

Traducteur : Philippe BOITE

07.05.2001 - Version 0.2 :

- espaces des guillemets « ajoutés. »;
- colorisation du code (armel)

12.11.2000 - Version 0.1 :

-

## Utilisation de l'Explorateur Windows

Si vous utilisez Windows, vous pouvez simplifier le lancement d'un programme Java en ligne de commande en configurant l'explorateur Windows (le navigateur de fichiers de Windows, *pas* Internet Explorer) de façon à pouvoir double-cliquer sur un fichier **.class** pour l'exécuter. Il y a plusieurs étapes à effectuer.

D'abord, téléchargez et installez le langage de programmation Perl depuis [www.Pperl.org](http://www.Pperl.org). Vous trouverez sur ce site les instructions et la documentation sur ce langage.

Ensuite, créez le script suivant sans la première et la dernière lignes (ce script fait partie du package de sources de ce livre) :

```
//:! c13:RunJava.bat

@rem = '---*-Perl-***

@echo off

perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9

goto endofperl

@rem ' ;

#!perl

$file = $ARGV[0];

$file =~ s/(.*)\..*/\1/;

$file =~ s/(.*\\)*(.*)/$+//;

`java $file`;

__END__

:endofperl

///:~
```

Maintenant, ouvrez l'explorateur Windows, sélectionnez Affichage, Options des dossiers, et cliquez sur l'onglet "Types de fichiers". Cliquez sur le bouton "Nouveau type...". Comme "Description du type", entrez "fichier classe Java". Comme "Extension associée", entrez class. Sous "Actions", cliquez sur le bouton "Nouveau...". Comme "Action" entrez "open", et pour "Application utilisée pour effectuer l'action" entrez une ligne telle que celle-ci :

```
"c:\aaa\Perl\RunJava.bat" "%L"
```

en personnalisant le chemin devant RunJava.bat en fonction de l'endroit où vous avez placé le fichier batch.

Une fois cette installation effectuée, vous pouvez exécuter tout programme Java simplement en double-cliquant sur le fichier **.class** qui contient un **main()**.

## Création d'un bouton

La création d'un bouton est assez simple: il suffit d'appeler le constructeur **JButton** avec le label désiré sur le bouton. On verra plus tard qu'on peut faire des choses encore plus jolies, comme par exemple y mettre des images graphiques.

En général on créera une variable pour le bouton dans la classe courante, afin de pouvoir s'y référer plus tard.

Le **JButton** est un composant possédant sa propre petite fenêtre qui sera automatiquement repeinte lors d'une mise à jour. Ceci signifie qu'on ne peint pas explicitement un bouton ni d'ailleurs les autres types de contrôles; on les place simplement sur le formulaire et on les laisse se repeindre automatiquement. Le placement d'un bouton sur un formulaire se fait dans **init()** :

```
//: c13:Button1.java
// Placement de boutons sur une applet.
// <applet code=Button1 width=200 height=50>
// </applet>

import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Button1 extends JApplet {
    JButton
    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");

    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b1);
        cp.add(b2);
    }

    public static void main(String[] args) {
```

```

    Console.run(new Button1(), 200, 50);
}
} ///::~

```

On a ajouté ici quelque chose de nouveau : avant d'ajouter un quelconque élément sur la "surface de contenu" [*content pane*], on lui attribue un nouveau gestionnaire de disposition [*layout manager*], de type **FlowLayout**. Le *layout manager* définit la façon dont la surface décide implicitement de l'emplacement du contrôle dans le formulaire. Le comportement d'une applet est d'utiliser le **BorderLayout**, mais cela ne marchera pas ici car (comme on l'apprendra plus tard dans ce chapitre lorsqu'on verra avec plus de détails le contrôle de l'organisation d'un formulaire) son comportement par défaut est de couvrir entièrement chaque contrôle par tout nouveau contrôle ajouté. Cependant, **FlowLayout** provoque l'alignement des contrôles uniformément dans le formulaire, de gauche à droite et de haut en bas.

