Interfaces graphiques - JavaFX

responsable : Wiesław Zielonka
zielonka@liafa.univ-paris-diderot.fr
https://www.irif.univ-paris-diderot.fr/~zielonka

March 29, 2016

Qu'est-ce c'est une animation?

Animation – changements de propriétés JavaFX d'un ou plusieurs objets de façon continue dans le temps.

Animation avec `Timeline`

`Timeline` est une classe dérivée de `Animation`. `Timeline` permet de faire des animations customisées qui ne sont pas implémentées par les Transitions.

Propriétés héritées de `Animation`:
`autoReverse` `currentRate` `currrentTime` `cycleCount` `cycleDuration` `delay` `rate` `status` `totalDuration`

Méthodes : `play()` `stop()` `pause()`

Chaque `Timeline` est composé d'un ou plusieurs `KeyFrame`s.

KeyFrame

`KeyFrame` possède les constructeurs suivants:

```
KeyFrame(Duration time,
         EnvetHandler<ActionEvent> onFinished,
         KeyValue... values)

KeyFrame(Duration time, KeyValue... values)
```

Si `EventHadler onFinished()` est spécifié ce handler sera exécuté à la fin l'animation définie par `KeyFrame`.

Chaque `KeyValue` décrit une propriété et la valeur finale (à la fin de l'animation) de cette propriété.

KeyValue

Constructeur :

```
KeyValue(WritableValue<T> target, T endValue)
```

`target` – la propriété modifiée pendant l'animation
`endValue` – la valeur finale de cette propriété

Timeline

```
// KeyFrame de la duree 0 juste pour initaliser les propriete
KeyFrame keyFrameInit = new KeyFrame(Duration.ZERO,
    new KeyValue(ellipse.strokeWidthProperty(), 1),
    new KeyValue(ellipse.strokeProperty(), Color.AQUA),
    new KeyValue(ellipse.fillProperty(), Color.RED),
    new KeyValue(ellipse.radiusXProperty(), 100),
    new KeyValue(ellipse.radiusYProperty(), 100)
);

//KeyFrame finale pour les valeurs finales de proprietes
KeyFrame keyFrameFin = new KeyFrame(Duration.millis(4000),
    new KeyValue(ellipse.strokeWidthProperty(), 30),
    new KeyValue(ellipse.strokeProperty(), Color.RED),
    new KeyValue(ellipse.fillProperty(), Color.AQUA),
    new KeyValue(ellipse.radiusXProperty(), 300),
    new KeyValue(ellipse.radiusYProperty(), 50)
);

Timeline timeline = new Timeline();
timeline.getKeyFrames().addAll(keyFrameInit, keyFrameFin);
timeline.play();
```

Interface `Property<T>`

L'interface `Property<T>` possède les méthodes :

- `void bind(ObservableValue<? extends T> observable)` – création de lien unidirectionnel avec cette propriété,
- `void bindBidirectional(Property<T> other)` – création de lien bidirectionnel avec la propriété,
- `void unbind()` – suppression de lien unidirectionnel,
- `void unbindBidirectional(Property<T> other)` – suppression de lien bidirectionnel avec la propriété.

Une propriété lié à une autre

```
Rectangle rectangle = new Rectangle();
Ellipse ellipse = new Ellipse();

rectangle.widthProperty().bind(ellipse.radiusXProperty());
```

La propriété `width` du rectangle est lié à la propriété `radiusX` de l'ellipse.
Les deux propriétés sont de type `DoubleProperty` et cette classe implémente les interfaces

- `Property<Number>` – donc la propriété `width` implémente la méthode `bind()`, et
- `ObservableValue<Number>` – donc la propriété `radiusX` peut être utilisée comme argument de `bind()`.

Construire des liens plus compliqués entre les propriétés

```
rectangle.widthProperty().bind(ellipse
                                   .radiusXProperty()
                                   .multiply(2));
```

la propriété `width` du rectangle sera toujours égale à deux fois la propriété `radiusX` de l'ellipse.

Explications

`ellipse.radiusXProperty()` est une `DoubleProperty` et hérite les méthodes :

- `DoubleBinding add(double other)`
- `DoubleBinding divide(double other)`
- `DoubleBinding multiply(double other)`
- `DoubleBinding substract(double other)`
- `DoubleBinding negate()`

dont le résultat peut être utilisé comme argument de la méthode `bind()`.

