WIESLAW ZIELONKA
ZIELONKA@IRIF.FR
WWW.IRIF.FR/~ZIELONKA

INTERFACES GRAPHIQUES

LES INTERFACES DE PROPRIÉTÉS ET DE BINDINGS

Observable
addListener(InvalidationListener void
removeListener(InvalidationListener) void

InvalidationListener
invalidated(Observable) void

ObservableValue<T>
addListener(ChangeListener<? super T>) void
removeListener(ChangeListener<? super T> void
getValue T

ChangeListener<T>
changed(ObservableValue<? extends T>,T,T)

ReadOnlyProperty<T>
getBean() Object
getName() String

Binding<T>
isValid() boolean
invalidate() void
getDependencies() ObservableList<?>
dispose() void

WritableValue<T>
getValue() T
setValue(T) void

Property<T>
bind(ObservableValue<? extends T> void
unbind() void
bindBidirectional(Property <T>) void
unbindBidirectional(Property<T>) void

BINDINGS

- A quoi sert un BINDING ?
  - fait un calcul d'une valeur à partir d'une ou plusieurs sources.
  - les sources sont des dépendances
  - binding observe les dépendances, détecte les changements et met à jour la valeur
- Pourquoi utiliser un binding?
  - permet d'éviter d'écrire les listeners
  - plus concis donc moins susceptible à conduire aux erreurs
  - permet de synchroniser l'interface avec les données

TYPE DE BINDING

- unidirectionnel : met à jour la propriété si la dépendance change
- bidirectionnel : mis à jour de la propriété si la dépendance change et mis à jour de la dépendance si la propriété change
- les méthodes Factory et API de chaînage (fluent API)
    - utilise les librairies des expressions binding
- custom binding (binding sur mesure), si les librairies ne suffisent pas
    - spécifie la propriété et calcule la valeur

binding bidirectionnel

- les méthodes de l'interface Property<T> :

```
void bindBidirectional(Property<T> property)

void unbindBidirectional(Property<T> property)
```

- utilisation

```
property1.bindBidirectional(property2);

property1.unbindBidirectional(property2);
```

la méthode isBound() retourne toujours false si le binding est bidirectionnel.

binding bidirectionnel

- binding bidirectionnel est possible uniquement entre deux propriétés
- si deux propriétés sont liées par binding bidirectionnel alors javas assure que les deux propriétés ont la même valeur :
  - sourceProperty change quand dependantProperty change
  - mais aussi dependantProperty change quand sourceProperty change

- les deux propriétés restent modifiables
- si p1 et p2 sont deux propriétés de même type alors

```
p1.bindBidirectional(p2);
```

et

```
p2.bindBidirectional(p1);
```

ont le même effet.

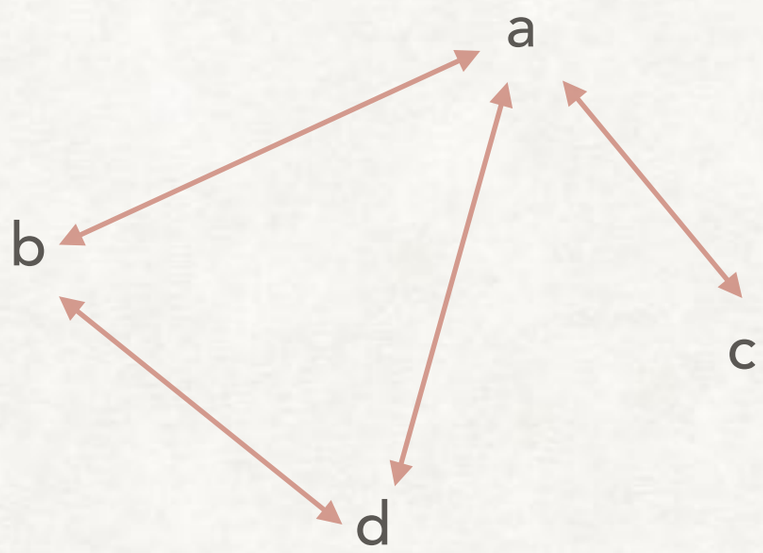
bidirectional binding

```
Circle c1 = new Circle(30);

Circle c2 = new Circle(40);

c1.radiusProperty().bindBidirectional(c2.radiusProperty());

 // c1 40 ;    c2 40

c1.setRadius(60);

  // c1 60 ;   c2 60

c2.setRadius(20);

   // c2 20;    c2 20

c1.radiusProperty().unbindBidirectional(c2.radiusProperty());

c1.setRadius(80);

   // c1 80;   c2 20
```

BIND BIDIRECTIONAL

- bind bidirectional seulement entre les propriétés du même type.

a,b,c,d —— les propriétés (par exemple DoubleProperty)



Les liens représentent bindBidirectional entre différentes propriétés.