## Capture d'un événement

Vous remarquerez que si vous compilez et exécutez l'applet ci-dessus, rien ne se passe lorsqu'on appuie sur le bouton. C'est à vous de jouer et d'écrire le code qui définira ce qui va se passer. La base de la programmation par événements, qui est très importante dans les interfaces utilisateurs graphiques, est de lier les événements au code qui répond à ces événements.

Ceci est effectué dans Swing par une séparation claire de l'interface (les composants graphiques) et l'implémentation (le code que vous voulez exécuter quand un événement arrive sur un composant). Chaque composant Swing peut répercuter tous les événements qui peuvent lui arriver, et il peut répercuter chaque type d'événement individuellement. Donc si par exemple on n'est pas intéressé par le fait que la souris est déplacée par-dessus le bouton, on n'enregistre pas son intérêt pour cet événement. C'est une façon très directe et élégante de gérer la programmation par événements, et une fois qu'on a compris les concepts de base on peut facilement utiliser les composants Swing qu'on n'a jamais vus auparavant. En fait, ce modèle s'étend à tout ce qui peut être classé comme un **JavaBean** (que nous verrons plus tard dans ce chapitre).

Au début, on s'intéressera uniquement à l'événement le plus important pour les composants utilisés. Dans le cas d'un **JButton**, l'événement intéressant est le fait qu'on appuie sur le bouton. Pour enregistrer son intérêt à l'appui sur un bouton, on appelle la méthode **addActionListener()** de **JButton**. Cette méthode attend un argument qui est un objet qui implémente l'interface **ActionListener**, qui contient une seule méthode appelée **actionPerformed()**. Donc tout ce qu'il faut faire pour attacher du code à un **JButton** est d'implémenter l'interface **ActionListener** dans une classe et d'enregistrer un objet de cette classe avec le **JButton** à l'aide de **addActionListener()**. La méthode sera appelée lorsque le bouton sera enfoncé (ceci est en général appelé une *callback*).

Mais que doit être le résultat de l'appui sur ce bouton ? On aimerait voir quelque chose changer à l'écran; pour cela on va introduire un nouveau composant Swing : le **JTextField**. C'est un endroit où du texte peut être tapé, ou dans notre cas modifié par le programme. Bien qu'il y ait plusieurs façons de créer un **JTextField**, la plus simple est d'indiquer au constructeur uniquement quelle largeur on désire pour ce champ. Une fois le **JTextField** placé sur le formulaire, on peut modifier son contenu en utilisant la méthode **setText()** (il y a beaucoup d'autres méthodes dans **JTextField**, que vous pouvez découvrir dans la documentation HTML pour le JDK depuis [java.sun.com](http://java.sun.com)). Voilà à quoi ça ressemble :

```

///: c13:Button2.java

// Réponse aux appuis sur un bouton.

```

```

// <applet code=Button2 width=200 height=75>
// </applet>

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

public class Button2 extends JApplet {

    JButton

    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");

    JTextField txt = new JTextField(10);

    class BL implements ActionListener {

        public void actionPerformed(ActionEvent e){

            String name =

                ((JButton)e.getSource()).getText();

            txt.setText(name);

        }

    }

    BL al = new BL();

    public void init() {

        b1.addActionListener(al);

        b2.addActionListener(al);

        Container cp = getContentPane();

        cp.setLayout(new FlowLayout());

        cp.add(b1);

        cp.add(b2);

        cp.add(txt);

    }

    public static void main(String[] args) {

        Console.run(new Button2(), 200, 75);

    }

} ///:~

```

La création d'un **JTextField** et son placement sur la surface comprend les mêmes opérations que pour les **JButtons**, comme pour tout composant Swing. La différence dans le programme ci-dessus est la création de la classe **BL** de type **ActionListener** mentionnée précédemment. L'argument de **actionPerformed()** est du type **ActionEvent**, qui contient toute l'information sur l'événement et d'où il vient. Dans ce cas, je voulais décrire le bouton qui a été enfoncé : **getSource()** fournit l'objet d'où l'événement est issu, et j'ai supposé que c'était un **JButton**. **getText()** retourne le texte qui est sur le bouton, qui est placé dans le **JTextField** pour prouver que le code a bien été appelé quand le bouton a été enfoncé.

Dans **init()**, **addActionListener()** est utilisée pour enregistrer l'objet **BL** pour chacun des boutons.

