Représentation d'un JavaBean graphique avec Swing ou JavaFX

Pierre Bettens

pbettens@heb.be

version 1.5
novembre 2015

Depuis JDK8, la librairie Swing est devenue obsolète et est remplacée par JavaFX. Il existe moult tutoriels présentant JavaFX. Cet article ne sera pas le *moult plus unième*.

En très bref, là où Swing s'organise avec un `JFrame` contenant des `JPanel` — auxquels on associe un *layout manager* — eux-mêmes contenant divers composants graphiques tels que des `JButton`, `JTextField`, `JLabel`... tous dans le package `javax.swing.*`, JavaFX se compare à un théâtre.

Le `Stage`, c'est le théâtre où se joue la pièce. La pièce se joue sur la `Scene`. La scène est affublée d'un *layout* — `BorderPane` par exemple — qui est à la fois l'ancien *panel* et son *layout manager*. C'est ce *layout* qui contient les divers composants graphiques tels que des `Button`, `TextField`, `Label`... tous dans les (sous)-packages `javafx.*`.

C'est beaucoup plus romantique — en tout cas imagé — comme présentation.

À l'aide d'un IDE, il est assez simple de planter le décors et d'y déposer ses objets également. Les détails se trouvent dans divers tutoriaux. Ceux qui préfèrent pourront utiliser Scene builder au lieu d'un positionnement « à la main ». Ce qui m'intéresse ici, c'est la gestion des évènements. Est-elle (fort)

différente qu'avec Swing ? D'ailleurs la suite est plutôt destinée à ceux qui veulent voir la différence entre les deux approches... ils pourront ensuite oublier l'approche « Swing » !

Le principe est identique, c'est le *design pattern* observateur / observé. Certains composants ont la capacité d'être observés, d'autres celle d'observer. Les observateurs doivent s'inscrire auprès de l'observé afin d'être ajouté à la liste des observateurs et, ainsi, être notifiés des changements.

La phrase précédente fonctionne très bien également avec le vocable écouteurs / écoutés.

Intéressons nous à la représentation graphique d'une LED électronique. *Ça s'allume et ça s'éteint. C'est généralement rouge !* Cette led — je l'écrirai toujours en minuscule même si c'est un acronyme pour *light emitting diode* — possède une couleur et un état allumé (*on*) ou éteint (*off*). Sa couleur et son état on/off sont deux propriétés que le *bean* présente. La led sera représentée par un disque rouge lorsque elle est allumée et par un disque blanc lorsqu'elle est éteinte. Sa couleur rouge par défaut pourra être changée. Plus tard, ce composant graphique pourra être cliqué et changé d'état on/off à chaque click de la souris.

Tous les codes présentés sont disponibles sur github.

Led

Un simple bean

Comment créer un JavaBean ? Un javabean n'est pas nécessairement un composant graphique. C'est une classe Java ayant certaines caractéristiques. La principale étant que cet « objet » expose des propriétés.

Pour être reconnu comme étant un *bean*, il faut:

- être `Serializable`;
- avoir un constructeur sans paramètre;
- présenter correctement ses propriétés. Une propriété a un accesseur et un mutateur. Dès lors que cette propriété change, le *bean* en informe tous ses écouteurs.

Avec Swing, il fallait hériter de `JPanel` et définir ses propriétés en utilisant des chaines. Un peu comme ça.

Je retire sciemment la javadoc, les imports et l'instruction package des codes exemples par soucis de concision. Pour plus de détails, voyez github comme je le propose plus haut.

```
public class GLed extends JPanel
        implements Serializable {

    public static final String PROPERTY_ON = "my.package, property on";
    public static final String PROPERTY_COLOR = "my.package, property color";

    private Color color ;
    private boolean on ;

    public GLed() {
        super();
        this.setPreferredSize(/*set dimension here*/));
```

```
        this.color = Color.RED;
        this.on = false;
    }

    public Color getColor() {
        return color;
    }

    public void setColor(Color newValue) {
        if (newValue.equals(Color.WHITE)) {
            throw new IllegalArgumentException(
                    "Invalid color (you try off color)");
        }
        Color oldValue = this.color;
        this.color = newValue;
        firePropertyChange(PROPERTY_COLOR, oldValue, newValue);
    }

    public boolean isOn() {
        return on;
    }

    public void setOn(boolean newValue) {
        boolean oldValue = this.on ;
        this.on = newValue ;
        this.firePropertyChange(PROPERTY_ON,oldValue,newValue);
    }

}
```

Le *bean* se présente bien. Il a son constructeur sans paramètre qui permettra d'instancier l'objet. À ses propriétés sont associées des accesseurs / mutateurs. Dès lors qu'une propriété change, il en informe ses écouteurs via la méthode `firePropertyChange()` héritée de `JPanel`. Cet héritage lui donne également les méthodes et attributs nécessaires pour gérer ses écouteurs.

