

La méthode **paintComponent()** dessine le carré autour du panneau, et le x ou le o. C'est rempli de calculs fastidieux, mais c'est direct.

Les clics de souris sont capturés par le **MouseListener**, qui vérifie d'abord si le panneau a déjà quelque chose d'écrit sur lui. Si ce n'est pas le cas, on recherche la fenêtre parente pour déterminer à qui est le tour, et on positionne l'état du **ToeButton** en conséquence. Le **ToeButton** retrouve le parent par le mécanisme de la classe interne, et passe au tour suivant. Si le bouton affiche déjà un x ou un o, son affichage est inversé. On peut voir dans ces calculs l'usage pratique du if-else ternaire décrit au Chapitre 3. On repeint le **ToeButton** chaque fois qu'il change d'état.

Le constructeur de **ToeDialog** est assez simple : il ajoute à un **GridLayout** autant de boutons que demandé, puis redimensionne chaque bouton à 50 pixels.

**TicTacToe** installe l'ensemble de l'application par la création de **JTextFields** (pour entrer le nombre de rangées et colonnes de la grille de boutons) et le bouton «go» avec son **ActionListener**. Lorsqu'on appuie sur le bouton, les données dans les **JTextFields** doivent être récupérées et, puisqu'elles sont au format **String**, transformées en **ints** en utilisant la méthode statique **Integer.parseInt()**.

## Dialogues pour les fichiers *[File dialogs]*

Certains systèmes d'exploitation ont des boîtes de dialogue standard pour gérer certaines choses telles que les fontes, les couleurs, les imprimantes, et cetera. En tout cas, pratiquement tous les systèmes d'exploitation graphiques fournissent les moyens d'ouvrir et de sauver les fichiers, et le **JFileChooser** de Java les encapsule pour une utilisation facile.

L'application suivante utilise deux sortes de dialogues **JFileChooser**, un pour l'ouverture et un pour la sauvegarde. La plupart du code devrait maintenant être familière, et toute la partie intéressante se trouve dans les *action listeners* pour les différents clics de boutons :

```

//: c13:FileChooserTest.java

// Démonstration de boîtes de dialogues de fichiers.

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import com.bruceeckel.swing.*;

public class FileChooserTest extends JFrame {

    JTextField

        filename = new JTextField(),

        dir = new JTextField();

    JButton

        open = new JButton("Open"),

        save = new JButton("Save");

    public FileChooserTest() {

```

```

JPanel p = new JPanel();

open.addActionListener(new OpenL());

p.add(open);

save.addActionListener(new SaveL());

p.add(save);

Container cp = getContentPane();

cp.add(p, BorderLayout.SOUTH);

dir.setEditable(false);

filename.setEditable(false);

p = new JPanel();

p.setLayout(new GridLayout(2,1));

p.add(filename);

p.add(dir);

cp.add(p, BorderLayout.NORTH);
}

class OpenL implements ActionListener {

    public void actionPerformed(ActionEvent e) {

        JFileChooser c = new JFileChooser();

        // Démontre le dialogue "Open" :

        int rVal =

            c.showOpenDialog(FileChooserTest.this);

        if(rVal == JFileChooser.APPROVE_OPTION) {

            filename.setText(

                c.getSelectedFile().getName());

            dir.setText(

                c.getCurrentDirectory().toString());

        }

        if(rVal == JFileChooser.CANCEL_OPTION) {

            filename.setText("You pressed cancel");

            dir.setText("");

        }

    }

}

class SaveL implements ActionListener {

```

```

public void actionPerformed(ActionEvent e) {

    JFileChooser c = new JFileChooser();

    // Démontre le dialogue "Save" :

    int rVal =

        c.showSaveDialog(FileChooserTest.this);

    if(rVal == JFileChooser.APPROVE_OPTION) {

        filename.setText(

            c.getSelectedFile().getName());

        dir.setText(

            c.getCurrentDirectory().toString());

    }

    if(rVal == JFileChooser.CANCEL_OPTION) {

        filename.setText("You pressed cancel");

        dir.setText("");

    }

}

}

public static void main(String[] args) {

    Console.run(new FileChooserTest(), 250, 110);

}

} ///:~

```

Remarquons qu'il y a de nombreuses variantes applicables à **JFileChooser**, parmi lesquelles des filtres pour restreindre les noms de fichiers acceptés.

Pour un dialogue d'ouverture de fichier, on appelle **showOpenDialog()**, et pour un dialogue de sauvegarde de fichier, on appelle **showSaveDialog()**. Ces commandes ne reviennent que lorsque le dialogue est fermé. L'objet **JFileChooser** existe encore, de sorte qu'on peut en lire les données. Les méthodes **getSelectedFile()** et **getCurrentDirectory()** sont deux façons d'obtenir les résultats de l'opération. Si celles-ci renvoient **null**, cela signifie que l'utilisateur a abandonné le dialogue.

