

Bordures

JComponent possède une méthode appelée **setBorder()**, qui permet de placer différentes bordures intéressantes sur tout composant visible. L'exemple suivant montre certaines des bordures existantes, en utilisant la méthode **showBorder()** qui crée un **JPanel** et lui attache une bordure à chaque fois. Il utilise aussi RTTI pour trouver le nom de la bordure qu'on utilise (en enlevant l'information du chemin), et met ensuite le nom dans un **JLabel** au milieu du panneau :

```
//: cl3:Borders.java

// Diverses bordures Swing.
// <applet code=Borders
// width=500 height=300></applet>

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

public class Borders extends JApplet {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER),
            BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }

    public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.setLayout(new GridLayout(2,4));
        cp.add(showBorder(new TitledBorder("Title")));
        cp.add(showBorder(new EtchedBorder()));
        cp.add(showBorder(new LineBorder(Color.blue)));
        cp.add(showBorder(
```

```

        new MatteBorder(5,5,30,30,Color.green)));
cp.add(showBorder(
    new BevelBorder(BevelBorder.RAISED)));
cp.add(showBorder(
    new SoftBevelBorder(BevelBorder.LOWERED)));
cp.add(showBorder(new CompoundBorder(
    new EtchedBorder(),
    new LineBorder(Color.red))));
}

public static void main(String[] args) {
    Console.run(new Borders(), 500, 300);
}
} ///:~

```

On peut également créer ses propres bordures et les placer dans des boutons, labels, et cetera, tout ce qui est dérivé de JComponent.

JScrollPane

La plupart du temps on laissera le **JScrollPane** tel quel, mais on peut aussi contrôler quelles barres de défilement sont autorisées, verticales, horizontales, les deux ou ni l'une ni l'autre :

```

///: c13:JScrollPane.java
// Contrôle des scrollbars d'un JScrollPane.
// <applet code=JScrollPane width=300 height=725>
// </applet>

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

public class JScrollPane extends JApplet {
    JButton
        b1 = new JButton("Text Area 1"),
        b2 = new JButton("Text Area 2"),
        b3 = new JButton("Replace Text"),

```

```

    b4 = new JButton("Insert Text");

JTextArea

    t1 = new JTextArea("t1", 1, 20),
    t2 = new JTextArea("t2", 4, 20),
    t3 = new JTextArea("t3", 1, 20),
    t4 = new JTextArea("t4", 10, 10),
    t5 = new JTextArea("t5", 4, 20),
    t6 = new JTextArea("t6", 10, 10);

JScrollPane

    sp3 = new JScrollPane(t3,
        JScrollPane.VERTICAL_SCROLLBAR_NEVER,
        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp4 = new JScrollPane(t4,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_NEVER),
    sp5 = new JScrollPane(t5,
        JScrollPane.VERTICAL_SCROLLBAR_NEVER,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS),
    sp6 = new JScrollPane(t6,
        JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
        JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);

class B1L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t5.append(t1.getText() + "\n");
    }
}

class B2L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t2.setText("Inserted by Button 2");
        t2.append(": " + t1.getText());
        t5.append(t2.getText() + "\n");
    }
}

class B3L implements ActionListener {

```

```

    public void actionPerformed(ActionEvent e) {
        String s = " Replacement ";
        t2.replaceRange(s, 3, 3 + s.length());
    }
}

class B4L implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t2.insert(" Inserted ", 10);
    }
}

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());

    // Création de Borders pour les composants:
    Border brd = BorderFactory.createMatteBorder(
        1, 1, 1, 1, Color.black);
    t1.setBorder(brd);
    t2.setBorder(brd);
    sp3.setBorder(brd);
    sp4.setBorder(brd);
    sp5.setBorder(brd);
    sp6.setBorder(brd);

    // Initialisation des listeners et ajout des composants:
    b1.addActionListener(new B1L());
    cp.add(b1);
    cp.add(t1);
    b2.addActionListener(new B2L());
    cp.add(b2);
    cp.add(t2);
    b3.addActionListener(new B3L());
    cp.add(b3);
    b4.addActionListener(new B4L());
    cp.add(b4);
    cp.add(sp3);

```

```

        cp.add(sp4);

        cp.add(sp5);

        cp.add(sp6);
    }

    public static void main(String[] args) {
        Console.run(new JScrollPanels(), 300, 725);
    }
} ///:~

```

L'utilisation des différents arguments du constructeur de **JScrollPane** contrôle la présence des scrollbars. Cet exemple est également un peu "habillé" à l'aide de bordures.

Un mini-éditeur

Le contrôle **JTextPane** fournit un support important pour l'édition de texte, sans grand effort. L'exemple suivant en fait une utilisation très simple, en ignorant le plus gros des fonctionnalités de la classe :

```

///: c13:TextPane.java

// Le contrôle JTextPane est un petit éditeur de texte.
// <applet code=TextPane width=475 height=425>
// </applet>

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

public class TextPane extends JApplet {
    JButton b = new JButton("Add Text");
    JTextPane tp = new JTextPane();

    static Generator sg =
        new Arrays2.RandStringGenerator(7);

    public void init() {
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                for(int i = 1; i < 10; i++)
                    tp.setText(tp.getText() +
                        sg.next() + "\n");
            }
        });
    }
}

```

```

    }

    });

    Container cp = getContentPane();

    cp.add(new JScrollPane(tp));

    cp.add(BorderLayout.SOUTH, b);

}

public static void main(String[] args) {

    Console.run(new TextPane(), 475, 425);

}

} ///:~

```

Le bouton ajoute simplement au hasard du texte généré. Le but du **JTextPane** est de permettre la modification de texte sur place, de sorte qu'on ne trouvera pas de méthode **append()**. Dans ce cas-ci (il est admis qu'il s'agit d'un piètre usage des capacités de **JTextPane**), le texte doit être saisi, modifié, et remplacé dans le panneau en utilisant **setText()**.