Il est souvent plus pratique de coder l'**ActionListener** comme une classe anonyme interne [*anonymous inner class*], particulièrement lorsqu'on a tendance à n'utiliser qu'une seule instance de chaque classe listener. **Button2.java** peut être modifié de la façon suivante pour utiliser une classe interne anonyme :

```

//: c13:Button2b.java

// Utilisation de classes anonymes internes.
// <applet code=Button2b width=200 height=75>
// </applet>

import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

import com.bruceeckel.swing.*;

public class Button2b extends JApplet {

    JButton

    b1 = new JButton("Button 1"),
    b2 = new JButton("Button 2");

    JTextField txt = new JTextField(10);

    ActionListener al = new ActionListener() {

        public void actionPerformed(ActionEvent e){

            String name =

                ((JButton)e.getSource()).getText();

            txt.setText(name);

        }

    };

    public void init() {

        b1.addActionListener(al);

        b2.addActionListener(al);
    }
}

```

```

    Container cp = getContentPane();

    cp.setLayout(new FlowLayout());

    cp.add(b1);

    cp.add(b2);

    cp.add(txt);
}

public static void main(String[] args) {

    Console.run(new Button2b(), 200, 75);

}

} ///:~

```

L'utilisation d'une classe anonyme interne sera préférée (si possible) pour les exemples de ce livre.

## Zones de texte

Un **JTextArea** est comme un **TextField**, sauf qu'il peut avoir plusieurs lignes et possède plus de fonctionnalités. Une méthode particulièrement utile est **append()**; avec cette méthode on peut facilement transférer une sortie dans un **JTextArea**, faisant de ce fait d'un programme Swing une amélioration (du fait qu'on peut scroller en arrière) par rapport à ce qui a été fait jusqu'ici en utilisant des programmes de ligne de commande qui impriment sur la sortie standard. Comme exemple, le programme suivant remplit un **JTextArea** avec la sortie du générateur **geography** du chapitre 9 :

```

///: c13:TextArea.java

// Utilisation du contrôle JTextArea.

// <applet code=TextArea width=475 height=425>

// </applet>

import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

import java.util.*;

import com.bruceeckel.swing.*;

import com.bruceeckel.util.*;

public class TextArea extends JApplet {

    JButton

    b = new JButton("Add Data"),

    c = new JButton("Clear Data");

    JTextArea t = new JTextArea(20, 40);

```

```

Map m = new HashMap();

public void init() {
    // Utilisation de toutes les données :
    Collections2.fill(m,
        Collections2.geography,
        CountryCapitals.pairs.length);
    b.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            for(Iterator it= m.entrySet().iterator();
                it.hasNext());{
                Map.Entry me = (Map.Entry)(it.next());
                t.append(me.getKey() + ": "
                    + me.getValue() + "\n");
            }
        }
    });
    c.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            t.setText("");
        }
    });
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(new JScrollPane(t));
    cp.add(b);
    cp.add(c);
}

public static void main(String[] args) {
    Console.run(new TextArea(), 475, 425);
}

} ///:~

```

Dans `init()`, le `Map` est rempli avec tous les pays et leurs capitales. Remarquons que pour chaque bouton l'**ActionListener** est créé et ajouté sans définir de variable intermédiaire, puisqu'on n'aura plus jamais besoin de s'y référer dans la suite du programme. Le bouton "Add Data" formate et ajoute à la fin toutes les données,

tandis que le bouton "Clear Data" utilise `setText()` pour supprimer tout le texte du **JTextArea**.

Lors de l'ajout du **JTextArea** à l'applet, il est enveloppé dans un **JScrollPane**, pour contrôler le scrolling quand trop de texte est placé à l'écran. C'est tout ce qu'il y a à faire pour fournir des fonctionnalités de scrolling complètes. Ayant essayé d'imaginer comment faire l'équivalent dans d'autres environnements de programmation de GUI, je suis très impressionné par la simplicité et la bonne conception de composants tels que le **JScrollPane**.