## HTML sur des composants Swing

Tout composant acceptant du texte peut également accepter du texte HTML, qu'il reformatera selon les règles HTML. Ceci signifie qu'on peut très facilement ajouter du texte de fantaisie à un composant Swing. Par exemple :

```

///: c13:HTMLButton.java

// Mettre du texte HTML sur des composants Swing.

// <applet code=HTMLButton width=200 height=500>

```

```

// </applet>

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

public class HTMLButton extends JApplet {
    JButton b = new JButton("<html><b><font size=+2>" +
        "<center>Hello!<br><i>Press me now!");
    public void init() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                getContentPane().add(new JLabel("<html>" +
                    "<i><font size=+4>Kapow!"));
                // Force un réalignement pour
                // inclure le nouveau label:
                validate();
            }
        });
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(b);
    }
    public static void main(String[] args) {
        Console.run(new HTMLButton(), 200, 500);
    }
} ///:~

```

Le texte doit commencer avec `<html>`, et ensuite on peut utiliser les *tags* HTML normaux. Remarquons que les *tags* de fermeture ne sont pas obligatoires.

L'**ActionListener** ajoute au formulaire un nouveau **JLabel** contenant du texte HTML. Comme ce label n'est pas ajouté dans la méthode **init()**, on doit appeler la méthode **validate()** du conteneur de façon à forcer une redistribution des composants (et de ce fait un affichage du nouveau label).

On peut également ajouter du texte HTML à un **JTabbedPane**, **JMenuItem**, **JToolTip**, **JRadioButton** ou un **JCheckBox**.

## Curseurs [*sliders*] et barres de progression [*progress bars*]

Un *slider* (qu'on a déjà utilisé dans l'exemple du sinus) permet à l'utilisateur de rentrer une donnée en déplaçant un point en avant ou en arrière, ce qui est intuitif dans certains cas (un contrôle de volume, par exemple). Un *progress bar* représente une donnée en remplissant proportionnellement un espace vide pour que l'utilisateur ait une idée de la valeur. Mon exemple favori pour ces composants consiste à simplement lier le curseur à la barre de progression, de sorte que lorsqu'on déplace le curseur la barre de progression change en conséquence :

```

//: c13:Progress.java

// Utilisation de progress bars et de sliders.

// <applet code=Progress
//   width=300 height=200></applet>

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

public class Progress extends JApplet {
    JProgressBar pb = new JProgressBar();
    JSlider sb =
        new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new GridLayout(2,1));
        cp.add(pb);
        sb.setValue(0);
        sb.setPaintTicks(true);
        sb.setMajorTickSpacing(20);
        sb.setMinorTickSpacing(5);
        sb.setBorder(new TitledBorder("Slide Me"));
        pb.setModel(sb.getModel()); // Partage du modèle
        cp.add(sb);
    }
    public static void main(String[] args) {
        Console.run(new Progress(), 300, 200);
    }
}

```

```

    }
} ///:~

```

La clé du lien entre les deux composants se trouve dans le partage de leur modèle, dans la ligne :

```
pb.setModel(sb.getModel());
```

Naturellement, on pourrait aussi contrôler les deux composants en utilisant un listener, mais ceci est plus direct pour les cas simples.

Le **JProgressBar** est assez simple, mais le **JSlider** a un grand nombre d'options, telles que l'orientation et les graduations mineures et majeures. Remarquons la simplicité de l'ajout d'une bordure avec titre.

## Arbres [Trees]

L'utilisation d'un **JTree** peut être aussi simple que ceci :

```
add(new JTree(
    new Object[] { "this", "that", "other" }));
```

Ceci affiche un arbre rudimentaire. L'API pour les arbres est vaste, probablement une des plus importantes de Swing. On peut faire à peu près tout ce qu'on veut avec des arbres, mais les tâches plus complexes demandent davantage de recherches et d'expérimentations.

Heureusement, il y a un niveau intermédiaire fourni dans la bibliothèque : les composants arbres par défaut, qui font en général ce dont on a besoin, de sorte que la plupart du temps on peut se contenter de ces composants, et ce n'est que dans des cas particuliers qu'on aura besoin d'approfondir et de comprendre plus en détail les arbres.

