudança, e a cada mudannça ele pode executar algo para true e algo para false, por isso geralmente usamos eles, para em seu listener quando for true ele executar a animation e quando for false ele parar a animation.

Vamos detalhar um pouco mais sobre a importância do Boolean em animations:

A classe `BooleanBinding` em JavaFX é uma subclasse abstrata de `BooleanExpression` que representa uma expressão booleana que pode ser avaliada e vinculada a outras propriedades booleanas. Ela é útil para criar ligações (bindings) e expressões booleanas complexas em aplicações JavaFX.

Aqui estão algumas das funcionalidades da classe `BooleanBinding`:

1. **Ligação (Binding) de Propriedades**: Você pode criar uma ligação entre duas ou mais propriedades booleanas usando métodos como `and`, `or`, `not`, etc. Por exemplo:

```java
BooleanProperty bool1 = new SimpleBooleanProperty(true);
BooleanProperty bool2 = new SimpleBooleanProperty(false);

BooleanBinding binding = bool1.and(bool2);
```

2. **Listeners**: Assim como as propriedades, você pode adicionar ouvintes para observar mudanças no valor da expressão booleana. Por exemplo:

```java
binding.addListener((observable, oldValue, newValue) -> {
```

```
    System.out.println("Valor da expressão alterado para: " + newValue);
});
```

3. **Atualização Automática**: As ligações são atualizadas automaticamente sempre que os valores das propriedades vinculadas mudam. Isso significa que você não precisa se preocupar em atualizar manualmente a expressão; ela é recalculada automaticamente.

4. **Expressões Complexas**: Você pode criar expressões booleanas complexas combinando ligações e operadores booleanos. Por exemplo:

```java
BooleanBinding complexBinding = bool1.and(bool2.or(bool3)).not();
```

Isso cria uma expressão booleana que é o resultado da negação de `bool1` e `bool2` ou `bool3`.

5. **Suporte a Fluent API**: A classe `BooleanBinding` e suas subclasses geralmente suportam uma API fluente, o que significa que você pode encadear chamadas de método para criar expressões mais complexas de forma mais legível.

```java
BooleanBinding complexBinding = bool1.and(bool2).or(bool3).not();
```

Essas são algumas das funcionalidades da classe `BooleanBinding` em JavaFX. Ela é especialmente útil quando você precisa criar ligações entre propriedades booleanas e expressões booleanas complexas em suas aplicações JavaFX.

Essas classes podem ser linkadas de diferentes formas com nossas animações, porém o mais importante é o uso do .pauseProperty() em seu timer, que pode ser ligado ao seu timer com condições da classe BooleanExpression utilizando Bindngs.

Aqui está uma possível implementação:

```java
import javafx.beans.binding.Bindings;
import javafx.beans.property.BooleanProperty;
import javafx.beans.property.SimpleBooleanProperty;

public class Controller_tela2 implements Initializable {

    // Outros códigos da classe...

    private BooleanProperty trem1InCriticalZone1 = new SimpleBooleanProperty(false);
    private BooleanProperty trem2InCriticalZone1 = new SimpleBooleanProperty(false);
    private BooleanProperty trem1InCriticalZone2 = new SimpleBooleanProperty(false);
    private BooleanProperty trem2InCriticalZone2 = new SimpleBooleanProperty(false);

    private BooleanProperty trem1EnteredFirstCriticalZone1 = new SimpleBooleanProperty(false);
    private BooleanProperty trem2EnteredFirstCriticalZone1 = new SimpleBooleanProperty(false);
    private BooleanProperty trem1EnteredFirstCriticalzone2 = new SimpleBooleanProperty(false);
    private BooleanProperty trem2EnteredFirstCriticalzone2 = new SimpleBooleanProperty(false);

    // Outros códigos da classe...

    public void colisionControl() {
        BooleanBinding colisaoCritica1 = trem1InCriticalZone1.and(trem2InCriticalZone1);
        BooleanBinding colisaoCritica2 = trem1InCriticalZone2.and(trem2InCriticalZone2);
```

```
        timer_train1.stop();
        timer_train2.stop();

        if (!colisaoCritica1.get() && !colisaoCritica2.get()) {
            timer_train1.start();
            timer_train2.start();
        } else {
            timer_train1.pauseProperty().bind(Bindings.when(colisaoCritica1)
                    .then(trem1EnteredFirstCriticalZone1.not())
                    .otherwise(false));

            timer_train2.pauseProperty().bind(Bindings.when(colisaoCritica1)
                    .then(trem2EnteredFirstCriticalZone1.not())
                    .otherwise(false));

            timer_train1.pauseProperty().bind(Bindings.when(colisaoCritica2)
                    .then(trem1EnteredFirstCriticalzone2.not())
                    .otherwise(false));

            timer_train2.pauseProperty().bind(Bindings.when(colisaoCritica2)
                    .then(trem2EnteredFirstCriticalzone2.not())
                    .otherwise(false));
        }
    }

    // Outros códigos da classe...
}
```