## Contrôle de la disposition

La façon dont on place les composants sur un formulaire en Java est probablement différente de tout système de GUI que vous avez utilisé. Premièrement, tout est dans le code ; il n'y a pas de ressources qui contrôlent le placement des composants. Deuxièmement, la façon dont les composants sont placés dans un formulaire est contrôlée non pas par un positionnement absolu mais par un *layout manager* qui décide comment les composants sont placés, selon l'ordre dans lequel on les ajoute ( `add()` ). La taille, la forme et le placement des composants seront notablement différents d'un *layout manager* à l'autre. De plus, les gestionnaires de disposition s'adaptent aux dimensions de l'applet ou de la fenêtre de l'application, de sorte que si la dimension de la fenêtre est changée, la taille, la forme et le placement des composants sont modifiés en conséquence.

**JApplet**, **JFrame**, **JWindow** et **JDialog** peuvent chacun fournir un **Container** avec `getContentPane()` qui peut contenir et afficher des **Components**. Dans **Container**, il y a une méthode appelée `setLayout()` qui permet de choisir le *layout manager*. D'autres classes telles que **JPanel** contiennent et affichent des composants directement, et donc il faut leur imposer directement le *layout manager*, sans utiliser le *content pane*.

Dans cette section nous allons explorer les divers gestionnaires de disposition en créant des boutons (puisque c'est ce qu'il y a de plus simple). Il n'y aura aucune capture d'événements de boutons puisque ces exemples ont pour seul but de montrer comment les boutons sont disposés.

### BorderLayout

L'applet utilise un *layout manager* par défaut : le **BorderLayout** (certains des exemples précédents ont modifié le *layout manager* par défaut pour **FlowLayout**). Sans autre information, il prend tout ce qu'on lui ajoute ( `add()` ) et le place au centre, en étirant l'objet dans toutes les directions jusqu'aux bords.

Cependant le **BorderLayout** ne se résume pas qu'à cela. Ce *layout manager* possède le concept d'une zone centrale et de quatre régions périphériques. Quand on ajoute quelque chose à un *panel* qui utilise un **BorderLayout**, on peut utiliser la méthode `add()` surchargée qui prend une valeur constante comme premier argument. Cette valeur peut être une des suivantes :

**BorderLayout.NORTH** (en haut)

**BorderLayout.SOUTH** (en bas)

**BorderLayout.EAST** (à droite)

**BorderLayout.WEST** (à gauche)

**BorderLayout.CENTER** (remplir le milieu, jusqu'aux autres composants ou jusqu'aux bords)

Si aucune région n'est spécifiée pour placer l'objet, le défaut est **CENTER**.

Voici un exemple simple. Le *layout* par défaut est utilisé, puisque **JApplet** a **BorderLayout** par défaut :

```
//: c13:BorderLayout1.java
```



```
// Démonstration de BorderLayout.

// <applet code=BorderLayout1
// width=300 height=250> </applet>

import javax.swing.*;

import java.awt.*;

import com.bruceeckel.swing.*;

public class BorderLayout1 extends JApplet {

    public void init() {

        Container cp = getContentPane();

        cp.add(BorderLayout.NORTH,

            new JButton("North"));

        cp.add(BorderLayout.SOUTH,

            new JButton("South"));

        cp.add(BorderLayout.EAST,

            new JButton("East"));

        cp.add(BorderLayout.WEST,

            new JButton("West"));

        cp.add(BorderLayout.CENTER,

            new JButton("Center"));

    }

    public static void main(String[] args) {

        Console.run(new BorderLayout1(), 300, 250);

    }

} ///:~
```

Pour chaque placement autre que **CENTER**, l'élément qu'on ajoute est comprimé pour tenir dans le plus petit espace le long d'une dimension et étiré au maximum le long de l'autre dimension. **CENTER**, par contre, s'étend dans chaque dimension pour occuper le milieu.

## FlowLayout

Celui-ci aligne simplement les composants sur le formulaire, de gauche à droite jusqu'à ce que l'espace du haut soit rempli, puis descend d'une rangée et continue l'alignement.