L'exemple suivant utilise les composants arbres par défaut pour afficher un arbre dans une applet. Lorsqu'on appuie sur le bouton, un nouveau sous-arbre est ajouté sous le noeud sélectionné (si aucun noeud n'est sélectionné, le noeud racine est utilisé) :

```

//: c13:Trees.java

// Exemple d'arbre Swing simple. Les arbres peuvent
// être beaucoup plus complexes que celui-ci.
// <applet code=Trees
//   width=250 height=250></applet>

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import javax.swing.tree.*;

import com.bruceeckel.swing.*;

// Prend un tableau de Strings et crée un noeud à partir
// du premier élément, et des feuilles avec les autres :

```

```

class Branch {
    DefaultMutableTreeNode r;

    public Branch(String[] data) {
        r = new DefaultMutableTreeNode(data[0]);
        for(int i = 1; i < data.length; i++)
            r.add(new DefaultMutableTreeNode(data[i]));
    }

    public DefaultMutableTreeNode node() {
        return r;
    }
}

```

```

public class Trees extends JApplet {
    String[][] data = {
        { "Colors", "Red", "Blue", "Green" },
        { "Flavors", "Tart", "Sweet", "Bland" },
        { "Length", "Short", "Medium", "Long" },
        { "Volume", "High", "Medium", "Low" },
        { "Temperature", "High", "Medium", "Low" },
        { "Intensity", "High", "Medium", "Low" },
    };

    static int i = 0;

    DefaultMutableTreeNode root, child, chosen;

    JTree tree;

    DefaultTreeModel model;

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

        root = new DefaultMutableTreeNode("root");
        tree = new JTree(root);

        // On l'ajoute et on le rend scrollable :
        cp.add(new JScrollPane(tree),
            BorderLayout.CENTER);

        // Obtention du modèle de l'arbre :
        model =(DefaultTreeModel)tree.getModel();
    }
}

```

```

JButton test = new JButton("Press me");

test.addActionListener(new ActionListener() {

    public void actionPerformed(ActionEvent e){

        if(i < data.length) {

            child = new Branch(data[i++]).node();

            // Quel est le dernier élément cliqué ?

            chosen = (DefaultMutableTreeNode)

                tree.getLastSelectedPathComponent();

            if(chosen == null) chosen = root;

            // Le modèle créera l'événement approprié.

            // En réponse, l'arbre se remettra à jour :

            model.insertNodeInto(child, chosen, 0);

            // Ceci place le nouveau noeud

            // sur le noeud actuellement sélectionné.

        }

    }

});

// Change les couleurs des boutons :

test.setBackground(Color.blue);

test.setForeground(Color.white);

JPanel p = new JPanel();

p.add(test);

cp.add(p, BorderLayout.SOUTH);

}

public static void main(String[] args) {

    Console.run(new Trees(), 250, 250);

}

} ///::~

```

La première classe, **Branch**, est un outil qui prend un tableau et construit un **DefaultMutableTreeNode** avec la première **String** comme racine, et le reste des **Strings** du tableau pour les feuilles. Ensuite **node()** peut être appelé pour créer la racine de cette branche.

La classe **Trees** contient un tableau de **Strings** à deux dimensions à partir duquel des **Branches** peuvent être créées, ainsi qu'un **static int i** pour servir d'index à travers ce tableau. L'objet **DefaultMutableTreeNode** contient les noeuds, mais la représentation physique à l'écran est contrôlée par le **JTree** et son modèle associé, le **DefaultTreeModel**. Notons que lorsque le **JTree** est ajouté à l'applet, il est encapsulé dans un **JScrollPane** :



c'est suffisant pour permettre un scrolling automatique.

Le **JTree** est contrôlé par son modèle. Lorsqu'on modifie les données du modèle, celui-ci génère un événement qui force le **JTree** à effectuer les mises à jour nécessaires de la partie visible de la représentation de l'arbre. Dans **init()**, le modèle est obtenu par appel à **getModel()**. Lorsqu'on appuie sur le bouton, une nouvelle branche est créée. Ensuite le composant actuellement sélectionné est recherché (on utilise la racine si rien n'est sélectionné) et la méthode **insertNodeInto()** du modèle effectue la modification de l'arbre et provoque sa mise à jour.

Un exemple comme ci-dessus peut vous donner ce dont vous avez besoin pour utiliser un arbre. Cependant les arbres ont la possibilité de faire à peu près tout ce qui est imaginable ; chaque fois que le mot *default* apparaît dans l'exemple ci-dessus, on peut y substituer sa propre classe pour obtenir un comportement différent. Mais attention : la plupart de ces classes a une interface importante, de sorte qu'on risque de passer du temps à comprendre la complexité des arbres. Cependant, on a affaire ici à une bonne conception, et les autres solutions sont en général bien moins bonnes.