S'ajouter comme écouteur du led, se fait avec Swing, comme suit:

```
led.addPropertyChangeListener(this);
```

Pour être écouteur, je dois être `PropertyChangeListener` et implémenter la méthode `propertyChange`. Par exemple:

```
@Override
    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(GLed.PROPERTY_ON)) {
            System.out.println("La led m'informe" + evt.getNewValue());
        }
    }
```

Il est bien sûr également possible d'ajouter un écouteur en utilisant une classe interne anonyme ou nommée. Par exemple pour une classe interne anonyme:

```
led.addPropertyChangeListener(new PropertyChangeListener() {

    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(GLed.PROPERTY_ON)) {
            System.out.println("La led m'informe" + evt.getNewValue());
        }
    }

});
```

La philosophie de JavaFX est un peu différente. En effet, JavaFX ajoute la notion de « propriété » *via* des classes `Property`. Comme d'habitude les classes existent pour les valeurs primitives et une classe *generics* existe pour tous les autres objets.

Pour définir une led ayant comme propriété le booléen *on*, il suffira d'écrire avec JavaFX:

```
public class LedOn implements Serializable {

    protected BooleanProperty on = new SimpleBooleanProperty(true);

    public final void setOn(boolean aon){
        on.set(aon);
    }

    public final boolean isOn(){
        return on.get();
    }

    public final BooleanProperty onProperty(){
        return on;
    }
}
```

Déroutant n'est-ce pas ? Il est également nécessaire de s'ajouter comme écouteur auprès de la led. La méthode `onProperty()` offre la gestion des écouteurs tandis que la méthode `set()` se chargera du *fire* dès que c'est nécessaire. Il n'est plus utile de s'en charger.

Pour pouvoir s'ajouter comme écouteur, on peut être « `ChangeListener<Boolean>` » et implémenter la méthode `changed()` ou simplemen utiliser une classe interne anonyme comme suit:

```
led.onProperty().addListener(new ChangeListener<Boolean>() {

            @Override
            public void changed(ObservableValue<? extends Boolean> observable,
                    Boolean oldValue, Boolean newValue) {
                // I've changed. Promise
                System.out.println("Led change de "
                    + oldValue + " vers " + newValue);


            }
        });
```

Pour une valeur de type `Color` par exemple — qui n'est donc pas un type primitif — c'est un peu plus complexe... quoique avec un IDE, le générateur de code aide bien. En utilisant une classe interne anonyme (*inner class*), l'ajout d'une propriété peut s'écrire:

```
//...

protected ObjectProperty<Color> color =
       new ObjectPropertyBase<Color>(Color.RED) {

    @Override
    public Object getBean() {
         return this;
    }

    @Override
    public String getName() {
        return "Color property";
    }
};

//...

public final Color getColor() {
    return color.get();
}

public final void setColor(Color c){
    color.set(c);
}

public final ObjectProperty<Color> colorProperty(){
    return color;
}
```

L'ajout des *getter / setter* ainsi que la méthode `colorProperty()` se fait comme pour la propriété *on*. On a maintenant un *bean* fonctionnel et exposant deux propriétés.

Quand mon bean devient graphique

Pour être un composant graphique avec Swing, le *bean* peut hériter de la classe `javax.swing.JPanel`. En héritant de cette classe:

- j'hérite également de `javax.swing.Container` qui offre la capacité d'être écouté. Je pourrai donc faire un `firePropertyChange()` dès qu'une propriété change. C'est bien le développeur qui est responsable de ce *fire*;
- je suis un composant **graphique** et il me suffit de réécrire la méthode `paintComponent(Graphics)` pour que mon composant ait l'allure qui me convient. Un appel à `repaint()` quand une propriété (graphique) change permet la mise à jour de l'aspect graphique du composant.

