

Programmation visuelle et Beans

Jusqu'ici, dans ce livre, nous avons vu que Java permet de créer des morceaux de code réutilisables. L'unité de code la plus réutilisable est la classe, car elle contient un ensemble cohérent de caractéristiques (champs) et comportements (méthodes) qui peuvent être réutilisées soit directement par combinaison, soit par héritage.

L'héritage et le polymorphisme sont des éléments essentiels de la programmation orientée objet, mais dans la majorité des cas, lorsqu'on bâtit une application, en fait on désire disposer de composants qui font exactement ce qu'on veut. On aimerait placer ces éléments dans notre conception comme l'ingénieur électronicien qui assemble des puces sur un circuit. On sent bien qu'il devrait y avoir une façon d'accélérer ce style de programmation modulaire.

La programmation visuelle est devenue très populaire d'abord avec le Visual Basic (VB) de Microsoft, ensuite avec une seconde génération d'outils, avec Delphi de Borland (l'inspiration principale de la conception des JavaBeans). Avec ces outils de programmation, les composants sont représentés visuellement, ce qui est logique car ils affichent d'habitude un composant visuel tel qu'un bouton ou un champ de texte. La représentation visuelle est en fait souvent exactement l'aspect du composant lorsque le programme tournera. Une partie du processus de programmation visuelle consiste à faire glisser un composant d'une palette pour le déposer dans un formulaire. Pendant qu'on fait cette opération, l'outil de construction d'applications génère du code, et ce code entraînera la création du composant lors de l'exécution du programme.

Le simple fait de déposer des composants dans un formulaire ne suffit généralement pas à compléter le programme. Il faut souvent modifier les caractéristiques d'un composant, telles que sa couleur, son texte, à quelle base de données il est connecté, et cetera. Des caractéristiques pouvant être modifiées au moment de la conception s'appellent des *propriétés* [*properties*]. On peut manipuler les propriétés du composant dans l'outil de construction d'applications, et ces données de configuration sont sauvegardées lors de la construction du programme, de sorte qu'elles puissent être régénérées lors de son exécution.

Vous êtes probablement maintenant habitués à l'idée qu'un objet est plus que des caractéristiques ; c'est aussi un ensemble de comportements. À la conception, les comportements d'un composant visuel sont partiellement représentés par des *événements* [*events*], signifiant : «ceci peut arriver à ce composant». En général on décide de ce qui se passera lorsqu'un événement apparaît en liant du code à cet événement.

C'est ici que se trouve le point critique du sujet : l'outil de construction d'applications utilise la réflexion pour interroger dynamiquement le composant et découvrir quelles propriétés et événements le composant accepte. Une fois connues, il peut afficher ces propriétés et en permettre la modification (tout en sauvegardant l'état lors de la construction du programme), et afficher également les événements. En général, on double-clique sur un événement et l'outil crée la structure du code relié à cet événement. Tout ce qu'il reste à faire est d'écrire le code qui s'exécute lorsque cet événement arrive.

Tout ceci fait qu'une bonne partie du travail est faite par l'outil de construction d'applications. On peut alors se concentrer sur l'aspect du programme et ce qu'il est supposé faire, et s'appuyer sur l'outil pour s'occuper du détail des connexions. La raison pour laquelle les outils de programmation visuels ont autant de succès est qu'ils accélèrent fortement le processus de construction d'une application, l'interface utilisateur à coup sûr, mais également d'autres parties de l'application.

Qu'est-ce qu'un Bean ?

Une fois la poussière retombée, un composant est uniquement un bloc de code, normalement intégré dans une

classe. La clé du système est la capacité du constructeur d'applications de découvrir les propriétés et événements de ce composant. Pour créer un composant VB, le programmeur devait écrire un bout de code assez compliqué, en suivant certaines conventions pour exposer les propriétés et événements. Delphi est un outil de programmation visuelle de seconde génération, pour lequel il est beaucoup plus facile de créer un composant visuel. Java de son côté a porté la création de composants visuels à son état le plus avancé, avec les JavaBeans, car un Bean est tout simplement une classe. Il n'y a pas besoin d'écrire de code supplémentaire ou d'utiliser des extensions particulières du langage pour transformer quelque chose en Bean. La seule chose à faire, en fait, est de modifier légèrement la façon de nommer les méthodes. C'est le nom de la méthode qui dit au constructeur d'applications s'il s'agit d'une propriété, d'un événement, ou simplement une méthode ordinaire.

Dans la documentation Java, cette convention de nommage est par erreur désignée comme un modèle de conception [*design pattern*]. Ceci est maladroit, car les modèles de conception (voir *Thinking in Patterns with Java*, téléchargeable à www.BruceEckel.com) sont suffisamment difficiles à comprendre sans ajouter ce genre de confusions. Ce n'est pas un modèle de conception, c'est uniquement une convention de nommage assez simple :

1. Pour une propriété nommée **xxx**, on crée deux méthodes : **getXxx()** et **setXxx()**. Remarquons que la première lettre après get ou set est transformée automatiquement en minuscule pour obtenir le nom de la propriété. Le type fourni par la méthode get est le même que le type de l'argument de la méthode set. Le nom de la propriété et le type pour les méthodes get et set ne sont pas liés.
2. Pour une propriété de type boolean, on peut utiliser les méthodes get et set comme ci-dessus, ou utiliser is au lieu de get.
3. Les méthodes ordinaires du Bean ne suivent pas la convention de nommage ci-dessus, mais elles sont **public**.
4. Pour les événements, on utilise la technique Swing du *listener*. C'est exactement la même chose que ce qu'on a déjà vu : **addFooBarListener(FooBarListener)** et **removeFooBarListener(FooBarListener)** pour gérer un **FooBarEvent**. La plupart du temps, les événements intégrés satisfont les besoins, mais on peut créer ses propres événements et interfaces *listeners*.