Voici un exemple qui positionne le *layout manager* en **FlowLayout** et place ensuite des boutons sur le formulaire. On remarquera qu'avec **FlowLayout** les composants prennent leur taille naturelle. Un **JButton**, par exemple, aura la taille de sa chaîne.

```

//: c13:FlowLayout1.java

// Démonstration de FlowLayout.

// <applet code=FlowLayout1
// width=300 height=250> </applet>

import javax.swing.*;

import java.awt.*;

import com.bruceeckel.swing.*;

public class FlowLayout1 extends JApplet {

    public void init() {
        Container cp = getContentPane();

        cp.setLayout(new FlowLayout());

        for(int i = 0; i < 20; i++)

            cp.add(new JButton("Button " + i));
    }

    public static void main(String[] args) {
        Console.run(new FlowLayout1(), 300, 250);
    }
} ///:~

```

Tous les composants sont compactés à leur taille minimum dans un **FlowLayout**, ce qui fait qu'on peut parfois obtenir un comportement surprenant. Par exemple, vu qu'un **JLabel** prend la taille de sa chaîne, une tentative de justifier à droite son texte ne donne pas de modification de l'affichage dans un **FlowLayout**.

## GridLayout

Un **GridLayout** permet de construire un tableau de composants, et lorsqu'on les ajoute ils sont placés de gauche à droite et de haut en bas dans la grille. Dans le constructeur on spécifie le nombre de rangées et de colonnes nécessaires, et celles-ci sont disposées en proportions identiques.

```

//: c13:GridLayout1.java

// Démonstration de GridLayout.

// <applet code=GridLayout1
// width=300 height=250> </applet>

import javax.swing.*;

import java.awt.*;

```

```
import com.bruceeckel.swing.*;

public class GridLayout1 extends JApplet {
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
            cp.add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        Console.run(new GridLayout1(), 300, 250);
    }
} ///:~
```

Dans ce cas il y a 21 cases mais seulement 20 boutons. La dernière case est laissée vide car il n'y a pas d'équilibrage dans un **GridLayout**.

## GridBagLayout

Le **GridBagLayout** nous donne un contrôle fin pour décider exactement comment les régions d'une fenêtre vont se positionner et se replacer lorsque la fenêtre est redimensionnée. Cependant, c'est aussi le plus compliqué des *layout managers*, et il est assez difficile à comprendre. Il est destiné principalement à la génération de code automatique par un constructeur d'interfaces utilisateurs graphiques [*GUI builder*] (les bons *GUI builders* utilisent **GridBagLayout** plutôt que le placement absolu). Si votre modèle est compliqué au point que vous sentiez le besoin d'utiliser le **GridBagLayout**, vous devrez dans ce cas utiliser un outil *GUI builder* pour générer ce modèle. Si vous pensez devoir en connaître les détails internes, je vous renvoie à *Core Java 2* par Horstmann & Cornell (Prentice-Hall, 1999), ou un livre dédié à Swing, comme point de départ.

## Positionnement absolu

Il est également possible de forcer la position absolue des composants graphiques de la façon suivante :

1. Positionner un *layout manager* **null** pour le **Container** : **setLayout(null)**.
2. Appeler **setBounds( )** ou **reshape( )** (selon la version du langage) pour chaque composant, en passant un rectangle de limites avec ses coordonnées en pixels. Ceci peut se faire dans le constructeur, ou dans **paint( )**, selon le but désiré.

Certains *GUI builders* utilisent cette approche de manière extensive, mais ce n'est en général pas la meilleure manière de générer du code. Les *GUI builders* les plus efficaces utilisent plutôt **GridBagLayout**.

## BoxLayout

Les gens ayant tellement de problèmes pour comprendre et utiliser **GridBagLayout**, Swing contient également le **BoxLayout**, qui offre la plupart des avantages du **GridBagLayout** sans en avoir la complexité, de sorte qu'on