J'ajoute cette réécriture de la méthode `paintComponent()` dans mon code ainsi qu'un appel à la méthode `repaint()` dans chaque *setter*.

```java
public class GLed extends JPanel
        implements Serializable {

    //...

    public void setOn(boolean b) {
        //...
        repaint();
    }

    public void setColor(Color c){
        //...
        repaint();
    }

    @Override
    protected void paintComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(
               RenderingHints.KEY_RENDERING,
               RenderingHints.VALUE_RENDER_QUALITY);
        g2.setRenderingHint(
               RenderingHints.KEY_ALPHA_INTERPOLATION,
               RenderingHints.VALUE_ALPHA_INTERPOLATION_QUALITY);
        g2.setRenderingHint(
               RenderingHints.KEY_ANTIALIASING,
               RenderingHints.VALUE_ANTIALIAS_ON);
        g2.setColor(Color.BLACK) ;
        g2.drawOval(/*...*/);
        if ( isOn() ) {
            g2.setColor(color) ;
        } else {
            g2.setColor(Color.WHITE) ;
        }
        g2.fillOval(/*...*/) ;
    }
```

Pour être un composant graphique avec JavaFX, le bean peut hériter de la classe `javafx.scene.Parent`. Pas de méthode `paintComponent()` appelée automatiquement ou *via* `repaint()`. Il suffit d'ajouter des composants graphiques comme attributs privés — par exemple un `Circle` — et de les mettre à jour dans les *setters*. Le reste se fait automatiquement. L'ajout de la figure dans le composant se fait en l'ajoutant comme nœud enfant du composant (voir par exemple ce tutoriel pour une explication sur l'organisation en nœuds).

Notez au passage l'apparition d'une couleur *transparente*.

```java
// ...
private Circle circle;

public GLed() {
    circle = new Circle(50);    // It's better to use a constant
    circle.setFill(color.get());
```

```
    circle.setStroke(Color.BLACK);
    getChildren().add(circle);
}

public final void setOn(boolean aon){
    on.set(aon);
    if (aon) {
        circle.setFill(color.get());
    } else {
        circle.setFill(Color.WHITE);
    }
}

public final void setColor(Color c){
    color.set(c);
    circle.setFill(c);
}
```

Puisqu'avec Swing, je suis dans une optique où je précise quel type de *listeners* j'utilise — en l'occurrence, des *PropertyChangeListener* — la classe qui veut écouter une led doit implémenter `PropertyChangeListener` et écrire la méthode `propertyChange()` **ou** bien utiliser les classes internes anonymes. Par exemple,

```
led.addPropertyChangeListener(new PropertyChangeListener() {

    public void propertyChange(PropertyChangeEvent evt) {
        if (evt.getPropertyName().equals(GLed.PROPERTY_ON)
                && !(Boolean) evt.getNewValue()) {
            // do something
        }
    }
});
```

Avec JavaFX, le principe est semblable bien que plus général. Pour attraper un événement (*event*) — tous les événements sont des `Event` — il faut implémenter `ChangeListener<T>` et écrire la méthode `changed` plus générique.

```
led.onProperty().addListener(new ChangeListener<Boolean>() {

    @Override
    public void changed(ObservableValue<? extends Boolean> observable,
            Boolean oldValue, Boolean newValue) {
        // do something
    }
});
```

Dans cet exemple, le seul changement graphique lorsque la led s'allume, s'éteint ou change de couleur, c'est la couleur de remplissage du cercle. Je dois pourtant me charger de mettre à jour cette couleur dans chaque *setter* par le biais de `circle.setFill(<color>)`. On me souffle dans l'oreillette — merci — qu'il est possible de lier des propriétés entre elles. *bind*.

```
circle.fillProperty().bind(color);
```

Avec ceci, dès que je change la propriété *color* via son *setter* par exemple, la propriété *fill* de *circle* est également mise à jour. Et le cercle change de couleur.

Ça ne suffira malheureusement pas dans notre cas. Pour que le cercle change de couleur lorsque *color* change ou lorsque *on* change... c'est un peu plus complexe. Il faut que les propriétés *color* et *on* soient liées dans un *binding* spécifique qu'il faudra définir. Il hérite de `ObjectBinding<T>` où `T` sera de type `Color` dans notre cas puisque c'est une couleur qu'il faut fournir à la propriété *fill*.