Nesta implementação:

- `BooleanProperty trem1EnteredFirstCriticalZone1`, `trem2EnteredFirstCriticalZone1`, `trem1EnteredFirstCriticalzone2` e `trem2EnteredFirstCriticalzone2` são usados para rastrear qual trem entrou primeiro em cada zona crítica.

- `BooleanBinding colisaoCritica1` e `colisaoCritica2` são utilizadas para determinar se há colisões em cada zona crítica.

- Os `AnimationTimers` são pausados ou retomados usando `pauseProperty()` de acordo com as condições de colisão e prioridade dos trens.

Portanto temos o controle da animação por BooleanExpressions.

Existem outras classes relacionadas que são úteis para trabalhar com expressões booleanas e propriedades observáveis. Aqui estão algumas delas:

- **BooleanProperty**: Esta classe representa uma propriedade booleana observável. Ela estende Property<Boolean> e pode ser vinculada a outras propriedades, ouvintes podem ser adicionados para monitorar mudanças em seu valor e ela pode ser usada em expressões booleanas.
- **BooleanPropertyBase**: É uma classe abstrata que fornece uma implementação básica de BooleanProperty. Você pode estender esta classe para criar suas próprias propriedades booleanas personalizadas, se necessário.
- **ReadOnlyBooleanProperty**: Uma versão somente leitura de BooleanProperty. Você pode usar essa classe quando desejar expor uma propriedade booleana publicamente, mas não deseja permitir que outros componentes modifiquem seu valor.
- **ReadOnlyBooleanPropertyBase**: Similar a BooleanPropertyBase, mas para propriedades somente leitura.

- **BooleanExpressionBase**: Uma classe abstrata que fornece uma implementação básica de BooleanExpression. Você pode estender esta classe para criar suas próprias expressões booleanas personalizadas, se necessário.
- **BooleanBindingBase**: Similar a BooleanExpressionBase, mas para ligações booleanas. Você pode estender esta classe para criar suas próprias ligações booleanas personalizadas.
- **BooleanBinding**: Já mencionada anteriormente, é uma subclasse de BooleanExpression que representa uma expressão booleana que pode ser avaliada e vinculada a outras propriedades booleanas.

Essas classes fornecem uma variedade de funcionalidades para trabalhar com expressões e propriedades booleanas em JavaFX, permitindo uma programação mais modular e orientada a objetos ao lidar com interfaces gráficas de usuário e outras aplicações que envolvam lógica booleana.

Agora vamos seguir com nosso AnimationTImer:

Em JavaFX, para criar um `AnimationTimer`, você precisa estender essa classe abstrata e implementar o método `handle(long now)`. Este método é chamado em cada quadro de animação, fornecendo o tempo atual em nanossegundos. Aqui estão os passos para criar um `AnimationTimer`:

1. **Estenda a classe `AnimationTimer`**: Crie uma classe que estenda `AnimationTimer`.