Comme mentionné auparavant, le comportement par défaut d'une applet est d'utiliser le **BorderLayout**. Si on ajoute quelque chose au panneau sans spécifier plus de détails, il remplit simplement le centre jusqu'aux bords. Cependant, si on spécifie une des régions des bords (NORTH, SOUTH, EAST, ou WEST) comme nous le faisons ici, le composant s'insérera de lui-même dans cette région. Dans notre cas, le bouton va s'installer au fond de l'écran.

Remarquez les fonctionnalités intrinsèques du **JTextPane**, telles que le retour à la ligne automatique. Il y a de nombreuses autres fonctionnalités à découvrir dans la documentation du JDK.

Boîtes à cocher [*Check boxes*]

Une boîte à cocher [*check box*] permet d'effectuer un choix simple vrai/faux ; il consiste en une petite boîte et un label. La boîte contient d'habitude un petit x (ou tout autre moyen d'indiquer qu'elle est cochée) ou est vide, selon qu'elle a été sélectionnée ou non.

On crée normalement une **JCheckBox** en utilisant un constructeur qui prend le label comme argument. On peut obtenir ou forcer l'état, et également obtenir ou forcer le label si on veut le lire ou le modifier après la création de la **JCheckBox**.

Chaque fois qu'une **JCheckBox** est remplie ou vidée, un événement est généré, qu'on peut capturer de la même façon que pour un bouton, en utilisant un **ActionListener**. L'exemple suivant utilise un **JTextArea** pour lister toutes les boîtes à cocher qui ont été cochées :

```

///: c13:CheckBoxes.java

// Utilisation des JCheckBoxes.

// <applet code=CheckBoxes width=200 height=200>

// </applet>

import javax.swing.*;

```

```

import java.awt.event.*;

import java.awt.*;

import com.bruceeckel.swing.*;

public class CheckBoxes extends JApplet {

    JTextArea t = new JTextArea(6, 15);

    JCheckBox

        cb1 = new JCheckBox("Check Box 1"),
        cb2 = new JCheckBox("Check Box 2"),
        cb3 = new JCheckBox("Check Box 3");

    public void init() {

        cb1.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e){

                trace("1", cb1);

            }

        });

        cb2.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e){

                trace("2", cb2);

            }

        });

        cb3.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e){

                trace("3", cb3);

            }

        });

        Container cp = getContentPane();

        cp.setLayout(new FlowLayout());

        cp.add(new JScrollPane(t));

        cp.add(cb1);

        cp.add(cb2);

        cp.add(cb3);

    }

    void trace(String b, JCheckBox cb) {

```

```

        if(cb.isSelected())
            t.append("Box " + b + " Set\n");
        else
            t.append("Box " + b + " Cleared\n");
    }

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

```

La méthode **trace()** envoie le nom et l'état de la **JCheckBox** sélectionnée au **JTextArea** en utilisant **append()**, de telle sorte qu'on voit une liste cumulative des boîtes à cocher qui ont été sélectionnées, et quel est leur état.

Boutons radio

Le concept d'un bouton radio en programmation de GUI provient des autoradios d'avant l'ère électronique, avec des boutons mécaniques : quand on appuie sur l'un d'eux, tout autre bouton enfoncé est relâché. Ceci permet de forcer un choix unique parmi plusieurs.

Pour installer un groupe de **JRadioButtons**, il suffit de les ajouter à un **ButtonGroup** (il peut y avoir un nombre quelconque de **ButtonGroups** dans un formulaire). En utilisant le second argument du constructeur, on peut optionnellement forcer à **true** l'état d'un des boutons. Si on essaie de forcer à **true** plus d'un bouton radio, seul le dernier forcé sera à **true**.

Voici un exemple simple d'utilisation de boutons radio. On remarquera que les événement des boutons radio s'interceptent comme tous les autres :

```

/// c13:RadioButtons.java
// Utilisation des JRadioButtons.
// <applet code=RadioButtons
// width=200 height=100> </applet>
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
import com.bruceeckel.swing.*;

public class RadioButtons extends JApplet {
    JTextField t = new JTextField(15);

    ButtonGroup g = new ButtonGroup();

    JRadioButton

    rb1 = new JRadioButton("one", false),

```



```

        rb2 = new JRadioButton("two", false),
        rb3 = new JRadioButton("three", false);
    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            t.setText("Radio button " +
                ((JRadioButton)e.getSource()).getText());
        }
    };
    public void init() {
        rb1.addActionListener(al);
        rb2.addActionListener(al);
        rb3.addActionListener(al);
        g.add(rb1); g.add(rb2); g.add(rb3);
        t.setEditable(false);
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        cp.add(rb1);
        cp.add(rb2);
        cp.add(rb3);
    }
    public static void main(String[] args) {
        Console.run(new RadioButtons(), 200, 100);
    }
} ///:~

```

Pour afficher l'état un champ texte est utilisé. Ce champ est déclaré non modifiable car il est utilisé uniquement pour afficher des données et pas pour en recevoir. C'est une alternative à l'utilisation d'un **JLabel**.

Boîtes combo (listes à ouverture vers le bas) [*combo boxes (drop-down lists)*]

Comme les groupes de boutons radio, une *drop-down list* est une façon de forcer l'utilisateur à choisir un seul élément parmi un groupe de possibilités. C'est cependant un moyen plus compact, et il est plus facile de modifier les éléments de la liste sans surprendre l'utilisateur (on peut modifier dynamiquement les boutons radio, mais ça peut devenir visuellement perturbant).