Le point 1 ci-dessus répond à la question que vous vous êtes peut-être posée en comparant un ancien et un nouveau code : un certain nombre de méthodes ont subi de petits changements de noms, apparemment sans raison. On voit maintenant que la plupart de ces changements avaient pour but de s'adapter aux conventions de nommage get et set de manière à transformer les composants en Beans.

On peut utiliser ces règles pour créer un Bean simple :

```
//: frogbean:Frog.java

// Un JavaBean trivial.

package frogbean;

import java.awt.*;

import java.awt.event.*;

class Spots {}

public class Frog {

    private int jumps;

    private Color color;
```

```

private Spots spots;

private boolean jmpr;

public int getJumps() { return jumps; }

public void setJumps(int newJumps) {

    jumps = newJumps;

}

public Color getColor() { return color; }

public void setColor(Color newColor) {

    color = newColor;

}

public Spots getSpots() { return spots; }

public void setSpots(Spots newSpots) {

    spots = newSpots;

}

public boolean isJumper() { return jmpr; }

public void setJumper(boolean j) { jmpr = j; }

public void addActionListener(

    ActionListener l) {

    //...

}

public void removeActionListener(

    ActionListener l) {

    // ...

}

public void addKeyListener(KeyListener l) {

    // ...

}

public void removeKeyListener(KeyListener l) {

    // ...

}

// Une méthode public "ordinaire" :

public void croak() {

    System.out.println("Ribbet!");

}

```

```
} ///:~
```

Tout d'abord, on voit qu'il s'agit d'une simple classe. En général, tous les champs seront **private**, et accessibles uniquement à l'aide des méthodes. En suivant la convention de nommage, les propriétés sont **jumps**, **color**, **spots** et **jumper** (remarquons le passage à la minuscule pour la première lettre du nom de la propriété). Bien que le nom de l'identificateur interne soit le même que le nom de la propriété dans les trois premiers cas, dans **jumper** on peut voir que le nom de la propriété n'oblige pas à utiliser un identificateur particulier pour les variables internes (ou même, en fait, d'*avoir* des variable internes pour cette propriété).

Les événements gérés par ce Bean sont **ActionEvent** et **KeyEvent**, basés sur le nom des méthodes `add` et `remove` pour le *listener* associé. Enfin on remarquera que la méthode ordinaire **croak()** fait toujours partie du Bean simplement parce qu'il s'agit d'une méthode **public**, et non parce qu'elle se conforme à une quelconque convention de nommage.

Extraction des informations sur les Beans *[BeanInfo]* à l'aide de l'introspecteur *[Introspector]*

L'un des points critiques du système des Beans est le moment où on fait glisser un bean d'une palette pour le déposer dans un formulaire. L'outil de construction d'applications doit être capable de créer le Bean (il y arrive s'il existe un constructeur par défaut) et, sans accéder au code source du Bean, extraire toutes les informations nécessaires à la création de la feuille de propriétés et de traitement d'événements.

Une partie de la solution est déjà évidente depuis la fin du Chapitre 12 : la *réflexion* Java permet de découvrir toutes les méthodes d'une classe anonyme. Ceci est parfait pour résoudre le problème des Beans, sans avoir accès à des mots clés du langage spéciaux, comme ceux utilisés dans d'autres langages de programmation visuelle. En fait, une des raisons principales d'inclure la réflexion dans Java était de permettre les Beans (bien que la réflexion serve aussi à la sérialisation des objets et à l'invocation de méthodes à distance [*RMI : remote method invocation*]). On pourrait donc s'attendre à ce qu'un outil de construction d'applications doive appliquer la réflexion à chaque Bean et à fureter dans ses méthodes pour trouver les propriétés et événements de ce Bean.

Ceci serait certainement possible, mais les concepteurs du langage Java voulaient fournir un outil standard, non seulement pour rendre les Beans plus faciles à utiliser, mais aussi pour fournir une plate-forme standard pour la création de Beans plus complexes. Cet outil est la classe **Introspector**, la méthode la plus importante de cette classe est le **static getBeanInfo()**. On passe la référence d'une **Class** à cette méthode, elle l'interroge complètement et retourne un objet **BeanInfo** qu'on peut disséquer pour trouver les propriétés, méthodes et événements.

Vous n'aurez probablement pas à vous préoccuper de tout ceci, vous utiliserez probablement la plupart du temps des beans prêts à l'emploi, et vous n'aurez pas besoin de connaître toute la magie qui se cache là-dessous. Vous ferez simplement glisser vos Beans dans des formulaires, vous en configurerez les propriétés et vous écrirez des traitements pour les événements qui vous intéressent. Toutefois, c'est un exercice intéressant et pédagogique d'utiliser l'**Introspector** pour afficher les informations sur un Bean, et voici donc un outil qui le fait :

```
///: cl3:BeanDumper.java
// Introspection d'un Bean.
// <applet code=BeanDumper width=600 height=500>
// </applet>

import java.beans.*;
import java.lang.reflect.*;
```

```

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import com.bruceeckel.swing.*;

public class BeanDumper extends JApplet {

    JTextField query =

        new JTextField(20);

    JTextArea results = new JTextArea();

    public void prt(String s) {

        results.append(s + "\n");

    }

    public void dump(Class bean){

        results.setText("");

        BeanInfo bi = null;

        try {

            bi = Introspector.getBeanInfo(

                bean, java.lang.Object.class);

        } catch(IntrospectionException e) {

            prt("Couldn't introspect " +

                bean.getName());

            return;

        }

        PropertyDescriptor[] properties =

            bi.getPropertyDescriptors();

        for(int i = 0; i < properties.length; i++) {

            Class p = properties[i].getPropertyType();

            prt("Property type:\n  " + p.getName() +

                "Property name:\n  " +

                properties[i].getName());

            Method readMethod =

                properties[i].getReadMethod();

            if(readMethod != null)

                prt("Read method:\n  " + readMethod);