2. **Implemente o método `handle`**: Implemente o método `handle(long now)`. Este método será chamado em cada quadro de animação.

3. **Crie uma instância do seu `AnimationTimer`**: Crie uma instância da sua classe `AnimationTimer` personalizada.

4. **Inicie o `AnimationTimer`**: Inicie o `AnimationTimer` chamando o método `start()`.

Aqui está um exemplo básico de como criar e usar um `AnimationTimer`:

```java
import javafx.animation.AnimationTimer;

public class MyAnimationTimer extends AnimationTimer {
    @Override
    public void handle(long now) {
        // Lógica de animação aqui
        System.out.println("Frame de animação");
    }
}

public class Main {
    public static void main(String[] args) {
        MyAnimationTimer animationTimer = new MyAnimationTimer();
        animationTimer.start();
    }
}
```

Este é um exemplo muito simples que apenas imprime uma mensagem a cada quadro de animação. Na prática, você implementaria a lógica de animação dentro do método `handle`.

Agora, sobre criar um `AnimationTimer` que é seu, você pode querer dizer que deseja ter controle sobre o tempo de execução e talvez querer pausá-lo, retomá-lo, ou pará-lo em um determinado momento. Isso é perfeitamente possível com a classe `AnimationTimer`, mas requer um pouco mais de lógica.

Aqui está um exemplo de como você poderia criar um `AnimationTimer` personalizado com métodos para pausar, retomar e parar:

```java
import javafx.animation.AnimationTimer;

public class MyCustomAnimationTimer extends AnimationTimer {
    private boolean running = false;
    private long lastTime = 0;

    @Override
    public void start() {
        lastTime = System.nanoTime();
        running = true;
        super.start();
    }

    @Override
    public void stop() {
        running = false;
        super.stop();
    }

    public void pause() {
        running = false;
    }

    public void resume() {
        running = true;
        lastTime = System.nanoTime();
    }

    @Override
    public void handle(long now) {
        if (running) {
            long deltaTime = now - lastTime;
            // Lógica de animação usando deltaTime, se necessário
            lastTime = now;
        }
    }
}
```

Neste exemplo, `MyCustomAnimationTimer` estende `AnimationTimer` e adiciona lógica para pausar, retomar e parar o timer. O método `handle` agora também verifica se o timer está sendo executado antes de atualizar a lógica de animação. Isso permite um controle mais granular sobre o comportamento do timer.

Controlando Nodes:

Para controlar Nodes e executar suas animações vamos utilizar basicamente seus Layouts X e Y e junto a isso adicionar uma movementVariable:

movementVariable **(int):** Variável que define o quanto o node deve se mover em cada atualização de quadro.
Essa variável é (somada/subtraida) ao Layout X (e/ou) Y , para assim a cada frame de animação o objeto se mover na cena.

Temos alguns ultimos conceitos a se rever, primeiramente para criarmos Animation Timers utilizamos duas formas, Criando uma Classe nossa ou seja criando o MyTimer e extendento de AnimationTimer, ou podemos criar uma “instância” da Classe AnimationTimer de forma anônima, apenas sobreescrevendo métodos obrigatórios, que nesse caso é o handler, vamos ver isso mais detalhadamente:

Vou explicar os dois métodos para criar Animation Timers em detalhes, destacando as diferenças e vantagens de cada um:

Método 1: Criando uma Classe Personalizada (ex. `MyTimer`)

Neste método, você cria uma classe personalizada que estende `AnimationTimer` e implementa o método `handle()`. Aqui está um exemplo básico:

```java
import javafx.animation.AnimationTimer;

public class MyTimer extends AnimationTimer {
    @Override
    public void handle(long now) {
        // Implemente o código que deseja executar em cada frame
    }
}
```

Vantagens:
1. **Clareza e organização:** Ao criar uma classe personalizada, você encapsula toda a lógica relacionada ao timer em um único lugar, tornando o código mais fácil de entender e manter.
2. **Reutilização:** Se você precisar de timers com funcionalidades semelhantes em vários lugares do seu código, pode reutilizar a classe `MyTimer` sem precisar reescrever o código de manipulação de timer.