La **JComboBox** java n'est pas comme la combo box de Windows qui permet de sélectionner dans une liste *ou*

de taper soi-même une sélection. Avec une **JComboBox** on choisit un et un seul élément de la liste. Dans l'exemple suivant, la **JComboBox** démarre avec un certain nombre d'éléments et ensuite des éléments sont ajoutés lorsqu'on appuie sur un bouton.

```

//: c13:ComboBoxes.java

// Utilisation des drop-down lists.

// <applet code=ComboBoxes

// width=200 height=100> </applet>

import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

import com.bruceeckel.swing.*;

public class ComboBoxes extends JApplet {

    String[] description = { "Ebullient", "Obtuse",

        "Recalcitrant", "Brilliant", "Somnescent",

        "Timorous", "Florid", "Putrescent" };

    JTextField t = new JTextField(15);

    JComboBox c = new JComboBox();

    JButton b = new JButton("Add items");

    int count = 0;

    public void init() {

        for(int i = 0; i < 4; i++)

            c.addItem(description[count++]);

        t.setEditable(false);

        b.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e){

                if(count < description.length)

                    c.addItem(description[count++]);

            }

        });

        c.addActionListener(new ActionListener() {

            public void actionPerformed(ActionEvent e){

                t.setText("index: " + c.getSelectedIndex()

                    + "    " + ((JComboBox)e.getSource())

```

```

        .getSelectedItem());
    }
});

Container cp = getContentPane();

cp.setLayout(new FlowLayout());

cp.add(t);

cp.add(c);

cp.add(b);

}

public static void main(String[] args) {
    Console.run(new ComboBoxes(), 200, 100);
}
} ///:~

```

Le **JTextField** affiche l'index sélectionné, qui est le numéro séquentiel de l'élément sélectionné, ainsi que le label du bouton radio.

Listes [*List boxes*]

Les listes sont différentes des **JComboBox**, et pas seulement en apparence. Alors qu'une **JComboBox** s'affiche vers le bas lorsqu'on l'active, une **JList** occupe un nombre fixe de lignes sur l'écran tout le temps et ne se modifie pas. Si on veut voir les éléments de la liste, il suffit d'appeler **getSelectedValues()**, qui retourne un tableau de **String** des éléments sélectionnés.

Une **JList** permet la sélection multiple : si on "control-clique" sur plus d'un élément (en enfonçant la touche control tout en effectuant des clics souris) l'élément initial reste surligné et on peut en sélectionner autant qu'on veut. Si on sélectionne un élément puis qu'on en "shift-clique" un autre, tous les éléments entre ces deux-là seront aussi sélectionnés. Pour supprimer un élément d'un groupe on peut le "control-cliquer".

```

///: c13:List.java

// <applet code=List width=250
// height=375> </applet>

import javax.swing.*;

import javax.swing.event.*;

import java.awt.*;

import java.awt.event.*;

import javax.swing.border.*;

import com.bruceeckel.swing.*;

public class List extends JApplet {

```

```

String[] flavors = { "Chocolate", "Strawberry",
    "Vanilla Fudge Swirl", "Mint Chip",
    "Mocha Almond Fudge", "Rum Raisin",
    "Praline Cream", "Mud Pie" };

DefaultListModel lItems=new DefaultListModel();

JList lst = new JList(lItems);

JTextArea t = new JTextArea(flavors.length,20);

JButton b = new JButton("Add Item");

ActionListener bl = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if(count < flavors.length) {
            lItems.add(0, flavors[count++]);
        } else {
            // Invalidier, puisqu'il n'y a plus
            // de parfums a ajouter a la liste
            b.setEnabled(false);
        }
    }
};

ListSelectionListener ll =
    new ListSelectionListener() {
        public void valueChanged(
            ListSelectionEvent e) {
            t.setText("");
            Object[] items=lst.getSelectedValues();
            for(int i = 0; i < items.length; i++)
                t.append(items[i] + "\n");
        }
    };

int count = 0;

public void init() {
    Container cp = getContentPane();
    t.setEditable(false);
    cp.setLayout(new FlowLayout());

```

```

// Création de Bords pour les composants:

Border brd = BorderFactory.createMatteBorder(

    1, 1, 2, 2, Color.black);

lst.setBorder(brd);

t.setBorder(brd);

// Ajout des quatre premiers élément à la liste

for(int i = 0; i < 4; i++)

    lItems.addElement(flavors[count++]);

// Ajout des éléments au Content Pane pour affichage

cp.add(t);

cp.add(lst);

cp.add(b);

// Enregistrement des listeners d'événements

lst.addListSelectionListener(ll);

b.addActionListener(bl);

}

public static void main(String[] args) {

    Console.run(new List(), 250, 375);

}

} ///:~

```

Quand on appuie sur le bouton il ajoute des élément au *début* de la liste (car le second argument de **addItem()** est 0).

On peut également voir qu'une bordure a été ajoutée aux listes.

Si on veut simplement mettre un tableau de **Strings** dans une **JList**, il y a une solution beaucoup plus simple : passer le tableau au constructeur de la **JList**, et il construit la liste automatiquement. La seule raison d'utiliser le modèle de liste de l'exemple ci-dessus est que la liste peut être manipulée lors de l'exécution du programme.

Les **JLists** ne fournissent pas de support direct pour le scrolling. Bien évidemment, il suffit d'encapsuler la **JList** dans une **JScrollPane** et tous les détails sont automatiquement gérés.