Avec une classe interne comme celle-ci

```
private class FillShapeBinding extends ObjectBinding<Color> {

        public FillShapeBinding() {
            bind(color, on);
        }

        @Override
        protected Color computeValue() {
            return isOn() ? getColor() : Color.TRANSPARENT;
        }

    }
}
```

on pourra ajouter le *binding* à la propriété *fill* et les trois propriétés seront liées.

```
circle.fillProperty().bind(new FillShapeBinding());
```

Les *setters* sont maintenant de simples setters dans lesquels n'apparaissent plus aucune référence à la classe *circle*.

Et la gestion des évènements ?

Je voudrais maintenant rendre ma led cliquable. Lorsque l'on clique dessus, elle passe de l'état éteint (*off*) à l'état allumé (*on*) et *vice versa*. Mon *JavaBean* doit donc écouter la souris. Il doit s'ajouter comme écouteur des clicks (ou autres événements de la souris).

Avec Swing, la led doit alors implémenter `MouseListener` et écrire toutes les méthodes de l'interface à savoir les cinq méthodes `MouseClicked`, `MousePressed`, `MouseRelease`, `MouseEntered` et `MouseExited`. Même si certaines méthodes peuvent avoir un corps vide, elles doivent être présentes. Je ne peut pas hériter de `MouseAdapter` et ne réécrire que les méthodes qui m'intéressent car j'ai déjà mangé mon héritage. Je pourrais utiliser une classe interne anonyme dans mon constructeur mais dans ce cas je ne serais pas « reconnu publiquement » comme étant écouteur de cette souris.

Bref, j'ajoute à mon code les instructions suivantes:

```
public class GLed extends JPanel
        implements Serializable, PropertyChangeListener, MouseListener {
```

```
    // ...

    public GLed() {
        // ...
        addMouseListener(this);
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        setOn(!isOn());
    }

    @Override
    public void mousePressed(MouseEvent e) {
        // nothing to do
    }

    // some " no code" for mouseReleased(MouseEvent e),
    // mouseEntered(MouseEvent e) and mouseExited(MouseEvent e)
}
```

Je peux toujours préférer la manière « classes internes ». De cette manière, je peux utiliser la classe `MouseAdapter`. Ce qui me donne un code plus concis:

```
public class GLed extends JPanel
        implements Serializable {

    // ...

    public GLed() {
        // ...
        addMouseListener(new MouseAdapter() {

            @Override
            public void mouseClicked(MouseEvent e) {
                setOn(!isOn());
            }

        });

    }
}
```

À nouveau, JavaFX est plus générique et la led doit implémenter `EventHandler<T>` ou `T` sera ici `MouseEvent` ou utiliser une classe interne anonyme. Lorsque l'on s'ajoute comme écouteur, il faut préciser quel évènement précis on veut écouter. Dans cet exemple, c'est `MouseEvent.MOUSE_CLICKED`.

```
public class GLed extends Parent {

        // ...
```

```
    public GLed() {
        // ...
        addEventHandler(MouseEvent.MOUSE_CLICKED, new EventHandler<MouseEvent>() {

            @Override
            public void handle(MouseEvent event) {
                setOn(!isOn());
            }
        });
    }
}
```

Plus générique et plus concis !

Et si j'utilise les lambdas — ce que je n'ai sciemment pas fait ici pour ne pas embrouiller le lecteur — je peux encore faire plus court. Il faudra trouver le bon équilibre entre concision et lisibilité mais ceci est une autre histoire.

```
public class GLed extends Parent {

        // ...

    public GLed() {
        // ...
        addEventHandler(MouseEvent.MOUSE_CLICKED, (MouseEvent event) -> {
            setOn(!isOn());
        });
    }
}
```

Vous pouvez reprendre une activité normale.

Remarques. En octobre 2014, JavaFX sans OpenGl ne fonctionnait. Il n'était pas possible de lancer une interface graphique. Pour résoudre le problème, ajouter une option à la jvm: `-Dprism.order=sw`*. Ça ne semble plus très utile en octobre 2015. Pour ceux qui cherchent un tutoriel classique, openclassrooms en propose un.*

Crédit photo par Peter Corbett; des leds d'une lampe de vélo. Conseil sécurité. En vélo, si vous êtes vu, c'est principalement grâce à votre ***vareuse****. Pas parce que vous allumez trois malheureux leds.*