```

```

Method writeMethod =
    properties[i].getWriteMethod();
    if(writeMethod != null)
        prt("Write method:\n  " + writeMethod);
    prt("=====");
}
prt("Public methods:");
MethodDescriptor[] methods =
    bi.getMethodDescriptors();
for(int i = 0; i < methods.length; i++)
    prt(methods[i].getMethod().toString());
prt("=====");
prt("Event support:");
EventSetDescriptor[] events =
    bi.getEventSetDescriptors();
for(int i = 0; i < events.length; i++) {
    prt("Listener type:\n  " +
        events[i].getListenerType().getName());
    Method[] lm =
        events[i].getListenerMethods();
    for(int j = 0; j < lm.length; j++)
        prt("Listener method:\n  " +
            lm[j].getName());
    MethodDescriptor[] lmd =
        events[i].getListenerMethodDescriptors();
    for(int j = 0; j < lmd.length; j++)
        prt("Method descriptor:\n  " +
            lmd[j].getMethod());
    Method addListener =
        events[i].getAddListenerMethod();
    prt("Add Listener Method:\n  " +
        addListener);
    Method removeListener =
        events[i].getRemoveListenerMethod();

```

```

        prt("Remove Listener Method:\n  " +
            removeListener);
        prt("=====");
    }
}

class Dumper implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String name = query.getText();

        Class c = null;

        try {
            c = Class.forName(name);
        } catch(ClassNotFoundException ex) {
            results.setText("Couldn't find " + name);
            return;
        }

        dump(c);
    }
}

public void init() {
    Container cp = getContentPane();
    JPanel p = new JPanel();
    p.setLayout(new FlowLayout());
    p.add(new JLabel("Qualified bean name:"));
    p.add(query);
    cp.add(BorderLayout.NORTH, p);
    cp.add(new JScrollPane(results));
    Dumper dmpr = new Dumper();
    query.addActionListener(dmpr);
    query.setText("frogbean.Frog");
    // Force evaluation
    dmpr.actionPerformed(
        new ActionEvent(dmpr, 0, ""));
}

public static void main(String[] args) {

```

```

        Console.run(new BeanDumper(), 600, 500);
    }
} ///:~

```

BeanDumper.dump() est la méthode qui fait tout le travail. Il essaie d'abord de créer un objet **BeanInfo**, et en cas de succès il appelle les méthodes de **BeanInfo** qui fournissent les informations sur les propriétés, méthodes et événements. Dans **Introspector.getBeanInfo()**, on voit qu'il y a un second argument. Celui-ci dit à l'**Introspector** où s'arrêter dans la hiérarchie d'héritage. Ici, il s'arrête avant d'analyser toutes les méthodes d'**Object**, parce qu'elles ne nous intéressent pas.

Pour les propriétés, **getPropertyDescriptors()** renvoie un tableau de **PropertyDescriptors**. Pour chaque **PropertyDescriptor**, on peut appeler **getPropertyType()** pour connaître la classe d'un objet passé par les méthodes de propriétés. Ensuite, on peut obtenir le nom de chaque propriété (issu du nom des méthodes) à l'aide de **getName()**, la méthode pour la lire à l'aide de **getReadMethod()**, et la méthode pour la modifier à l'aide de **getWriteMethod()**. Ces deux dernières méthodes retournent un objet **Method** qui peut être utilisé pour appeler la méthode correspondante de l'objet (ceci fait partie de la réflexion).

Pour les méthodes **public** (y compris les méthodes des propriétés), **getMethodDescriptors()** renvoie un tableau de **MethodDescriptors**. Pour chacun de ces descripteurs, on peut obtenir l'objet **Method** associé, et imprimer son nom.

Pour les événements, **getEventSetDescriptors()** renvoie un tableau de (que pourrait-il renvoyer d'autre ?) **EventSetDescriptors**. Chacun de ces descripteurs peut être utilisé pour obtenir la classe du *listener*, les méthodes de cette classe *listener*, et les méthodes pour ajouter et enlever ce *listener*. Le programme **BeanDumper** imprime toutes ces informations.