Panneaux à tabulations [*Tabbed panes*]

Les **JTabbedPane** permettent de créer un dialogue tabulé, avec sur un côté des tabulations semblables à celles de classeurs de fiches, permettant d'amener au premier plan un autre dialogue en cliquant sur l'une d'entre elles.

```

//: c13:TabbedPane1.java

// Démonstration de Tabbed Pane.

// <applet code=TabbedPane1

```

```

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

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

public class TabbedPanel extends JApplet {
    String[] flavors = { "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie" };
    JTabbedPane tabs = new JTabbedPane();
    JTextField txt = new JTextField(20);
    public void init() {
        for(int i = 0; i < flavors.length; i++)
            tabs.addTab(flavors[i],
                new JButton("Tabbed pane " + i));
        tabs.addChangeListener(new ChangeListener(){
            public void stateChanged(ChangeEvent e) {
                txt.setText("Tab selected: " +
                    tabs.getSelectedIndex());
            }
        });
        Container cp = getContentPane();
        cp.add(BorderLayout.SOUTH, txt);
        cp.add(tabs);
    }
    public static void main(String[] args) {
        Console.run(new TabbedPanel(), 350, 200);
    }
} ///:~

```

En Java, l'utilisation d'un mécanisme de panneaux à tabulations est important car, pour la programmation

d'applets, l'utilisation de dialogues *pop-ups* est découragé par l'apparition automatique d'un petit avertissement à chaque dialogue qui surgit d'une applet.

Lors de l'exécution de ce programme on remarquera que le **JTabbedPane** empile automatiquement les tabulations s'il y en a trop pour une rangée. On peut s'en apercevoir en redimensionnant la fenêtre lorsque le programme est lancé depuis la ligne de commande.

Boîtes de messages

Les environnements de fenêtrage comportent classiquement un ensemble standard de boîtes de messages qui permettent d'afficher rapidement une information à l'utilisateur, ou lui demander une information. Dans Swing, ces boîtes de messages sont contenues dans les **JOptionPane**. Il y a de nombreuses possibilités (certaines assez sophistiquées), mais celles qu'on utilise le plus couramment sont les messages et les confirmations, appelées en utilisant **static JOptionPane.showMessageDialog()** et **JOptionPane.showConfirmDialog()**. L'exemple suivant montre un sous-ensemble des boîtes de messages disponibles avec **JOptionPane** :

```

//: c13:MessageBoxes.java

// Démonstration de JOptionPane.

// <applet code=MessageBoxes

// width=200 height=150> </applet>

import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

import com.bruceeckel.swing.*;

public class MessageBoxes extends JApplet {

    JButton[] b = { new JButton("Alert"),

        new JButton("Yes/No"), new JButton("Color"),

        new JButton("Input"), new JButton("3 Vals")

    };

    JTextField txt = new JTextField(15);

    ActionListener al = new ActionListener() {

        public void actionPerformed(ActionEvent e){

            String id =

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

            if(id.equals("Alert"))

                JOptionPane.showMessageDialog(null,

                    "There's a bug on you!", "Hey!",

                    JOptionPane.ERROR_MESSAGE);
        }
    };
}

```

```

else if(id.equals("Yes/No"))

    JOptionPane.showConfirmDialog(null,

        "or no", "choose yes",

        JOptionPane.YES_NO_OPTION);
else if(id.equals("Color")) {
    Object[] options = { "Red", "Green" };
    int sel = JOptionPane.showOptionDialog(

        null, "Choose a Color!", "Warning",

        JOptionPane.DEFAULT_OPTION,

        JOptionPane.WARNING_MESSAGE, null,

        options, options[0]);

    if(sel != JOptionPane.CLOSED_OPTION)

        txt.setText(

            "Color Selected: " + options[sel]);
} else if(id.equals("Input")) {

    String val = JOptionPane.showInputDialog(

        "How many fingers do you see?");

    txt.setText(val);
} else if(id.equals("3 Vals")) {

    Object[] selections = {

        "First", "Second", "Third" };

    Object val = JOptionPane.showInputDialog(

        null, "Choose one", "Input",

        JOptionPane.INFORMATION_MESSAGE,

        null, selections, selections[0]);

    if(val != null)

        txt.setText(

            val.toString());

    }

}

};

public void init() {

    Container cp = getContentPane();

    cp.setLayout(new FlowLayout());

```



```

        for(int i = 0; i < b.length; i++) {
            b[i].addActionListener(al);
            cp.add(b[i]);
        }
        cp.add(txt);
    }

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

```

Pour pouvoir écrire un seul **ActionListener**, j'ai utilisé l'approche un peu risquée de tester les **Strings** des labels sur les boutons. Le problème de cette technique est qu'il est facile de se tromper légèrement dans les labels, classiquement dans les majuscules et minuscules, et ce bug peut être difficile à détecter.

On remarquera que **showOptionDialog()** et **showInputDialog()** retournent des objets contenant la valeur entrée par l'utilisateur.

Menus

Chaque composant capable de contenir un menu, y compris **JApplet**, **JFrame**, **JDialog** et leurs descendants, possède une méthode **setJMenuBar()** qui prend comme paramètre un **JMenuBar** (il ne peut y avoir qu'un seul **JMenuBar** sur un composant donné). On ajoute les **JMenus** au **JMenuBar**, et les **JMenuItems** aux **JMenus**. On peut attacher un **ActionListener** à chaque **JMenuItem**, qui sera lancé quand l'élément de menu est sélectionné.