Desvantagens:
1. **Overhead de Criação de Classe:** Criar uma classe personalizada pode parecer excessivo para casos simples e pode adicionar um pouco de sobrecarga ao código.

Método 2: Criando uma Instância Anônima de `AnimationTimer`

Neste método, você instancia a classe `AnimationTimer` anonimamente, sobrescrevendo o método `handle()`. Aqui está um exemplo:

```java
import javafx.animation.AnimationTimer;

public class Main {
    public static void main(String[] args) {
        AnimationTimer timer = new AnimationTimer() {
            @Override
            public void handle(long now) {
                // Implemente o código que deseja executar em cada frame
            }
        };

        timer.start();
    }
}
```

Vantagens:

1. **Conciso:** Se você precisa de um timer simples e não quer criar uma classe separada apenas para isso, este método é mais conciso.
2. **Localidade do Código:** O código do timer está diretamente no local onde é utilizado, o que pode facilitar a compreensão do fluxo de execução.

Desvantagens:

1. **Reutilização Limitada:** Como a instância do timer é criada localmente, não é fácil reutilizá-la em outros lugares do código sem copiar e colar.
2. **Menos Organizado:** Para código mais complexo, pode tornar o código menos legível e organizado, especialmente se o manipulador do timer for longo.

Escolha do Método:

A escolha entre esses métodos depende do contexto e da complexidade do código. Para timers simples e usos únicos, criar uma instância anônima pode ser mais conveniente. Para casos mais complexos ou onde a reutilização é provável, ou existe a necessidade de se sobrescrever mais métodos como o start e stop e criar mais métodos como o resume e o pause, criar uma classe personalizada é preferível por razões de clareza e organização.

Convertendo de uma classe de instância anônima para uma classe dedicada:

Para converter os timers de instância anônima para uma classe que estende `AnimationTimer`, você precisará criar uma classe separada que estenda `AnimationTimer` e sobrescrever o método `handle()` nessa classe. Aqui está como você pode fazer isso:

1. Crie uma nova classe que estenda `AnimationTimer`:

```java
import javafx.animation.AnimationTimer;

public class MyAnimationTimer extends AnimationTimer {

    private Controller_tela2 controller; // Referência ao controlador para acessar seus métodos e variáveis

    // Construtor que recebe uma referência ao controlador
    public MyAnimationTimer(Controller_tela2 controller) {
        this.controller = controller;
    }

    @Override
    public void handle(long now) {
        // Chame o método do controlador que deseja executar no timer
        controller.handleTimer();
    }
}
```

2. No seu controlador (`Controller_tela2`), declare uma instância da nova classe `MyAnimationTimer`:

```java
private MyAnimationTimer timer_train1;
private MyAnimationTimer timer_train2;
```

3. Inicialize os timers na função `initialize()` do seu controlador:

```java
```

```
@Override
public void initialize(URL location, ResourceBundle resources) {
    // Outros códigos de inicialização...

    // Inicialize os timers com uma instância da classe MyAnimationTimer, passando o próprio
controlador como parâmetro
    timer_train1 = new MyAnimationTimer(this);
    timer_train2 = new MyAnimationTimer(this);

    // Outros códigos de inicialização...
}
```

4. Crie um método no seu controlador para lidar com a lógica que você quer executar nos timers:

```java
public void handleTimer() {
    // Lógica que você quer executar no timer
    trem1.setSpeed(slider_trem1.getValue() * movement_variable);
    trainMovementController.moveTrain(trem1);

    // Outras operações se necessário...
}
```

Com isso, você converteu os timers de instância anônima para uma classe separada que estende `AnimationTimer`. Certifique-se de adaptar a lógica do timer de acordo com suas necessidades específicas.

Dessa forma citamos algumas das características do nosso AnimationTimer.

#ID