Au démarrage, le programme force l'évaluation de **frogbean.Frog**. La sortie, après suppression de détails inutiles ici, est :

```

class name: Frog

Property type:

    Color

Property name:

    color

Read method:

    public Color getColor()

Write method:

    public void setColor(Color)

=====

Property type:

    Spots

Property name:

    spots

```


Read method:

```
public Spots getSpots()
```

Write method:

```
public void setSpots(Spots)
```

=====

Property type:

```
boolean
```

Property name:

```
jumper
```

Read method:

```
public boolean isJumper()
```

Write method:

```
public void setJumper(boolean)
```

=====

Property type:

```
int
```

Property name:

```
jumps
```

Read method:

```
public int getJumps()
```

Write method:

```
public void setJumps(int)
```

=====

Public methods:

```
public void setJumps(int)
```

```
public void croak()
```

```
public void removeActionListener(ActionListener)
```

```
public void addActionListener(ActionListener)
```

```
public int getJumps()
```

```
public void setColor(Color)
```

```
public void setSpots(Spots)
```

```
public void setJumper(boolean)
```

```
public boolean isJumper()
```

```
public void addKeyListener(KeyListener)
```

```

public Color getColor()

public void removeKeyListener(KeyListener)

public Spots getSpots()

=====

Event support:

Listener type:

    KeyListener

Listener method:

    keyTyped

Listener method:

    keyPressed

Listener method:

    keyReleased

Method descriptor:

    public void keyTyped(KeyEvent)

Method descriptor:

    public void keyPressed(KeyEvent)

Method descriptor:

    public void keyReleased(KeyEvent)

Add Listener Method:

    public void addKeyListener(KeyListener)

Remove Listener Method:

    public void removeKeyListener(KeyListener)

=====

Listener type:

    ActionListener

Listener method:

    actionPerformed

Method descriptor:

    public void actionPerformed(ActionEvent)

Add Listener Method:

    public void addActionListener(ActionListener)

Remove Listener Method:

    public void removeActionListener(ActionListener)

```

=====

Ceci révèle la plus grosse partie de ce que voit l'**Introspector** lorsqu'il produit un objet **BeanInfo** à partir d'un Bean. On peut voir que le type de la propriété et son nom sont indépendants. Remarquons la minuscule du nom de la propriété (la seule fois où ceci ne se produit pas est lorsque le nom de la propriété commence par plus d'une majuscule l'une derrière l'autre). Il faut également retenir que les noms de méthodes qu'on voit ici (tels que les noms des méthodes de lecture et d'écriture) sont en fait obtenues à partir d'un objet **Method** qui peut être utilisé pour invoquer les méthodes correspondantes de l'objet.

La liste des méthodes **public** contient les méthodes qui ne sont pas associées à une propriété ou un événement, telles que **croak()**, ainsi que celles qui le sont. Ce sont toutes les méthodes pouvant être appelées par programme pour un Bean, et l'outil de construction d'applications peut décider de les lister toutes lors de la création des appels de méthodes, pour nous faciliter la tâche.

Enfin, on peut voir que les événements sont tous triés entre le *listener*, ses méthodes et les méthodes pour ajouter et supprimer les *listeners*. Fondamentalement, une fois obtenu le **BeanInfo**, on peut trouver tout ce qui est important pour un Bean. On peut également appeler les méthodes de ce Bean, bien qu'on n'ait aucune autre information à l'exception de l'objet (ceci est également une caractéristique de la réflexion).

Un Bean plus complexe

L'exemple suivant est un peu plus compliqué, bien que futile. Il s'agit d'un **JPanel** qui dessine un petit cercle autour de la souris chaque fois qu'elle se déplace. Lorsqu'on clique, le mot Bang! apparaît au milieu de l'écran, et un *action listener* est appelé.

Les propriétés modifiables sont la taille du cercle, ainsi que la couleur, la taille et le texte du mot affiché lors du clic. Un **BangBean** a également ses propres **addActionListener()** et **removeActionListener()**, de sorte qu'on peut y attacher son propre listener qui sera appelé lorsque l'utilisateur clique sur le **BangBean**. Vous devriez être capables de reconnaître le support des propriétés et événements :

```
//: bangbean:BangBean.java
// Un Bean graphique.

package bangbean;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import com.bruceeckel.swing.*;

public class BangBean extends JPanel
    implements Serializable {
    protected int xm, ym;

    protected int cSize = 20; // Taille du cercle

    protected String text = "Bang!";
```

```

protected int fontSize = 48;

protected Color tColor = Color.red;

protected ActionListener actionListener;

public BangBean() {
    addMouseListener(new ML());
    addMouseMotionListener(new MML());
}

public int getCircleSize() { return cSize; }

public void setCircleSize(int newSize) {
    cSize = newSize;
}

public String getBangText() { return text; }

public void setBangText(String newText) {
    text = newText;
}

public int getFontSize() { return fontSize; }

public void setFontSize(int newSize) {
    fontSize = newSize;
}

public Color getTextColor() { return tColor; }

public void setTextColor(Color newColor) {
    tColor = newColor;
}

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    g.drawOval(xm - cSize/2, ym - cSize/2,
        cSize, cSize);
}

// Ceci est un "unicast listener", qui est
// la forme la plus simple de gestion des listeners :

public void addActionListener (
    ActionListener l)
    throws TooManyListenersException {

```

```

        if(actionListener != null)

            throw new TooManyListenersException();

        actionListener = l;
    }

    public void removeActionListener(
        ActionListener l) {
        actionListener = null;
    }

    class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            Graphics g = getGraphics();
            g.setColor(tColor);
            g.setFont(
                new Font(
                    "TimesRoman", Font.BOLD, fontSize));
            int width =
                g.getFontMetrics().stringWidth(text);
            g.drawString(text,
                (getSize().width - width) /2,
                getSize().height/2);
            g.dispose();
            // Appel de la méthode du listener :
            if(actionListener != null)
                actionListener.actionPerformed(
                    new ActionEvent(BangBean.this,
                        ActionEvent.ACTION_PERFORMED, null));
        }
    }

    class MML extends MouseMotionAdapter {
        public void mouseMoved(MouseEvent e) {
            xm = e.getX();
            ym = e.getY();
            repaint();
        }
    }

```

```

    }

    public Dimension getPreferredSize() {

        return new Dimension(200, 200);

    }

} ///:~

```

La première chose qu'on remarquera est que **BangBean** implémente l'interface **Serializable**. Ceci signifie que l'outil de construction d'applications peut conserver toutes les informations sur le **BangBean** en utilisant la sérialisation, lorsque les valeurs des propriétés ont été ajustées par l'utilisateur. Lors de la création du Bean au moment de l'exécution du programme, ces propriétés sauvegardées sont restaurées de manière à obtenir exactement ce qu'elles valaient lors de la conception.