Contrairement à un système qui utilise des ressources, en Java et Swing il faut assembler à la main tous les menus dans le code source. Voici un exemple très simple de menu :

```

///: c13:SimpleMenus.java

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

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

public class SimpleMenus extends JApplet {
    JTextField t = new JTextField(15);

    ActionListener al = new ActionListener() {
        public void actionPerformed(ActionEvent e){

```

```

        t.setText(
            ((JMenuItem)e.getSource()).getText());
    }
};

JMenu[] menus = { new JMenu("Winken"),
    new JMenu("Blinken"), new JMenu("Nod") };

JMenuItem[] items = {
    new JMenuItem("Fee"), new JMenuItem("Fi"),
    new JMenuItem("Fo"), new JMenuItem("Zip"),
    new JMenuItem("Zap"), new JMenuItem("Zot"),
    new JMenuItem("Olly"), new JMenuItem("Oxen"),
    new JMenuItem("Free") };

public void init() {
    for(int i = 0; i < items.length; i++) {
        items[i].addActionListener(al);
        menus[i%3].add(items[i]);
    }

    JMenuBar mb = new JMenuBar();
    for(int i = 0; i < menus.length; i++)
        mb.add(menus[i]);

    setJMenuBar(mb);

    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(t);
}

public static void main(String[] args) {
    Console.run(new SimpleMenus(), 200, 75);
}

} ///:~

```

L'utilisation de l'opérateur modulo [*modulus*] dans **i%3** distribue les éléments de menus parmi les trois **JMenus**. Chaque **JMenuItem** doit avoir un **ActionListener** attaché ; ici le même **ActionListener** est utilisé partout mais on en aura besoin normalement d'un pour chaque **JMenuItem**.

JMenuItem hérite d'**AbstractButton**, et il a donc certains comportements des boutons. En lui-même, il fournit un élément qui peut être placé dans un menu déroulant. Il y a aussi trois types qui héritent de **JMenuItem** :

JMenu pour contenir d'autres **JMenuItems** (pour réaliser des menus en cascade), **JCheckBoxMenuItem**, qui

fournit un marquage pour indiquer si l'élément de menu est sélectionné ou pas, et **JRadioButtonMenuItem**, qui contient un bouton radio.

En tant qu'exemple plus sophistiqué, voici à nouveau les parfums de crèmes glacées, utilisés pour créer des menus. Cet exemple montre également des menus en cascade, des mnémoniques clavier, des **JCheckBoxMenuItems**, et la façon de changer ces menus dynamiquement :

```

//: c13:Menus.java

// Sous-menus, éléments de menu avec boîtes à cocher, permutations de menus,
// mnémoniques (raccourcis) et commandes d'actions.

// <applet code=Menus width=300
// height=100> </applet>

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

public class Menus extends JApplet {

    String[] flavors = { "Chocolate", "Strawberry",
        "Vanilla Fudge Swirl", "Mint Chip",
        "Mocha Almond Fudge", "Rum Raisin",
        "Praline Cream", "Mud Pie" };

    JTextField t = new JTextField("No flavor", 30);

    JMenuBar mbl = new JMenuBar();

    JMenu

        f = new JMenu("File"),
        m = new JMenu("Flavors"),
        s = new JMenu("Safety");

    // Approche alternative :

    JCheckBoxMenuItem[] safety = {

        new JCheckBoxMenuItem("Guard"),
        new JCheckBoxMenuItem("Hide")
    };

    JMenuItem[] file = {

        new JMenuItem("Open"),
    };

    // Une seconde barre de menu pour échanger :

```

```

JMenuBar mb2 = new JMenuBar();
JMenu fooBar = new JMenu("fooBar");
JMenuItem[] other = {
    // Ajouter un raccourci de menu (mnémonique) est très
    // simple, mais seuls les JMenuItem peuvent les avoir
    // dans leurs constructeurs:
    new JMenuItem("Foo", KeyEvent.VK_F),
    new JMenuItem("Bar", KeyEvent.VK_A),
    // Pas de raccourci :
    new JMenuItem("Baz"),
};
JButton b = new JButton("Swap Menus");
class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuBar m = getJMenuBar();
        setJMenuBar(m == mb1 ? mb2 : mb1);
        validate(); // Rafraîchissement de la fenêtre
    }
}
class ML implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        String actionCommand =
            target.getActionCommand();
        if(actionCommand.equals("Open")) {
            String s = t.getText();
            boolean chosen = false;
            for(int i = 0; i < flavors.length; i++)
                if(s.equals(flavors[i])) chosen = true;
            if(!chosen)
                t.setText("Choose a flavor first!");
            else
                t.setText("Opening " + s + ". Mmm, mm!");
        }
    }
}

```

```

    }
}

class FL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JMenuItem target = (JMenuItem)e.getSource();
        t.setText(target.getText());
    }
}

// Alternativement, on peut créer une classe
// différente pour chaque JMenuItem. Ensuite
// il n'est plus nécessaire de rechercher de laquelle il s'agit :
class FooL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Foo selected");
    }
}

class BarL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Bar selected");
    }
}

class BazL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        t.setText("Baz selected");
    }
}

class CMIL implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        JCheckBoxMenuItem target =
            (JCheckBoxMenuItem)e.getSource();
        String actionCommand =
            target.getActionCommand();
        if(actionCommand.equals("Guard"))
            t.setText("Guard the Ice Cream! " +