## Tables

Comme les arbres, les tables en Swing sont vastes et puissantes. Elles sont destinées principalement à être la populaire grille d'interface avec les bases de données via la Connectivité Bases de Données Java : *Java DataBase Connectivity* (JDBC, présenté dans le Chapitre 15) et pour cela elles ont une flexibilité énorme, que l'on paie en complexité. Il y a ici suffisamment pour servir de base à un tableur complet et pourrait probablement être le sujet d'un livre complet. Cependant, il est également possible de créer une **JTable** relativement simple si on en comprend les bases.

La **JTable** contrôle la façon dont les données sont affichées, tandis que le **TableModel** contrôle les données elles-mêmes. Donc pour créer une **JTable** on créera d'abord un **TableModel**. On peut implémenter complètement l'interface **TableModel**, mais il est souvent plus simple d'hériter de la classe utilitaire **AbstractTableModel** :

```

//: c13:Table.java

// Démonstration simple d'une JTable.

// <applet code=Table
// width=350 height=200></applet>

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

public class Table extends JApplet {
    JTextArea txt = new JTextArea(4, 20);

    // Le TableModel contrôle toutes les données :

```

```

class DataModel extends AbstractTableModel {
    Object[][] data = {
        {"one", "two", "three", "four"},
        {"five", "six", "seven", "eight"},
        {"nine", "ten", "eleven", "twelve"},
    };

    // Imprime les données lorsque la table change :
    class TML implements TableModelListener {
        public void tableChanged(TableModelEvent e){
            txt.setText(""); // Vider le texte
            for(int i = 0; i < data.length; i++) {
                for(int j = 0; j < data[0].length; j++)
                    txt.append(data[i][j] + " ");
                txt.append("\n");
            }
        }
    }

    public DataModel() {
        addTableModelListener(new TML());
    }

    public int getColumnCount() {
        return data[0].length;
    }

    public int getRowCount() {
        return data.length;
    }

    public Object getValueAt(int row, int col) {
        return data[row][col];
    }

    public void
    setValueAt(Object val, int row, int col) {
        data[row][col] = val;

        // Indique que le changement a eu lieu :
        fireTableDataChanged();
    }
}

```

```

    }

    public boolean
    isCellEditable(int row, int col) {

        return true;

    }

}

public void init() {

    Container cp = getContentPane();

    JTable table = new JTable(new DataModel());

    cp.add(new JScrollPane(table));

    cp.add(BorderLayout.SOUTH, txt);

}

public static void main(String[] args) {

    Console.run(new Table(), 350, 200);

}

} ///:~

```

**DataModel** contient un tableau de données, mais on pourrait aussi obtenir les données depuis une autre source telle qu'une base de données. Le constructeur ajoute un **TableModelListener** qui imprime le tableau chaque fois que la table est modifiée. Les autres méthodes suivent les conventions de nommage des Beans ; elles sont utilisées par la **JTable** lorsqu'elle veut présenter les informations contenues dans **DataModel**.

**AbstractTableModel** fournit des méthodes par défaut pour **setValueAt()** et **isCellEditable()** qui interdisent les modifications de données, de sorte que ces méthodes devront être redéfinies si on veut pouvoir modifier les données.

Une fois obtenu un **TableModel**, il suffit de le passer au constructeur de la **JTable**. Tous les détails concernant l'affichage, les modifications et la mise à jour seront automatiquement gérés. Cet exemple place également la **JTable** dans un **JScrollPane**.

## Sélection de l'aspect de l'interface *[Look & Feel]*

Un des aspects très intéressants de Swing est le *Pluggable Look & Feel*. Il permet à un programme d'émuler le *look and feel* de divers environnements d'exploitation. On peut même faire toutes sortes de choses comme par exemple changer le *look and feel* pendant l'exécution du programme. Toutefois, en général on désire soit sélectionner le *look and feel* toutes plateformes (qui est le Metal de Swing), soit sélectionner le *look and feel* du système courant, de sorte à donner l'impression que le programme Java a été créé spécifiquement pour ce système. Le code permettant de sélectionner chacun de ces comportements est assez simple, mais il faut l'exécuter *avant* de créer les composants visuels, car ceux-ci seront créés selon le *look and feel* courant et ne seront pas changés par le simple changement de *look and feel* au milieu du programme (ce processus est compliqué et peu banal, et nous en laisserons le développement à des livres spécifiques sur Swing).