On peut voir que tous les champs sont **private**, ce qu'on fait en général avec un Bean : autoriser l'accès uniquement à travers des méthodes, normalement en utilisant le système de propriétés.

En observant la signature de **addActionListener()**, on voit qu'il peut émettre une **TooManyListenersException**. Ceci indique qu'il est *unicast*, ce qui signifie qu'il signale à un seul *listener* l'arrivée d'un événement. En général on utilise des événements *multicast*, de sorte que de nombreux *listeners* puissent être notifiés de l'arrivée d'un événement. Cependant on entre ici dans des problèmes que vous ne pouvez pas comprendre avant le chapitre suivant; on en reparlera donc (sous le titre «*JavaBeans revisited*»). Un événement *unicast* contourne le problème.

Lorsqu'on clique avec la souris, le texte est placé au milieu du **BangBean**, et si le champ de l'**actionListener** n'est pas nul, on appelle son **actionPerformed()**, ce qui crée un nouvel objet **ActionEvent**. Lorsque la souris est déplacée, ses nouvelles coordonnées sont lues et le panneau est redessiné (ce qui efface tout texte sur ce panneau, comme on le remarquera).

Voici la classe **BangBeanTest** permettant de tester le *bean* en tant qu'applet ou en tant qu'application :

```

///: c13:BangBeanTest.java

// <applet code=BangBeanTest
// width=400 height=500></applet>

import bangbean.*;

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import java.util.*;

import com.bruceeckel.swing.*;

public class BangBeanTest extends JApplet {

    JTextField txt = new JTextField(20);

    // Affiche les actions lors des tests :

    class BBL implements ActionListener {

```

```

    int count = 0;

    public void actionPerformed(ActionEvent e){
        txt.setText("BangBean action "+ count++);
    }
}

public void init() {
    BangBean bb = new BangBean();

    try {
        bb.addActionListener(new BBL());
    } catch(TooManyListenersException e) {
        txt.setText("Too many listeners");
    }

    Container cp = getContentPane();
    cp.add(bb);
    cp.add(BorderLayout.SOUTH, txt);
}

public static void main(String[] args) {
    Console.run(new BangBeanTest(), 400, 500);
}
} ///::~

```

Lorsqu'un *Bean* est dans un environnement de développement, cette classe n'est pas utilisée, mais elle est utile pour fournir une méthode de test rapide pour chacun de nos *Beans*. **BangBeanTest** place un **BangBean** dans une applet, attache un simple **ActionListener** au **BangBean** pour afficher un compteur d'événements dans le **TextField** chaque fois qu'un **ActionEvent** arrive. En temps normal, bien sûr, l'outil de construction d'applications créerait la plupart du code d'utilisation du *Bean*.

Lorsqu'on utilise le **BangBean** avec le **BeanDumper**, ou si on place le **BangBean** dans un environnement de développement acceptant les *Beans*, on remarquera qu'il y a beaucoup plus de propriétés et d'actions que ce qui n'apparaît dans le code ci-dessus. Ceci est dû au fait que **BangBean** hérite de **JPanel**, et comme **JPanel** est également un *Bean*, on en voit également ses propriétés et événements.

Emballage d'un Bean

Pour installer un *Bean* dans un outil de développement visuel acceptant les *Beans*, le *Bean* doit être mis dans le conteneur standard des *Beans*, qui est un fichier JAR contenant toutes les classes, ainsi qu'un fichier *manifest* qui dit «ceci est un *Bean*». Un fichier manifest est un simple fichier texte avec un format particulier. Pour le **BangBean**, le fichier manifest ressemble à ceci (sans la première et la dernière ligne) :

```

//::! :BangBean.mf

Manifest-Version: 1.0

```

```
Name: bangbean/BangBean.class
Java-Bean: True
///:~
```

La première ligne indique la version de la structure du fichier manifest, qui est 1.0 jusqu'à nouvel ordre de chez Sun. la deuxième ligne (les lignes vides étant ignorées) nomme le fichier **BangBean.class**, et la troisième précise que c'est un Bean. Sans la troisième ligne, l'outil de construction de programmes ne reconnaîtra pas la classe en tant que Bean.

Le seul point délicat est de s'assurer d'avoir le bon chemin dans le champ «Name:». Si on retourne à **BangBean.java**, on voit qu'il est dans le **package bangbean** (et donc dans un sous-répertoire appelé bangbean qui est en dehors du classpath), et le nom dans le fichier manifest doit contenir cette information de package. De plus, il faut placer le fichier manifest dans le répertoire *au-dessus* de la racine du package, ce qui dans notre cas veut dire le placer dans le répertoire au-dessus du répertoire bangbean. Ensuite il faut lancer **jar** depuis le répertoire où se trouve le fichier manifest, de la façon suivante :

```
jar cfm BangBean.jar BangBean.mf bangbean
```

Ceci suppose qu'on veut que le fichier JAR résultant s'appelle **BangBean.jar**, et qu'on a placé le fichier manifest dans un fichier appelé **BangBean.mf**.