```

```

        "Guarding is " + target.getState());
    else if(actionCommand.equals("Hide"))
        t.setText("Hide the Ice Cream! " +
            "Is it cold? " + target.getState());
    }
}

public void init() {
    ML ml = new ML();
    CMIL cmil = new CMIL();
    safety[0].setActionCommand("Guard");
    safety[0].setMnemonic(KeyEvent.VK_G);
    safety[0].addItemListener(cmil);
    safety[1].setActionCommand("Hide");
    safety[0].setMnemonic(KeyEvent.VK_H);
    safety[1].addItemListener(cmil);
    other[0].addActionListener(new FooL());
    other[1].addActionListener(new BarL());
    other[2].addActionListener(new BazL());
    FL fl = new FL();
    for(int i = 0; i < flavors.length; i++) {
        JMenuItem mi = new JMenuItem(flavors[i]);
        mi.addActionListener(fl);
        m.add(mi);
        // Ajout de séparateurs par intervalles:
        if((i+1) % 3 == 0)
            m.addSeparator();
    }
    for(int i = 0; i < safety.length; i++)
        s.add(safety[i]);
    s.setMnemonic(KeyEvent.VK_A);
    f.add(s);
    f.setMnemonic(KeyEvent.VK_F);
    for(int i = 0; i < file.length; i++) {
        file[i].addActionListener(fl);
    }
}

```

```

        f.add(file[i]);
    }
    mb1.add(f);
    mb1.add(m);

    setJMenuBar(mb1);

    t.setEditable(false);

    Container cp = getContentPane();
    cp.add(t, BorderLayout.CENTER);

    // Installation du système d'échange de menus:
    b.addActionListener(new BL());
    b.setMnemonic(KeyEvent.VK_S);
    cp.add(b, BorderLayout.NORTH);

    for(int i = 0; i < other.length; i++)
        fooBar.add(other[i]);

    fooBar.setMnemonic(KeyEvent.VK_B);
    mb2.add(fooBar);
}

public static void main(String[] args) {
    Console.run(new Menus(), 300, 100);
}
} ///:~

```

Dans ce programme j'ai placé les éléments de menus dans des tableaux et ensuite j'ai parcouru chaque tableau en appelant **add()** pour chaque **JMenuItem**. Ceci rend l'ajout ou la suppression d'un élément de menu un peu moins fastidieux.

Ce programme crée deux **JMenuBars** pour démontrer que les barres de menu peuvent être échangées dynamiquement à l'exécution du programme. On peut voir qu'un **JMenuBar** est composé de **JMenus**, et que chaque **JMenu** est composé de **JMenuItems**, **JCheckBoxMenuItems**, ou même d'autres **JMenus** (qui produisent des sous-menus). Une fois construit un **JMenuBar**, il peut être installé dans le programme courant avec la méthode **setJMenuBar()**. Remarquons que lorsque le bouton est cliqué, il regarde quel menu est installé en appelant **getJMenuBar()**, et à ce moment il le remplace par l'autre barre de menu.

Lorsqu'on teste "Open", il faut remarquer que l'orthographe et les majuscules/minuscules sont cruciaux, et que Java ne signale pas d'erreur s'il n'y a pas correspondance avec "Open". Ce genre de comparaison de chaînes est une source d'erreurs de programmation.

Le cochage et le décochage des éléments de menus est pris en compte automatiquement. Le code gérant les **JCheckBoxMenuItems** montre deux façons différentes de déterminer ce qui a été coché : la correspondance des chaînes (qui, comme mentionné ci-dessus, n'est pas une approche très sûre bien qu'on la rencontre) et la correspondance de l'objet cible de l'événement. Il montre aussi que la méthode **getState()** peut être utilisée pour

connaître son état. On peut également changer l'état d'un **JCheckBoxMenuItem** à l'aide de **setState()**.

Les événements des menus sont un peu inconsistants et peuvent prêter à confusion : les **JMenuItems** utilisent des **ActionListeners**, mais les **JCheckboxMenuItems** utilisent des **ItemListeners**. Les objets **JMenu** peuvent aussi supporter des **ActionListeners**, mais ce n'est généralement pas très utile. En général, on attache des *listeners* à chaque **JMenuItem**, **JCheckBoxMenuItem**, ou **JRadioButtonMenuItem**, mais l'exemple montre des **ItemListeners** et des **ActionListeners** attachés aux divers composants de menus.

Swing supporte les mnémoniques, ou raccourcis clavier, de sorte qu'on peut sélectionner tout ce qui est dérivé de **AbstractButton** (bouton, menu, élément, et cetera) en utilisant le clavier à la place de la souris. C'est assez simple : pour le **JMenuItem** on peut utiliser le constructeur surchargé qui prend en deuxième argument l'identificateur de la touche. Cependant, la plupart des **AbstractButtons** n'ont pas ce constructeur; une manière plus générale de résoudre ce problème est d'utiliser la méthode **setMnemonic()**. L'exemple ci-dessus ajoute une mnémonique au bouton et à certains des éléments de menus : les indicateurs de raccourcis apparaissent automatiquement sur les composants.

On peut aussi voir l'utilisation de **setActionCommand()**. Ceci paraît un peu bizarre car dans chaque cas la commande d'action [*action command*] est exactement la même que le label sur le composant du menu. Pourquoi ne pas utiliser simplement le label plutôt que cette chaîne de remplacement ? Le problème est l'internationalisation. Si on réoriente ce programme vers une autre langue, on désire changer uniquement le label du menu, et pas le code (ce qui introduirait à coup sûr d'autres erreurs). Donc pour faciliter ceci pour les codes qui testent la chaîne associée à un composant de menu, la commande d'action peut être invariante tandis que le label du menu peut changer. Tout le code fonctionne avec la commande d'action, de sorte qu'il n'est pas touché par les modifications des labels des menus. Remarquons que dans ce programme on ne recherche pas des commandes d'actions pour tous les composants de menus, de sorte que ceux qui ne sont pas examinés n'ont pas de commande d'action positionnée.