peut souvent l'utiliser lorsqu'on doit coder à la main des *layouts* (encore une fois, si votre modèle devient trop compliqué, utilisez un *GUI builder* qui générera les **GridBagLayouts** à votre place). `BoxLayout` permet le contrôle du placement des composants soit verticalement soit horizontalement, et le contrôle de l'espace entre les composants en utilisant des choses appelées *struts* (entretoises) et *glue* (colle). D'abord, voyons comment utiliser **BoxLayout** directement, en faisant le même genre de démonstration que pour les autres layout managers :

```

//: c13:BoxLayout1.java
// BoxLayouts vertical et horizontal.
// <applet code=BoxLayout1
// width=450 height=200> </applet>

import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class BoxLayout1 extends JApplet {
    public void init() {
        JPanel jpv = new JPanel();
        jpv.setLayout(
            new BoxLayout(jpv, BoxLayout.Y_AXIS));
        for(int i = 0; i < 5; i++)
            jpv.add(new JButton(" " + i));
        JPanel jph = new JPanel();
        jph.setLayout(
            new BoxLayout(jph, BoxLayout.X_AXIS));
        for(int i = 0; i < 5; i++)
            jph.add(new JButton(" " + i));
        Container cp = getContentPane();
        cp.add(BorderLayout.EAST, jpv);
        cp.add(BorderLayout.SOUTH, jph);
    }
    public static void main(String[] args) {
        Console.run(new BoxLayout1(), 450, 200);
    }
} ///:~

```

Le constructeur du **BoxLayout** est un peu différent des autres *layout managers* : on fournit le **Container** que le **BoxLayout** doit contrôler comme premier argument, et la direction du *layout* comme deuxième argument.

Pour simplifier les choses, il y a un *container* spécial appelé **Box** qui utilise **BoxLayout** comme *manager* d'origine. L'exemple suivant place les composants horizontalement et verticalement en utilisant **Box**, qui possède deux méthodes **static** pour créer des *boxes* avec des alignements verticaux et horizontaux :

```

//: c13:Box1.java

// BoxLayouts vertical et horizontal.
// <applet code=Box1
// width=450 height=200> </applet>

import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box1 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();

        for(int i = 0; i < 5; i++)
            bv.add(new JButton("" + i));

        Box bh = Box.createHorizontalBox();

        for(int i = 0; i < 5; i++)
            bh.add(new JButton("" + i));

        Container cp = getContentPane();
        cp.add(BorderLayout.EAST, bv);
        cp.add(BorderLayout.SOUTH, bh);
    }

    public static void main(String[] args) {
        Console.run(new Box1(), 450, 200);
    }
} ///:~

```

Une fois qu'on a obtenu un **Box**, on le passe en second argument quand on ajoute des composants au *content pane*.

Les *struts* ajoutent de l'espace entre les composants, mesuré en pixels. Pour utiliser un *strut*, on l'ajoute simplement entre les ajouts de composants que l'on veut séparer :

```

//: c13:Box2.java

// Ajout de struts.
// <applet code=Box2
// width=450 height=300> </applet>

import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box2 extends JApplet {

    public void init() {
        Box bv = Box.createVerticalBox();

        for(int i = 0; i < 5; i++) {
            bv.add(new JButton(" " + i));

            bv.add(Box.createVerticalStrut(i*10));
        }

        Box bh = Box.createHorizontalBox();

        for(int i = 0; i < 5; i++) {
            bh.add(new JButton(" " + i));

            bh.add(Box.createHorizontalStrut(i*10));
        }

        Container cp = getContentPane();

        cp.add(BorderLayout.EAST, bv);
        cp.add(BorderLayout.SOUTH, bh);
    }

    public static void main(String[] args) {
        Console.run(new Box2(), 450, 300);
    }
} ///:~

```

Les struts séparent les composant d'une quantité fixe, mais *glue* fait le contraire : il sépare les composant d'autant que possible. C'est donc plus un *spring* (ressort) qu'une *glue* (et dans la conception sur laquelle ceci était basé, cela s'appelait *springs* et *struts*, ce qui rend un peu mystérieux le choix de ce terme).

```

//: c13:Box3.java

// Utilisation de Glue.