En fait, si on veut utiliser le *look and feel* toutes plateformes (metal) qui est la caractéristique des programmes Swing, il n'y a rien de particulier à faire, c'est la valeur par défaut. Si au contraire on veut utiliser le *look and feel* de l'environnement d'exploitation courant, il suffit d'insérer le code suivant, normalement au début du

**main()** mais de toutes façons avant d'ajouter des composants :

```
try {
    UIManager.setLookAndFeel(UIManager.
        getSystemLookAndFeelClassName());
} catch(Exception e) {}
```

Il n'y a pas besoin de faire quoi que ce soit dans la clause **catch** car le **UIManager** se positionnera par défaut au *look and feel* toutes plateformes si votre tentative d'installation d'un des autres échoue. Toutefois, pour le *debug*, l'exception peut être utile, de sorte qu'on peut au minimum placer une instruction d'impression dans la clause **catch**.

Voici un programme qui utilise un argument de ligne de commande pour sélectionner un *look and feel*, et montre à quoi ressemblent différents composants dans le *look and feel* choisi :

```
//: c13:LookAndFeel.java
// Sélection de divers looks & feels.

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

public class LookAndFeel extends JFrame {
    String[] choices = {
        "eeny", "meeny", "minie", "moe", "toe", "you"
    };
    Component[] samples = {
        new JButton("JButton"),
        new JTextField("JTextField"),
        new JLabel("JLabel"),
        new JCheckBox("JCheckBox"),
        new JRadioButton("Radio"),
        new JComboBox(choices),
        new JList(choices),
    };

    public LookAndFeel() {
        super("Look And Feel");
        Container cp = getContentPane();
```

```

        cp.setLayout(new FlowLayout());
        for(int i = 0; i < samples.length; i++)
            cp.add(samples[i]);
    }
    private static void usageError() {
        System.out.println(
            "Usage:LookAndFeel [cross|system|motif]");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length == 0) usageError();
        if(args[0].equals("cross")) {
            try {
                UIManager.setLookAndFeel(UIManager.
                    getCrossPlatformLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace(System.err);
            }
        } else if(args[0].equals("system")) {
            try {
                UIManager.setLookAndFeel(UIManager.
                    getSystemLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace(System.err);
            }
        } else if(args[0].equals("motif")) {
            try {
                UIManager.setLookAndFeel("com.sun.java."+
                    "swing.plaf.motif.MotifLookAndFeel");
            } catch(Exception e) {
                e.printStackTrace(System.err);
            }
        } else usageError();
        // Remarquons que le look & feel doit être positionné

```

```

    // avant la création des composants.

    Console.run(new LookAndFeel(), 300, 200);

}

} ///:~

```

On peut voir qu'une option permet de spécifier explicitement une chaîne pour un *look and feel* donné, comme par exemple **MotifLookAndFeel**. Toutefois seuls celui-ci et le *look and feel* metal sont les seuls à pouvoir être utilisés légalement sur toute plateforme ; bien qu'il existe des chaînes pour les *look and feels* Windows et Macintosh, ceux-ci ne peuvent être utilisés que sur leurs plateformes respectives (ceux-ci sont fournis lorsqu'on appelle **getSystemLookAndFeelClassName()** et qu'on est sur cette plateforme donnée).

Il est également possible de créer un package de *look and feel* sur mesure, par exemple si on crée un environnement de travail pour une société qui désire une apparence spéciale. C'est un gros travail qui est bien au-delà de la portée de ce livre (en fait vous découvrirez qu'il est au-delà de la portée de beaucoup de livres dédiés à Swing).

## Le presse-papier [*clipboard*]

JFC permet des opérations limitées avec le presse-papier système (dans le package **java.awt.datatransfer**). On peut copier des objets **String** dans le presse-papier en tant que texte, et on peut coller du texte depuis le presse-papier dans des objets **String**. Bien sûr, le presse-papier est prévu pour contenir n'importe quel type de données, mais la représentation de ces données dans le presse-papier est du ressort du programme effectuant les opérations de couper et coller. L'API *Java clipboard* permet ces extensions à l'aide du concept de «parfum» [*flavor*]. Les données en provenance du presse-papier sont associées à un ensemble de *flavors* dans lesquels on peut les convertir (par exemple, un graphe peut être représenté par une chaîne de nombres ou par une image) et on peut vérifier si les données contenues dans le presse-papier acceptent le *flavor* qui nous intéresse.