La plus grosse partie du travail se trouve dans les *listeners*. **BL** effectue l'échange des **JMenuBar**s. Dans **ML**, l'approche du "qui a sonné ?" est utilisée en utilisant la source de l'**ActionEvent** et en l'émettant vers un **JMenuItem**, en faisant passer la chaîne de la commande d'action à travers une instruction **if** en cascade.

Le *listener* **FL** est simple, bien qu'il gère les différents parfums du menu parfums. Cette approche est utile si la logique est simple, mais en général, on utilisera l'approche de **FooL**, **BarL** et **BazL**, dans lesquels ils sont chacun attaché à un seul composant de menu de sorte qu'il n'est pas nécessaire d'avoir une logique de détection supplémentaire, et on sait exactement qui a appelé le *listener*. Même avec la profusion de classes générées de cette façon, le code interne tend à être plus petit et le traitement est plus fiable.

On peut voir que le code d'un menu devient rapidement long et désordonné. C'est un autre cas où l'utilisation d'un *GUI builder* est la solution appropriée. Un bon outil gèrera également la maintenance des menus.

Menus pop-up

La façon la plus directe d'implémenter un **JPopupMenu** est de créer une classe interne qui étend **MouseAdapter**, puis d'ajouter un objet de cette classe interne à chaque composant pour lequel on veut créer le pop-up :

```

//: c13:Popup.java
// Création de menus popup avec Swing.
// <applet code=Popup
// width=300 height=200></applet>

```



```

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import com.bruceeckel.swing.*;

public class Popup extends JApplet {

    JPopupMenu popup = new JPopupMenu();

    JTextField t = new JTextField(10);

    public void init() {

        Container cp = getContentPane();

        cp.setLayout(new FlowLayout());

        cp.add(t);

        ActionListener al = new ActionListener() {

            public void actionPerformed(ActionEvent e){

                t.setText(

                    ((JMenuItem)e.getSource()).getText());

            }

        };

        JMenuItem m = new JMenuItem("Hither");

        m.addActionListener(al);

        popup.add(m);

        m = new JMenuItem("Yon");

        m.addActionListener(al);

        popup.add(m);

        m = new JMenuItem("Afar");

        m.addActionListener(al);

        popup.add(m);

        popup.addSeparator();

        m = new JMenuItem("Stay Here");

        m.addActionListener(al);

        popup.add(m);

        PopupListener pl = new PopupListener();

        addMouseListener(pl);

        t.addMouseListener(pl);
    }
}

```

```

    }

    class PopupListener extends MouseAdapter {

        public void mousePressed(MouseEvent e) {

            maybeShowPopup(e);

        }

        public void mouseReleased(MouseEvent e) {

            maybeShowPopup(e);

        }

        private void maybeShowPopup(MouseEvent e) {

            if(e.isPopupTrigger()) {

                popup.show(

                    e.getComponent(), e.getX(), e.getY());

            }

        }

    }

    public static void main(String[] args) {

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

    }

} ///:~

```

Le même **ActionListener** est ajouté à chaque **JMenuItem** de façon à prendre le texte du label du menu et l'insérer dans le **JTextField**.

Dessiner

Dans un bon outil de GUI, dessiner devrait être assez facile, et ça l'est dans la bibliothèque Swing. Le problème de tout exemple de dessin est que les calculs qui déterminent où vont les éléments sont souvent beaucoup plus compliqués que les appels aux sous-programmes de dessin, et que ces calculs sont souvent mélangés aux appels de dessin, de sorte que l'interface semble plus compliquée qu'elle ne l'est.

Pour simplifier, considérons le problème de la représentation de données sur l'écran. Ici, les données sont fournies par la méthode intrinsèque **Math.sin()** qui est la fonction mathématique sinus. Pour rendre les choses un peu plus intéressantes, et pour mieux montrer combien il est facile d'utiliser les composants Swing, un curseur sera placé en bas du formulaire pour contrôler dynamiquement le nombre de cycles du sinus affiché. De plus, si on redimensionne la fenêtre, on verra le sinus s'adapter de lui-même à la nouvelle taille de la fenêtre.

Bien que tout **JComponent** puisse être peint et donc utilisé en tant que canevas, si on désire la surface de dessin la plus simple, on hérite habituellement d'un **JPanel**. La seule méthode qu'on doit redéfinir est **paintComponent()**, qui est appelée chaque fois que le composant doit être redessiné (on ne doit normalement pas s'occuper de ceci, la décision étant prise par Swing). Lorsque cette méthode est appelée, Swing lui passe un objet **Graphics**, et on peut alors utiliser cet objet pour dessiner ou peindre sur la surface.

Dans l'exemple suivant, toute l'intelligence concernant le dessin est contenue dans la classe **SineDraw**; la classe **SineWave** configure simplement le programme et le curseur. Dans **SineDraw**, la méthode **setCycles()** fournit un moyen pour permettre à un autre objet (le curseur dans ce cas) de contrôler le nombre de cycles.

```

//: c13:SineWave.java

// Dessin avec Swing, en utilisant un JSlider.
// <applet code=SineWave
//   width=700 height=400></applet>

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

class SineDraw extends JPanel {
    static final int SCALEFACTOR = 200;

    int cycles;

    int points;

    double[] sines;

    int[] pts;

    SineDraw() { setCycles(5); }

    public void setCycles(int newCycles) {
        cycles = newCycles;

        points = SCALEFACTOR * cycles * 2;

        sines = new double[points];

        pts = new int[points];

        for(int i = 0; i < points; i++) {
            double radians = (Math.PI/SCALEFACTOR) * i;

            sines[i] = Math.sin(radians);
        }

        repaint();
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        int maxWidth = getWidth();