On peut se demander ce qui se passe pour les autres classes générées lorsqu'on a compilé **BangBean.java**. Eh bien, elles ont toutes abouti dans le sous-répertoire **bangbean**. Lorsqu'on donne à la commande **jar** le nom d'un sous-répertoire, il embarque tout le contenu de ce sous-répertoire dans le fichier jar (y compris, dans notre cas, le code source original **BangBean.java** qu'on peut ne pas vouloir inclure dans les Beans). Si on regarde à l'intérieur du fichier JAR, on découvre que le fichier manifest n'y est pas, mais que **jar** a créé son propre fichier manifest (partiellement sur la base du nôtre) appelé **MANIFEST.MF** et l'a placé dans le sous-répertoire **META-INF** (pour meta-information). Si on ouvre ce fichier manifest on remarquera également qu'une information de signature numérique a été ajoutée par **jar** pour chaque fichier, de la forme :

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/O0=
MD5-Digest: 04NcSlhE3Smnzlp2hj6qeg==
```

En général, on ne se préoccupe pas de tout ceci, et lors de modifications il suffit de modifier le fichier manifest d'origine et rappeler **jar** pour créer un nouveau fichier JAR pour le *Bean*. On peut aussi ajouter d'autres *Beans* dans le fichier JAR en ajoutant simplement leurs informations dans le manifest.

Une chose à remarquer est qu'on mettra probablement chaque Bean dans son propre sous-répertoire, puisque lorsqu'on crée un fichier JAR on passe à la commande **jar** le nom d'un sous-répertoire et qu'il place tout ce qui est dans ce répertoire dans le fichier JAR. On peut voir que **Frog** et **BangBean** sont chacun dans leur propre sous-répertoire.

Une fois le Bean convenablement inséré dans un fichier JAR, on peut l'installer dans un environnement de développement de programmes acceptant les Beans. La façon de le faire varie d'un outil à l'autre, mais Sun fournit gratuitement un banc de test pour les JavaBeans dans leur *Beans Development Kit* (BDK) appelé la *beanbox* (le BDK se télécharge à partir de java.sun.com/beans). Pour placer un Bean dans la *beanbox*, il suffit de copier le fichier JAR dans le sous-répertoire jars du BDK avant de lancer la *beanbox*.

Un support des Beans plus sophistiqué

On a vu comme il était simple de fabriquer un Bean. Mais on n'est pas limité à ce qu'on a vu ici. L'architecture des JavaBeans fournit un point d'entrée simple, mais peut aussi s'adapter à des cas plus complexes. Ceux-ci ne sont pas du ressort de ce livre, mais on va les introduire rapidement ici. On trouvera plus de détails à java.sun.com/beans.

Un endroit où l'on peut apporter des perfectionnements est le traitement des propriétés. Les exemples ci-dessus n'ont montré que des propriétés uniques, mais il est également possible de représenter plusieurs propriétés dans un tableau. C'est ce qu'on appelle une *propriété indexée* [*indexed property*]. Il suffit de fournir les méthodes appropriées (également en suivant une convention de nommage pour les noms de méthodes) et l'**Introspector** reconnaît une propriété indexée, de sorte qu'un outil de construction d'applications puisse y répondre correctement.

Les propriétés peuvent être *liées* [*bound*], ce qui signifie qu'elles avertiront les autres objets à l'aide d'un **PropertyChangeEvent**. Les autres objets peuvent alors décider de se modifier eux-mêmes suite à la modification de ce Bean.

Les propriétés peuvent être *contraintes* [*constrained*], ce qui signifie que les autres objets peuvent mettre leur veto sur la modification de cette propriété si c'est inacceptable. Les autres objets sont avertis à l'aide d'un **PropertyChangeEvent**, et ils peuvent émettre un **PropertyVetoException** pour empêcher la modification et pour rétablir les anciennes valeurs.

On peut également modifier la façon de représenter le Bean lors de la conception :

1. On peut fournir une feuille de propriétés spécifique pour un Bean particulier. La feuille de propriétés normale sera utilisée pour tous les autres Beans, mais la feuille spéciale sera automatiquement appelée lorsque ce Bean sera sélectionné.
2. On peut créer un éditeur spécifique pour une propriété particulière, de sorte que la feuille de propriétés normale est utilisée, mais si on veut éditer cette propriété, c'est cet éditeur qui est automatiquement appelé.
3. On peut fournir une classe **BeanInfo** spécifique pour un Bean donné, pour fournir des informations différentes de celles créées par défaut par l'**Introspector**.
4. Il est également possible de valider ou dévalider le mode expert dans tous les **FeatureDescriptors** pour séparer les caractéristiques de base de celles plus compliquées.

Davantage sur les Beans

Il y a un autre problème qui n'a pas été traité ici. Chaque fois qu'on crée un Bean, on doit s'attendre à ce qu'il puisse être exécuté dans un environnement *multithread*. Ceci signifie qu'il faut comprendre les problèmes de *threading*, qui sera présenté au Chapitre 14. On y trouvera un paragraphe appelé «*JavaBeans revisited*» qui parlera de ce problème et de sa solution.

Il y a plusieurs livres sur les JavaBeans, par exemple *JavaBeans* par Elliotte Rusty Harold (IDG, 1998).

Résumé