```

```

// <applet code=Box3
// width=450 height=300> </applet>

import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box3 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();
        bv.add(new JLabel("Hello"));
        bv.add(Box.createVerticalGlue());
        bv.add(new JLabel("Applet"));
        bv.add(Box.createVerticalGlue());
        bv.add(new JLabel("World"));
        Box bh = Box.createHorizontalBox();
        bh.add(new JLabel("Hello"));
        bh.add(Box.createHorizontalGlue());
        bh.add(new JLabel("Applet"));
        bh.add(Box.createHorizontalGlue());
        bh.add(new JLabel("World"));
        bv.add(Box.createVerticalGlue());
        bv.add(bh);
        bv.add(Box.createVerticalGlue());
        getContentPane().add(bv);
    }

    public static void main(String[] args) {
        Console.run(new Box3(), 450, 300);
    }
} ///:~

```

Un *strut* fonctionne dans une direction, mais une *rigid area* (surface rigide) fixe l'espace entre les composants dans chaque direction :

```

//: c13:Box4.java

// Des Rigid Areas sont comme des paires de struts.

```

```

// <applet code=Box4
// width=450 height=300> </applet>

import javax.swing.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class Box4 extends JApplet {
    public void init() {
        Box bv = Box.createVerticalBox();

        bv.add(new JButton("Top"));

        bv.add(Box.createRigidArea(
            new Dimension(120, 90)));

        bv.add(new JButton("Bottom"));

        Box bh = Box.createHorizontalBox();

        bh.add(new JButton("Left"));

        bh.add(Box.createRigidArea(
            new Dimension(160, 80)));

        bh.add(new JButton("Right"));

        bv.add(bh);

        getContentPane().add(bv);
    }

    public static void main(String[] args) {
        Console.run(new Box4(), 450, 300);
    }
} ///:~

```

Il faut savoir que les *rigid areas* sont un peu controversées. Comme elles utilisent des valeurs absolues, certaines personnes pensent qu'elles causent plus de problèmes qu'elles n'en résolvent.

## La meilleure approche ?

Swing est puissant; il peut faire beaucoup de choses en quelques lignes de code. Les exemples de ce livre sont raisonnablement simples, et dans un but d'apprentissage il est normal de les écrire à la main. On peut en fait réaliser pas mal de choses en combinant des *layouts* simples. A un moment donné, il devient cependant déraisonnable de coder à la main des formulaires de GUI. Cela devient trop compliqué et ce n'est pas une bonne manière d'utiliser son temps de programmation. Les concepteurs de Java et de Swing ont orienté le langage et ses bibliothèques de manière à soutenir des outils de construction de GUI, qui ont été créés dans le but de rendre la tâche de programmation plus facile. A partir du moment où on comprend ce qui se passe dans les layouts et



comment traiter les événements (décrits plus loin), il n'est pas particulièrement important de connaître effectivement tous les détails sur la façon de positionner les composants à la main. Laissons les outils appropriés le faire pour nous (Java est après tout conçu pour augmenter la productivité du programmeur).

## Le modèle d'événements de Swing

Dans le modèle d'événements de Swing, un composant peut initier (envoyer [*fire*]) un événement. Chaque type d'événement est représenté par une classe distincte. Lorsqu'un événement est envoyé, il est reçu par un ou plusieurs écouteurs [*listeners*], qui réagissent à cet événement. De ce fait, la source d'un événement et l'endroit où cet événement est traité peuvent être séparés. Puisqu'on utilise en général les composants Swing tels quels, mais qu'il faut écrire du code appelé lorsque les composants reçoivent un événement, ceci est un excellent exemple de la séparation de l'interface et de l'implémentation.

Chaque écouteur d'événements [*event listener*] est un objet d'une classe qui implémente une **interface** particulière de type *listener*. En tant que programmeur, il faut créer un objet *listener* et l'enregistrer avec le composant qui envoie l'événement. Cet enregistrement se fait par appel à une méthode **addXXXListener()** du composant envoyant l'événement, dans lequel **XXX** représente le type d'événement qu'on écoute. On peut facilement savoir quels types d'événements peuvent être gérés en notant les noms des méthodes `addListener`, et si on essaie d'écouter des événements erronés, l'erreur sera signalée à la compilation. On verra plus loin dans ce chapitre que les JavaBeans utilisent aussi les noms des méthodes `addListener` pour déterminer quels événements un Bean peut gérer.