Le programme suivant est une démonstration simple de couper, copier et coller des données **String** dans une **JTextArea**. On remarquera que les séquences clavier utilisées normalement pour couper, copier et coller fonctionnent également. Mais si on observe un **JTextField** ou un **JTextArea** dans tout autre programme, on verra qu'ils acceptent aussi automatiquement les séquences clavier du presse-papier. Cet exemple ajoute simplement un contrôle du presse-papier par le programme, et on peut utiliser ces techniques pour capturer du texte du presse-papier depuis autre chose qu'un **JTextComponent**.

```

/// c13:CutAndPaste.java
// Utilisation du presse-papier.

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import java.awt.datatransfer.*;

import com.bruceeckel.swing.*;

public class CutAndPaste extends JFrame {

    JMenuBar mb = new JMenuBar();

```

```

JMenu edit = new JMenu("Edit");

JMenuItem

    cut = new JMenuItem("Cut"),
    copy = new JMenuItem("Copy"),
    paste = new JMenuItem("Paste");

JTextArea text = new JTextArea(20, 20);

Clipboard clipbd =
    getToolkit().getSystemClipboard();

public CutAndPaste() {
    cut.addActionListener(new CutL());
    copy.addActionListener(new CopyL());
    paste.addActionListener(new PasteL());
    edit.add(cut);
    edit.add(copy);
    edit.add(paste);
    mb.add(edit);
    setJMenuBar(mb);
    getContentPane().add(text);
}

class CopyL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String selection = text.getSelectedText();
        if (selection == null)
            return;
        StringSelection clipString =
            new StringSelection(selection);
        clipbd.setContents(clipString, clipString);
    }
}

class CutL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        String selection = text.getSelectedText();
        if (selection == null)
            return;

```

```

StringSelection clipString =
    new StringSelection(selection);
clipbd.setContents(clipString, clipString);
text.replaceRange("",
    text.getSelectionStart(),
    text.getSelectionEnd());
}
}

class PasteL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        Transferable clipData =
            clipbd.getContents(CutAndPaste.this);
        try {
            String clipString =
                (String)clipData.
                    getTransferData(
                        DataFlavor.stringFlavor);
            text.replaceRange(clipString,
                text.getSelectionStart(),
                text.getSelectionEnd());
        } catch (Exception ex) {
            System.err.println("Not String flavor");
        }
    }
}

public static void main(String[] args) {
    Console.run(new CutAndPaste(), 300, 200);
}
} ///:~

```

La création et l'ajout du menu et du **JTextArea** devraient être maintenant une activité naturelle. Ce qui est différent est la création du champ **Clipboard clipbd**, qui est faite à l'aide du **Toolkit**.

Toutes les actions sont effectuées dans les *listeners*. Les listeners **CopyL** et **CutL** sont identiques à l'exception de la dernière ligne de **CutL**, qui efface la ligne qui a été copiée. Les deux lignes particulières sont la création d'un objet **StringSelection** à partir du **String**, et l'appel à **setContents()** avec ce **StringSelection**. C'est tout ce qu'il y a à faire pour mettre une **String** dans le presse-papier.



Dans PasteL, les données sont extraites du presse-papier à l'aide de `getContents()`. Ce qu'il en sort est un objet **Transferable** assez anonyme, et on ne sait pas exactement ce qu'il contient. Un moyen de le savoir est d'appeler `getTransferDataFlavors()`, qui renvoie un tableau d'objets **DataFlavor** indiquant quels *flavors* sont acceptés par cet objet. On peut aussi le demander directement à l'aide de `isDataFlavorSupported()`, en passant en paramètre le *flavor* qui nous intéresse. Dans ce programme, toutefois, on utilise une approche téméraire : on appelle `getTransferData()` en supposant que le contenu accepte le *flavor* **String**, et si ce n'est pas le cas le problème est pris en charge par le traitement d'exception.

Dans le futur, on peut s'attendre à ce qu'il y ait plus de *flavors* acceptés.

## Empaquetage d'une applet dans un fichier JAR

Une utilisation importante de l'utilitaire JAR est l'optimisation du chargement d'une applet. En Java 1.0, les gens avaient tendance à entasser tout leur code dans une seule classe, de sorte que le client ne devait faire qu'une seule requête au serveur pour télécharger le code de l'applet. Ceci avait pour résultat des programmes désordonnés, difficiles à lire (et à maintenir), et d'autre part le fichier **.class** n'était pas compressé, de sorte que le téléchargement n'était pas aussi rapide que possible.

Les fichiers JAR résolvent le problème en compressant tous les fichiers **.class** en un seul fichier qui est téléchargé par le navigateur. On peut maintenant avoir une conception correcte sans se préoccuper du nombre de fichiers **.class** qui seront nécessaires, et l'utilisateur aura un temps de téléchargement beaucoup plus court.

Prenons par exemple **TicTacToe.java**. Il apparaît comme une seule classe, mais en fait il contient cinq classes internes, ce qui fait six au total. Une fois le programme compilé on l'emballe dans un fichier JAR avec l'instruction :

```
jar cf TicTacToe.jar *.class
```

Ceci suppose que dans le répertoire courant il n'y a que les fichiers **.class** issus de **TicTacToe.java** (sinon on emporte du bagage supplémentaire).