De toutes les bibliothèques Java, c'est la bibliothèque de GUI qui a subi les changements les plus importants de Java 1.0 à Java 2. L'AWT de Java 1.0 était nettement critiqué comme étant une des moins bonnes conceptions jamais vues, et bien qu'il permette de créer des programmes portables, la GUI résultante était aussi médiocre sur toutes les plateformes. Il était également limité, malaisé et peu agréable à utiliser en comparaison des outils de

développement natifs disponibles sur une plate-forme donnée.

Lorsque Java 1.1 introduisit le nouveau modèle d'événements et les JavaBeans, la scène était installée. Il était désormais possible de créer des composants de GUI pouvant être facilement glissés et déposés à l'intérieur d'outils de développement visuels. De plus, la conception du modèle d'événements et des Beans montre l'accent mis sur la facilité de programmation et la maintenabilité du code (ce qui n'était pas évident avec l'AWT du 1.0). Mais le travail ne s'est terminé qu'avec l'apparition des classes JFC/Swing. Avec les composants Swing, la programmation de GUI toutes plateformes devient une expérience civilisée.

En fait, la seule chose qui manque est l'outil de développement, et c'est là que se trouve la révolution. Visual Basic et Visual C++ de Microsoft nécessitent des outils de développement de Microsoft, et il en est de même pour Delphi et C++ Builder de Borland. Si on désire une amélioration de l'outil, on n'a plus qu'à croiser les doigts et espérer que le fournisseur le fera. Mais java est un environnement ouvert, et de ce fait, non seulement il permet la compétition des outils, mais il l'encourage. Et pour que ces outils soient pris au sérieux, ils doivent permettre l'utilisation des JavaBeans. Ceci signifie un terrain de jeu nivelé : si un meilleur outil de développement apparaît, on n'est pas lié à celui qu'on utilisait jusqu'alors, on peut migrer vers le nouvel outil et augmenter sa productivité. Cet environnement compétitif pour les outils de développement ne s'était jamais vu auparavant, et le marché résultant ne peut que générer des résultats positifs pour la productivité du programmeur.

Ce chapitre était uniquement destiné à vous fournir une introduction à la puissance de Swing et vous faire démarrer, et vous avez pu voir comme il était simple de trouver son chemin à travers les librairies. Ce qu'on a vu jusqu'ici suffira probablement pour une bonne part à vos besoins en développement d'interfaces utilisateurs. Cependant, Swing ne se limite pas qu'à cela. Il est destiné à être une boîte à outils de conception d'interfaces utilisateurs très puissante. Il y a probablement une façon de faire à peu près tout ce qu'on peut imaginer.

Si vous ne trouvez pas ici ce dont vous avez besoin, fouillez dans la documentation en ligne de Sun, et recherchez sur le Web, et si cela ne suffit pas, cherchez un livre consacré à Swing. Un bon endroit pour démarrer est *The JFC Swing Tutorial*, par Walrath & Campione (Addison Wesley, 1999).

Exercices

Les solutions des exercices sélectionnés se trouvent dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une faible somme depuis www.BruceEckel.com.

1. Créer une applet/application utilisant la classe **Console** comme montré dans ce chapitre. Inclure un champ texte et trois boutons. Lorsqu'on appuie sur chaque bouton, faire afficher un texte différent dans le champ texte.
2. Ajouter une boîte à cocher à l'applet de l'exercice 1, capturer l'événement, et insérer un texte différent dans le champ texte.
3. Créer une applet/application utilisant **Console**. Dans la documentation HTML de java.sun.com, trouver le **JPasswordField** et l'ajouter à ce programme. Si l'utilisateur tape le mot de passe correct, utiliser **JOptionPane** pour fournir un message de succès à l'utilisateur.
4. Créer une applet/application utilisant **Console**, et ajouter tous les composants qui ont une méthode **addActionListener()** (rechercher celles-ci dans la documentation HTML de java.sun.com; conseil : utiliser l'index). Capturer ces événements et afficher un message approprié pour chacun dans un champ texte.
5. Créer une applet/application utilisant **Console**, avec un **JButton** et un **JTextField**. Ecrire et attacher le listener approprié de sorte que si le bouton a le focus, les caractères tapés dessus apparaissent dans le **JTextField**.
6. Créer une applet/application utilisant **Console**. Ajouter à la fenêtre principale tous les composants décrits dans ce chapitre, y compris les menus et une boîte de dialogue.

7. Modifier **TextFields.java** de sorte que les caractères de **t2** gardent la casse qu'ils avaient lorsqu'ils ont été tapés, plutôt que les forcer automatiquement en majuscules.
8. Rechercher et télécharger un ou plusieurs des environnements de développement de GUI disponibles sur Internet, ou acheter un produit du commerce. Découvrir ce qu'il faut faire pour ajouter **Bangbean** à cet environnement et pour l'utiliser.
9. Ajouter **Frog.class** au fichier manifest comme montré dans ce chapitre, et lancer **jar** pour créer un fichier JAR contenant à la fois **Frog** et **BangBean**. Ensuite, télécharger et installer le BDK de Sun ou utiliser un outil de développement admettant les Beans et ajouter le fichier JAR à votre environnement de manière à pouvoir tester les deux Beans.