Toute notre logique des événements va se trouver dans une classe *listener*. Lorsqu'on crée une classe *listener*, la seule restriction est qu'elle doit implémenter l'interface appropriée. On peut créer une classe *listener* globale, mais on est ici dans un cas où les classes internes sont assez utiles, non seulement parce qu'elles fournissent un groupement logique de nos classes *listener* à l'intérieur des classes d'interface utilisateur ou de logique métier qu'elles desservent, mais aussi (comme on le verra plus tard) parce que le fait qu'un objet d'une classe interne garde une référence à son objet parent fournit une façon élégante d'appeler à travers les frontières des classes et des sous-systèmes.

Jusqu'ici, tous les exemples de ce chapitre ont utilisé le modèle d'événements Swing, mais le reste de cette section va préciser les détails de ce modèle.

### Événements et types de *listeners*

Chaque composant Swing contient des méthodes **addXXXListener()** et **removeXXXListener()** de manière à ce que les types de *listeners* adéquats puissent être ajoutés et enlevés de chaque composant. On remarquera que le **XXX** dans chaque cas représente également l'argument de cette méthode, par exemple :

**addMyListener(MyListener m)**. Le tableau suivant contient les événements, listeners et méthodes de base associées aux composants de base qui supportent ces événements particuliers en fournissant les méthodes **addXXXListener()** et **removeXXXListener()**. Il faut garder en tête que le modèle d'événements est destiné à être extensible, et donc on pourra rencontrer d'autres types d'événements et de *listeners* non couverts par ce tableau.

Événement, interface <i>listener</i> et méthodes <code>add</code> et <code>remove</code>	Composants supportant cet événement
<b>ActionEvent</b> <b>ActionListener</b> <b>addActionListener()</b> <b>removeActionListener()</b>	<b>JButton, JList, JPasswordField,</b> <b>JMenuItem</b> et ses dérivés, comprenant <b>JCheckBoxMenuItem, JMenu, et</b> <b>JpopupMenu.</b>

<b>AdjustmentEvent</b> <b>AdjustmentListener</b> <b>addAdjustmentListener()</b> <b>removeAdjustmentListener()</b>	<b>JScrollbar</b> et tout ce qu'on crée qui implémente l'interface <b>Adjustable</b> .
<b>ComponentEvent</b> <b>ComponentListener</b> <b>addComponentListener()</b> <b>removeComponentListener()</b>	<b>*Component</b> et ses dérivés, comprenant <b>JButton, JCanvas, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, JFrame, JLabel, JList, JScrollbar, JTextArea, et JTextField</b> .
<b>ContainerEvent</b> <b>ContainerListener</b> <b>addContainerListener()</b> <b>removeContainerListener()</b>	<b>Container</b> et ses dérivés, comprenant <b>JPanel, JApplet, JScrollPane, Window, JDialog, JFileDialog, et JFrame</b> .
<b>FocusEvent</b> <b>FocusListener</b> <b>addFocusListener()</b> <b>removeFocusListener()</b>	<b>Component</b> et dérivés*.
<b>KeyEvent</b> <b>KeyListener</b> <b>addKeyListener()</b> <b>removeKeyListener()</b>	<b>Component</b> et dérivés*.
<b>MouseEvent</b> (à la fois pour les clics et pour le déplacement) <b>MouseListener</b> <b>addMouseListener()</b> <b>removeMouseListener()</b>	<b>Component</b> et dérivés*.
<b>MouseEvent</b> <a href="#">[68]</a> (à la fois pour les clics et pour le déplacement) <b>MouseMotionListener</b> <b>addMouseMotionListener()</b> <b>removeMouseMotionListener()</b>	<b>Component</b> et dérivés*.
<b>WindowEvent</b> <b>WindowListener</b> <b>addWindowListener()</b> <b>removeWindowListener()</b>	<b>Window</b> et ses dérivés, comprenant <b>JDialog, JFileDialog, and JFrame</b> .
<b>ItemEvent</b> <b>ItemListener</b> <b>addItemListener()</b> <b>removeItemListener()</b>	<b>JCheckBox, JCheckBoxMenuItem, JComboBox, JList</b> , et tout ce qui implémente l'interface <b>ItemSelectable</b> .
<b>TextEvent</b> <b>TextListener</b> <b>addTextListener()</b> <b>removeTextListener()</b>	Tout ce qui est dérivé de <b>JTextComponent</b> , comprenant <b>JTextArea</b> et <b>JTextField</b> .