Construire des liens plus compliqués entre les propriétés

Pour obtenir la valeur de la propriété `x` du rectangle maintient le lien :

$$\texttt{x} \leftarrow \texttt{centerX - radiusX}$$

avec les propriété `centerX` et `radiusX` de l'ellipse :

```
rectangle.xProperty().bind(Bindings
        .subtract(ellipse.centerXProperty(),
                  ellipse.radiusXProperty()));
```

Justification

La classe `Bindings` possède les méthodes statiques :

- `static NumberBinding add(ObservableNumberValue op1, ObservableNumberValue op2)`
- `static NumberBinding multiply(ObservableNumberValue op1, ObservableNumberValue op2)`
- `static NumberBinding substract(ObservableNumberValue op1, ObservableNumberValue op2)`
- `static NumberBinding divide(ObservableNumberValue op1, ObservableNumberValue op2)`

qui permettent de combiner deux propriété numériques.
`Bindings` possède de dizaines d'autres méthodes statiques dont le résultat peut être NumberBinding ou BooleanBinding etc. Le résultats de ces méthodes peuvent être utilisés comme l'argument de `bind()`.

bind() avec When

```
Button starStop = new Button();

startStop.textProperty()
    .bind(new When(timeline.statusProperty()
                        .isEqualTo(Animation.Status.RUNNING))
            .then("pause").otherwise("run"));
```

Pourquoi `isEqualTo()` et pas `equals()` ?
Réponse : `isEqualTo()` retourne `BooleanBinding` et `equals()` retourne boolean.

Le constructeur :

```
When(ObservableBooleanvalue condition)
```

When then otherwise

La classe `When` possède plusieurs méthodes `then` :

- `When.BooleanConditionBuilder then(boolean cond)`
- `When.NumberConditionBuilder then(double thenVal)`
- `When.StringConditionBuilder then(String thenVal)`

et bien d'autres.

L'objet de la classe `When.???Builder` retourné par `then()` contient la méthode `otherwise()` dont le résultat est de type `???Binding` approprié.

AnchorPane

Permet d'accrocher les bord des enfants aux bord de AnchorPane.

Les contraintes :

- `topAnchor`
- `leftAnchor`
- `bottomAnchor`
- `rightAnchor`

Les contraintes indiquées par des méthodes statiques dont le premier argument est un composant et le deuxième la distance entre le bord de `AnchorPane` et le composant :

```
AnchorPane anchorPane=new AnchorPane();
AnchorPane.setTopAnchor(group, 10.0);
AnchorPane.setBottomAnchor(startStop, 10.0);
AnchorPane.setLeftAnchor(startStop, 20.0);
anchorPane.getChildren().addAll(group,startStop);
```

Plusieurs enfants peuvent avoir les mêmes contraintes.

Concurrence et JavaFX

les animations permettent de faire les changements fluides de propriétés.
Mais les animations sont inappropriées si les modifications sont effectuées de façon périodique mais mais non fluide, par exemple chaque 2 secondes.

Le thread d'application est le thread principale qui affiche la scène.
Dans le thread d'application il faut éviter

- de long calcul,
- jamais utiliser `sleep()`.

Si on ne respecte pas ces consignes l'interface devient non réactive.

Package javafx.concurrent

Ce package contient une interface

`Worker<T>`

avec deux implémentations

`Task<V>` et `Service<V>`

Les implémentation de Worker effectue une tâche en arrière plan dans un thread différent du thread de l'application.
Les états possibles de Worker :

- Worker.State.READY - l'état initial,
- Worker.State.SCHEDULED - soumis à l'exécution mais pas encore exécuté,
- Worker.State.RUNNING – en train d'être exécuté
- Worker.State.SUCCEDED – terminé avec succès,
- Worker.State.FAILED – terminé par une exception,
- Worker.State.CANCELLED – terminé par cancel().

Propriétés de Worker

- ReadOnlyStringProperty message
- ReadOnlyDoubleProperty progress
- ReadOnlyBooleanProperty running
- ReadOnlyDoubleProperty totalWork
- ReadOnlyObjectProperty¡V¿ value
- ReadOnlyDoubleProperty workDone

Chaque thread peut lire ces propriétés.

Task<V>

`Task<V>` `Task<V>` est dérivé de `java.util.concurrent.FutureTask` et implémente `Worker<V>`. Taks peut être utilisé une seule fois (pas de réutilisation du même task).

Chaque `Taks<V>` doit implémenter la méthode

```
protected V call()
```

Dans la plupart de cas `call()` ne retourne rien et dans ce cas on définit `Task<Void>` et on termine `Void call()` par

```
return null;
```

Task<V>

Si le task doit modifier la scène ou exécuter un code dans le thread d'application alors on procède de façon suivante :

```
final Group group = new Group();
Task<Void> task = new Task<Void>() {
    @Override protected Void call() throws Exception {
        // le code execute dans le thread arriere plan
        ...
        Platform.runLater(new Runnable(){
            @Override public void run(){

                // ici le code qui modifie la scene
                // par exemple ajoute ou supprime les noeuds
                //group.add(new Rectangle(20,20));

            }
        });

        //encore un code dans le thread erriere plan
}
```

cancel un Task

`cancel()` appliqué à un Task ne le termine pas (pas de méthode fiable de terminer un thread par un autre thread).
Il faut que le task lui même vérifie périodiquement (et à la sortie de `sleep()`) s'il n'est pas cancelled avec la méthode `boolean isCancelled()`.

Démarrer un task

```
Thread th = new Thread(task);
th.setDaemon(true);
th.start();
```

La machine virtuelle java termine si tous les threads restants sont des démons.