10. Créer un JavaBean appelé **Valve** qui contient deux propriétés : un **boolean** appelé **on** et un **int** appelé **level**. Créer un fichier manifest, utiliser **jar** pour empaqueter le Bean, puis le charger dans la *beanbox* ou dans un outil de développement acceptant les Beans, de manière à pouvoir le tester.
11. Modifier **MessageBoxes.java** de manière à ce qu'il ait un **ActionListener** individuel pour chaque bouton (au lieu d'un correspondant au texte du bouton).
12. Surveiller un nouveau type d'événement dans **TrackEvent.java** en ajoutant le nouveau code de traitement de l'événement. Il faudra découvrir vous-même le type d'événement que vous voulez surveiller.
13. Créer un nouveau type de bouton hérité de **JButton**. Chaque fois qu'on appuie sur le bouton, celui-ci doit modifier sa couleur selon une couleur choisie de façon aléatoire. Voir **ColorBoxes.java** au Chapitre 14 pour un exemple de la manière de générer aléatoirement une valeur de couleur.
14. Modifier **TextPane.java** de manière à utiliser un **JTextArea** à la place du **JTextPane**.
15. Modifier **Menus.java** pour utiliser des boutons radio au lieu de boîtes à cocher dans les menus.
16. Simplifier **List.java** en passant le tableau au constructeur et en éliminant l'ajout dynamique d'éléments à la liste.
17. Modifier **SineWave.java** pour transformer **SineDraw** en JavaBean en lui ajoutant des méthodes **get** et **set**.
18. Vous vous souvenez du jouet permettant de faire des dessins avec deux boutons, un qui contrôle le mouvement vertical, et un qui contrôle le mouvement horizontal ? En créer un, en utilisant **SineWave.java** comme point de départ. A la place des boutons, utiliser des curseurs. Ajouter un bouton d'effacement de l'ensemble du dessin.
19. Créer un indicateur de progression asymptotique qui ralentit au fur et à mesure qu'il s'approche de la fin. Ajouter un comportement aléatoire de manière à donner l'impression qu'il se remet à accélérer.
20. Modifier **Progress.java** de manière à utiliser un *listener* plutôt que le partage du modèle pour connecter le curseur et la barre de progression.
21. Suivre les instructions du paragraphe «Empaquetage d'une applet dans un fichier JAR» pour placer **TicTacToe.java** dans un fichier JAR. Créer une page HTML avec la version brouillonne et compliquée du tag **applet**, et la modifier pour utiliser le tag **archive** de manière à utiliser le fichier JAR (conseil : commencer par utiliser la page HTML pour **TicTacToe.java** qui est fournie avec le code source pour ce livre).
22. Créer une applet/application utilisant **Console**. Celle-ci doit avoir trois curseurs, un pour le rouge, un pour le vert et un pour le bleu de **java.awt.Color**. Le reste du formulaire sera un **JPanel** qui affiche la couleur fixée par les trois curseurs. Ajouter également des champs textes non modifiables qui indiquent les valeurs courantes des valeurs RGB.
23. Dans la documentation HTML de **javax.swing**, rechercher le **JColorChooser**. Ecrire un programme avec un bouton qui ouvre le sélectionneur de couleur comme un dialogue.
24. Presque tous les composants Swing sont dérivés de **Component**, qui a une méthode **setCursor()**. Rechercher ceci dans la documentation HTML Java . Créer une applet et modifier le curseur selon un des curseurs disponibles dans la classe **Cursor**.
25. En prenant comme base **ShowAddListeners.java**, créer un programme avec toutes les fonctionnalités de **ShowMethodsClean.java** du Chapitre 12.

- [61] Une variante est appelée le principe de l'étonnement minimum, qui dit essentiellement : «ne pas surprendre l'utilisateur».
- [62] Ceci est un exemple du modèle de conception *[design pattern]* appelé la *méthode du modèle [template method]*.
- [63] On suppose que le lecteur connaît les bases du HTML. Il n'est pas très difficile à comprendre, et il y a beaucoup de livres et de ressources disponibles.
- [64] Cette page (en particulier la partie clsid) semblait bien fonctionner avec le JDK1.2.2 et le JDK1.3rc-1. Cependant il se peut que vous ayez parfois besoin de changer le tag dans le futur. Des détails peuvent être trouvés à *java.sun.com*.
- [65] Selon moi. Et après avoir étudié Swing, vous n'aurez plus envie de perdre votre temps sur les parties plus anciennes.
- [66] Comme décrit plus avant, Frame était déjà utilisé par AWT, de sorte que Swing utilise JFrame.
- [67] Ceci s'éclaircira après avoir avancé dans ce chapitre. D'abord faire de la référence **JApplet** un membre static de la classe (au lieu d'une variable locale de **main()**), et ensuite appeler **applet.stop()** et **applet.destroy()** dans **WindowAdapter.windowClosing()** avant d'appeler **System.exit()**.
- [68] Il n'y a pas de **MouseEvent** bien qu'il semble qu'il devrait y en avoir un. Le cliquage et le déplacement sont combinés dans **MouseEvent**, de sorte que cette deuxième apparition de **MouseEvent** dans le tableau n'est pas une erreur.
- [69] En Java 1.0/1.1 on ne pouvait pas hériter de l'objet bouton de façon exploitable. C'était un des nombreux défauts de conception.

[[Chapitre précédent](#)] [[TDM réduite](#)] [[Table des Matières](#)] [[Index](#)] [[Chapitre Suivant](#)]

Last Update of Original Version:04/24/2000

Dernière mise à jour de la version française : 12 novembre 2000