On peut maintenant créer une page HTML avec le nouveau *tag* **archive** pour indiquer le nom du fichier JAR. Voici pour exemple le *tag* utilisant l'ancienne forme du *tag* HTML :

```
<head><title>TicTacToe Example Applet
</title></head>

<body>

<applet code=TicTacToe.class
        archive=TicTacToe.jar
        width=200 height=100>

</applet>

</body>
```

Il faudra le mettre dans la [nouvelle](#) forme (confuse, compliquée) montrée plus haut dans ce chapitre pour le faire fonctionner.

## Techniques de programmation

La programmation de *GUI* en Java étant une technologie évolutive, avec des modifications très importantes entre Java 1.0/1.1 et la bibliothèque Swing, certains styles de programmation anciens ont pu s'insinuer dans des exemples qu'on peut trouver pour Swing. D'autre part, Swing permet une meilleure programmation que ce que permettaient les anciens modèles. Dans cette partie, certains de ces problèmes vont être montrés en présentant et en examinant certains styles de programmation.

## Lier des événements dynamiquement

Un des avantages du modèle d'événements Swing est sa flexibilité. On peut ajouter ou retirer un comportement sur événement à l'aide d'un simple appel de méthode. L'exemple suivant le montre :

```

//: c13:DynamicEvents.java

// On peut modifier dynamiquement le comportement sur événement.
// Montre également plusieurs actions pour un événement.
// <applet code=DynamicEvents
//   width=250 height=400></applet>

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

public class DynamicEvents extends JApplet {
    ArrayList v = new ArrayList();
    int i = 0;
    JButton
        b1 = new JButton("Button1"),
        b2 = new JButton("Button2");
    JTextArea txt = new JTextArea();
    class B implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            txt.append("A button was pressed\n");
        }
    }
    class CountListener implements ActionListener {
        int index;
    }
}

```

```

    public CountListener(int i) { index = i; }

    public void actionPerformed(ActionEvent e) {
        txt.append("Counted Listener "+index+"\n");
    }
}

class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        txt.append("Button 1 pressed\n");

        ActionListener a = new CountListener(i++);
        v.add(a);

        b2.addActionListener(a);
    }
}

class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        txt.append("Button2 pressed\n");

        int end = v.size() - 1;

        if(end >= 0) {
            b2.removeActionListener(
                (ActionListener)v.get(end));
            v.remove(end);
        }
    }
}

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

    b1.addActionListener(new B());
    b1.addActionListener(new B1());
    b2.addActionListener(new B());
    b2.addActionListener(new B2());

    JPanel p = new JPanel();

    p.add(b1);
    p.add(b2);

    cp.add(BorderLayout.NORTH, p);
}

```

```

        cp.add(new JScrollPane(txt));
    }

    public static void main(String[] args) {
        Console.run(new DynamicEvents(), 250, 400);
    }
} ///:~

```

Les nouvelles astuces dans cet exemple sont :

1. Il y a plus d'un *listener* attaché à chaque **Button**. En règle générale, les composants gèrent les événements en tant que *multicast*, ce qui signifie qu'on peut enregistrer plusieurs *listeners* pour un seul événement. Pour les composants spéciaux dans lesquels un événement est géré en tant que *unicast*, on obtiendra une exception **TooManyListenersException**.
2. Lors de l'exécution du programme, les *listeners* sont ajoutés et enlevés du **Button b2** dynamiquement. L'ajout est réalisé de la façon vue précédemment, mais chaque composant a aussi une méthode **removeXXXListener()** pour enlever chaque type de *listener*.

Ce genre de flexibilité permet une grande puissance de programmation.

Il faut remarquer qu'il n'est pas garanti que les listeners d'événements soient appelés dans l'ordre dans lequel ils sont ajoutés (bien que la plupart des implémentations le fasse de cette façon).

## Séparation entre la logique applicative [*business logic*] et la logique de l'interface utilisateur [*UI logic*]

En général on conçoit les classes de manière à ce que chacune fasse une seule chose. Ceci est particulièrement important pour le code d'une interface utilisateur, car il arrive souvent qu'on lie ce qu'on fait à la manière dont on l'affiche. Ce genre de couplage empêche la réutilisation du code. Il est de loin préférable de séparer la logique applicative de la partie *GUI*. De cette manière, non seulement la logique applicative est plus facile à réutiliser, mais il est également plus facile de récupérer la *GUI*.