- une propriété pout être liée avec plusieurs d'autres par bin bidirectionnel
- le changement de la valeur d'une propriété entraine le changement de valeur de toutes les propriétés liées par bind bidirectionnel

BIND BIDIRECTIONAL (EXEMPLE)

```
Slider slider = new Slider(0, 50, 25);
Spinner spinner = new Spinner(0,50,25.0);

slider.valueProperty()
    .bindBidirectional(spinner.getValueFactory().valueProperty());

//slider et spinner auront tjrs la même valeur
```

Spinner possède la propriété value mais cette propriété est de type ReadOnly donc pas adaptée pour faire bindBidirectional.

Mais Spinner possède valueFactory qui contient le modèle de données pour Spinner, en particulier valueFactory possède la propriété value qui est modifiable.

binding unidirectionnel

- l'interface `Property<T>`

```
void bind(ObservableValue <? extends T> observable)

void unbind()

boolean isBound()
```

- utilisation:

```
sourceProperty.bind(valeurObservable)

sourceProperty.unbind()
```

- `sourceProperty` est mis à jour si `valeurObservable` change mais pas inverse

- quand `sourceProperty` est "bound" alors cette propriété ne peut plus être modifiée directement

Exemple

```
Circle c1 = new Circle(20.0);

Circle c2 = new Circle(30.0);

c2.radiusProperty().bind(c1.radiusProperty());

//c1 et c2 ont le rayon 20.0

c1.setRadius(50.0);

// c1 et c2 : le rayon 50

c2.setRadius(25.0);//exception, radius
//de c2 ne peut pas être modifié explicitement
//s'il est lié par bind()
c2.radiusProperty().unbind();
c2.setRadius(25.0); //maintenant OK
```

Factory Method Binding

Interfaces : `Binding<T>` et `NumberBinding`

Classes abstraites :

IntegerBinding	**FloatBinding**	**DoubleBinding**
BooleanBinding	**LongBinding**	StringBinding
ObjectBinding<T>	ListBinding<E>	SetBinding<E>
MapBinding<K,V>		

implementent `Binding<T>`.

`IntegerBinding, FloatBinding, DoubleBinding, LongBinding` implementent aussi `NumberBinding`

Factory Methods de la classe Bindings

La classe

`Bindings`

possède plus de 200 méthodes static (Factory methods) qui retournent différents bindings du transparent précédent qui peuvent être utilisés pour créer d'autre bindings (par composition) ou comme argument de la méthode `bind()` d'une propriété.

Les méthodes static de la classe Bindings

- arithmétiques et numériques :

add	substrat	multiply
divide	max	min
negate		

- relationnelles et logiques :

and	or	not
equal	notEqual	lessThan
greaterThan	lessThanOrEqual	greaterThanOrEqual
equalIgnoreCase	noEqualIgnoreCase	

LES MÉTHODES STATIC DE LA CLASS BINDINGS

- Collection:

booleanValueAt	doubleValueAt	flowValueAt
integerValueAt	longValueAt	stringValueAt
valueAt	isEmpty	isNonempty
bindContent	bindContentBidirectional	
unbindContent	unbindContentBidirectional	
size		

Création de différents bindings pour les observable listes/sets et maps.
valueAt – ObjectBinding
isEmpty, isNonempty – BooleanBinding
bindContent – ContentBinding entre une liste observable et non-observable (liste aura les mêmes éléments que ObservableList)
binContentBidirectional – binding entre deux observable listes/maps/sets
size – retourne IntegerBinding pour la taille de la liste/set/map

LES MÉTHODES STATIC DE LA CLASS BINDINGS

- String:

concat	convert	format
length	isEmpty	isNonEmpty

- concat – retourne StringExpression (concatenation de plusieurs objets)
- convert – retourne StringExpression pour ObservableValue
- format – retourne StringExpression – formatage de plusieurs objets
- length – retourne IntegerBinding pour ObservableStringValue

LES MÉTHODES STATIC DE LA CLASS BINDINGS

Pour créer les bindings customisés

createBooleanBinding
createDoubleBinding
createFloatBinding
createIntegerBinding
createLongBinding
createStringBinding
createObjectBinding

Autres :
bindBidirectional
unbindBidirectional
when

Les méthodes pour créer un binding customisé. Un paramètre Callable – la fonction qui réalise le binding, d'autres paramètres : observables.