Un autre problème concerne les systèmes répartis [*multitiered systems*], dans lesquels les objets applicatifs se trouvent sur une machine séparée. Cette centralisation des règles applicatives permet des modifications ayant un effet immédiat pour toutes les nouvelles transactions, ce qui est une façon intéressante d'installer un système. Cependant, ces objets applicatifs peuvent être utilisés dans de nombreuses applications, et de ce fait ils ne devraient pas être liés à un mode d'affichage particulier. Ils devraient se contenter d'effectuer les opérations applicatives, et rien de plus.

L'exemple suivant montre comme il est facile de séparer la logique applicative du code *GUI* :

```

///: c13:Separation.java

// Séparation entre la logique GUI et les objets applicatifs.

// <applet code=Separation

// width=250 height=150> </applet>

import javax.swing.*;

import java.awt.*;

```

```

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

class BusinessLogic {
    private int modifier;

    public BusinessLogic(int mod) {
        modifier = mod;
    }

    public void setModifier(int mod) {
        modifier = mod;
    }

    public int getModifier() {
        return modifier;
    }

    // Quelques opérations applicatives :
    public int calculation1(int arg) {
        return arg * modifier;
    }

    public int calculation2(int arg) {
        return arg + modifier;
    }
}

public class Separation extends JApplet {
    JTextField
        t = new JTextField(15),
        mod = new JTextField(15);

    BusinessLogic bl = new BusinessLogic(2);

    JButton
        calc1 = new JButton("Calculation 1"),
        calc2 = new JButton("Calculation 2");

    static int getValue(JTextField tf) {

```

```

    try {
        return Integer.parseInt(tf.getText());
    } catch (NumberFormatException e) {
        return 0;
    }
}

class Calc1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText(Integer.toString(
            bl.calculation1(getValue(t))));
    }
}

class Calc2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText(Integer.toString(
            bl.calculation2(getValue(t))));
    }
}

// Si vous voulez que quelque chose se passe chaque fois
// qu'un JTextField est modifié, ajoutez ce listener :

class ModL implements DocumentListener {
    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
        bl.setModifier(getValue(mod));
    }
    public void removeUpdate(DocumentEvent e) {
        bl.setModifier(getValue(mod));
    }
}

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
    calc1.addActionListener(new Calc1L());

```

```

        calc2.addActionListener(new Calc2L());

        JPanel p1 = new JPanel();

        p1.add(calc1);

        p1.add(calc2);

        cp.add(p1);

        mod.getDocument().

            addDocumentListener(new ModL());

        JPanel p2 = new JPanel();

        p2.add(new JLabel("Modifier:"));

        p2.add(mod);

        cp.add(p2);

    }

    public static void main(String[] args) {

        Console.run(new Separation(), 250, 100);

    }

} ///:~

```

On peut voir que **BusinessLogic** est une classe toute simple, qui effectue ses opérations sans même avoir idée qu'elle puisse être utilisée dans un environnement GUI. Elle se contente d'effectuer son travail.

**Separation** garde la trace des détails de l'interface utilisateur, et elle communique avec **BusinessLogic** uniquement à travers son interface **public**. Toutes les opérations sont concentrées sur l'échange d'informations bidirectionnel entre l'interface utilisateur et l'objet **BusinessLogic**. De même, **Separation** fait uniquement son travail. Comme **Separation** sait uniquement qu'il parle à un objet **BusinessLogic** (c'est à dire qu'il n'est pas fortement couplé), il pourrait facilement être transformé pour parler à d'autres types d'objets.

Penser à séparer l'interface utilisateur de la logique applicative facilite également l'adaptation de code existant pour fonctionner avec Java.

## Une forme canonique

Les classes internes, le modèle d'événements de Swing, et le fait que l'ancien modèle d'événements soit toujours disponible avec de nouvelles fonctionnalités des bibliothèques qui reposent sur l'ancien style de programmation, ont ajouté un nouvel élément de confusion dans le processus de conception du code. Il y a maintenant encore plus de façons d'écrire du mauvais code.

À l'exception de circonstances qui tendent à disparaître, on peut toujours utiliser l'approche la plus simple et la plus claire : les classes *listener* (normalement des classes internes) pour tous les besoins de traitement d'événements. C'est la forme utilisée dans la plupart des exemples de ce chapitre.

En suivant ce modèle on devrait pouvoir réduire les lignes de programmes qui disent : «Je me demande ce qui a provoqué cet événement». Chaque morceau de code doit se concentrer sur une action, et non pas sur des vérifications de types. C'est la meilleure manière d'écrire le code; c'est non seulement plus facile à

conceptualiser, mais également beaucoup plus facile à lire et à maintenir.