EXEMPLES

```
DoubleProperty a = new SimpleDoubleProperty(3.5);

DoubleProperty b = new SimpleDoubleProperty(4.9);

NumberBinding sum = Bindings.add(a,b);

NumberBinding mx = Bindings.max(a,b);

BooleanBinding meme = Bindings.equal(a,b);

circle.radiusProperty().bind(sum);

//a partir de ce moment la valeur de la propriété
// radius est toujours a+b même quand a et b changent

circle.strokeWidthProperty().bind(mx);

circle.visibleProperty().bind(meme);
```

CREER BINDINGS CUSTOMIZE

Les méthodes create… de la classe Bindings permettent de créer des bindings customisés :

```
static DoubleBinding createDoubleBinding(Callable<Double> func,
                    Observable dependencies …)
```

Les méthodes similaires create pour créer FloatBinding, BooleanBinding, IntegerBinding, StringBinding, LongBinding. Il y a aussi:

```
static <T> ObjectBinding<T> createObjectBinding(Callable<T> funct,
          Observable dependencies …)
```

- Qu'est-ce que c'est Callable<T> ?
  - C'est une fonction qui implémente l'algorithme qui calcule la valeur de binding en fonctions des dépendances.
- les paramètres dependencies indique les Observables utilisés dans le calcul de binding
- Intuitivement si une de dépendances change alors javas fait en sorte que la fonction funct soit recalculée pour que le binding soit mis à jour

CREER BINDINGS CUSTOMIZE

```
interface java.util.concurrent.Callable<T>{
     T call();
}
```

Pour implémenter Callable<T> il suffit implémenter une méthode sans paramètre qui retourne un objet de type T (utilisez lambda pour définir la méthode).

BINDINGS CUSTOMIZE - EXEMPLE

8 CheckBox qui sont censés de donner le 8 bits de poids faible d'un entier affiché dans Text.

```
//construire un tableau de 8 CheckBox
GridPane gridPane = new GridPane();
CheckBox[] checks = new CheckBox[8];
for (int i = 0; i < checks.length; i++) {
    checks[i] = new CheckBox(i + "");
    GridPane.setColumnIndex(checks[i], i);
    GridPane.setRowIndex(checks[i], 0);
    gridPane.getChildren().add(checks[i]);
 }
//l'unique méthode de Callable calcule l'entier à partir de 8
//CheckBox et transforme cet entier en String
Callable<String> callable = () -> {
   int m = 0;
   for (int k = 0; k < checks.length; k++) {
      if (checks[k].isSelected()) {
          m |= 1 << k;
      }
   }
   return m + "";
};
```

BINDINGS CUSTOMIZE - EXEMPLE

```
//créer le tableau de dépendances composé de propriétés selected
// de tous les CheckBox


BooleanProperty[] prop = new BooleanProperty[checks.length];
for(int k=0; k< prop.length; k++){
      prop[k]=checks[k].selectedProperty();
}

//et enfin construire le binding et l'attacher à la propriété
// text de TextField

StringBinding binding = Bindings.createStringBinding(callable,
prop);
Text text = new Text();
text.setFont(Font.font(40));
text.textProperty().bind(binding);
```

Bindings.When()

```
Bindings.when(ObservableBooleanValue cond)
        .then(thenValue)
        .otherwise(elseValue)
```

`thenValue` et `elseValue` doivent être du même type :

`boolean, double, int, float, long, String, ObservableBooleanValue, ObservableNumberValue, ObservableStringValue, ObservableObjectValue<T>.`

La valeur retournée est un `Binding` dont le type dépend de type d'argument de `then()` et `otherwise()`

- `Bindings.when()` retourne l'objet de la classe `When()`
- la classe `When()` possède des méthodes `then()` qui retournent un objet `o`
- l'objet o possède la méthode `otherwise()` qui retourne un `Binding`

Bindings.When().then().otherwise()

```
Border focusedBorder = new Border(new BorderStroke(
                        Color.YELLOWGREEN,
                        BorderStrokeStyle.SOLID,
                        CornerRadii.EMPTY,
                        new BorderWidths(4)));

Border unfocusedBorder = new Border(new BorderStroke(
                        Color.RED,
                        BorderStrokeStyle.SOLID,
                        CornerRadii.EMPTY,
                        new BorderWidths(4)));


ObjectBinding<Border> binding =
          Bindings.when(pane.focusedProperty())
                  .then(focusedBorder)
                  .otherwise(unfocusedBorder);
pane.borderProperty().bind(binding);
// pane aura le bord soit YELLOWGREEN soit RED, en fonction
// de la valuer de la propriété focused de pane
```

bindings - API de chaînage

NumberExpression	
add()	NumberBinding
divide()	NumberBinding
multiply()	NumberBinding
subtract()	NumberBinding
greaterThan()	BooleanBinding
lessThan()	BooleanBinding
isEqualTo()	BooleanBinding
isNotEqualTo()	BooleanBinding
etc.	

Les classes qui implémentent `NumberExpression`:

`DoubleBinding` `DoubleExpression` `DoubleProperty` `SimpleDoubleProperty`, etc. etc.