```

```

double hstep = (double)maxWidth/(double)points;
int maxHeight = getHeight();
for(int i = 0; i < points; i++)
    pts[i] = (int)(sines[i] * maxHeight/2 * .95
                  + maxHeight/2);
g.setColor(Color.red);
for(int i = 1; i < points; i++) {
    int x1 = (int)((i - 1) * hstep);
    int x2 = (int)(i * hstep);
    int y1 = pts[i-1];
    int y2 = pts[i];
    g.drawLine(x1, y1, x2, y2);
}
}
}

```

```

public class SineWave extends JApplet {
    SineDraw sines = new SineDraw();
    JSlider cycles = new JSlider(1, 30, 5);
    public void init() {
        Container cp = getContentPane();
        cp.add(sines);
        cycles.addChangeListener(new ChangeListener(){
            public void stateChanged(ChangeEvent e) {
                sines.setCycles(
                    ((JSlider)e.getSource()).getValue());
            }
        });
        cp.add(BorderLayout.SOUTH, cycles);
    }
    public static void main(String[] args) {
        Console.run(new SineWave(), 700, 400);
    }
} ///:~

```

Tous les membres de données et tableaux sont utilisés dans le calcul des points du sinus : **cycles** indique le nombre de périodes complètes de sinus désiré, **points** contient le nombre total de points qui sera tracé, **sines** contient les valeurs de la fonction sinus, et **pts** contient les coordonnées y des points qui seront tracés sur le **JPanel**. La méthode **setCycles()** construit le tableau selon le nombre de points nécessaires et remplit le tableau **sines** de valeurs. En appelant **repaint()**, **setCycles** force l'appel de **paintComponent()**, afin que le reste des calculs et le dessin aient lieu.

La première chose à faire lorsqu'on redéfinit **paintComponent()** est d'appeler la version de base de la méthode. Ensuite on peut faire ce que l'on veut; normalement cela signifie utiliser les méthodes de **Graphics** qu'on peut trouver dans la documentation de **java.awt.Graphics** (dans la documentation HTML de *java.sun.com*) pour dessiner et peindre des pixels sur le **JPanel**. On peut voir ici que la plupart du code concerne l'exécution des calculs, les deux seules méthodes qui manipulent effectivement l'écran sont **setColor()** et **drawLine()**. Vous aurez probablement la même sensation lorsque vous créerez votre propre programme d'affichage de données graphiques : vous passerez la plus grande partie du temps à déterminer ce qu'il faut dessiner, mais le dessin en lui-même sera assez simple.

Lorsque j'ai créé ce programme, j'ai passé le plus gros de mon temps à obtenir la courbe du sinus à afficher. Ceci fait, j'ai pensé que ce serait bien de pouvoir modifier dynamiquement le nombre de cycles. Mes expériences de programmation de ce genre de choses dans d'autres langages me rendaient un peu réticent, mais cette partie se révéla la partie la plus facile du projet. J'ai créé un **JSlider** (les arguments sont respectivement la valeur de gauche du **JSlider**, la valeur de droite, et la valeur initiale, mais il existe d'autres constructeurs) et je l'ai déposé dans le **JApplet**. Ensuite j'ai regardé dans la documentation HTML et j'ai vu que le seul *listener* était le **addChangeListener**, qui était déclenché chaque fois que le curseur était déplacé suffisamment pour produire une nouvelle valeur. La seule méthode pour cela était évidemment appelée **stateChanged()**, qui fournit un objet **ChangeEvent**, de manière à pouvoir rechercher la source de la modification et obtenir la nouvelle valeur. En appelant **setCycles()** des objets **sines**, la nouvelle valeur est prise en compte et le **JPanel** est redessiné.

En général, on verra que la plupart des problèmes Swing peuvent être résolus en suivant un processus semblable, et on verra qu'il est en général assez simple, même si on n'a pas utilisé auparavant un composant donné.

Si le problème est plus compliqué, il y a d'autres solutions plus sophistiquées, par exemple les composants JavaBeans de fournisseurs tiers, et l'API Java 2D. Ces solutions sortent du cadre de ce livre, mais vous devriez les prendre en considération si votre code de dessin devient trop coûteux.

Boîtes de dialogue

Une boîte de dialogue est une fenêtre qui est issue d'une autre fenêtre. Son but est de traiter un problème spécifique sans encombrer la fenêtre d'origine avec ces détails. Les boîtes de dialogue sont fortement utilisées dans les environnements de programmation à fenêtres, mais moins fréquemment utilisées dans les applets.

Pour créer une boîte de dialogue, il faut hériter de **JDialog**, qui est simplement une sorte de **Window**, comme les **JFrames**. Un **JDialog** possède un *layout manager* (qui est par défaut le **BorderLayout**) auquel on ajoute des *listeners* d'événements pour traiter ceux-ci. Il y a une différence importante : on ne veut pas fermer l'application lors de l'appel de **windowClosing()**. Au lieu de cela, on libère les ressources utilisées par la fenêtre de dialogue en appelant **dispose()**. Voici un exemple très simple :

```
///  
// Création et utilisation de boîtes de dialogue.  
// <applet code=Dialogs width=125 height=75>
```

```

// </applet>

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

class MyDialog extends JDialog {
    public MyDialog(JFrame parent) {
        super(parent, "My dialog", true);

        Container cp = getContentPane();

        cp.setLayout(new FlowLayout());

        cp.add(new JLabel("Here is my dialog"));

        JButton ok = new JButton("OK");

        ok.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                dispose(); // Ferme le dialogue
            }
        });

        cp.add(ok);

        setSize(150,125);
    }
}

public class Dialogs extends JApplet {
    JButton b1 = new JButton("Dialog Box");

    MyDialog dlg = new MyDialog(null);

    public void init() {
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e){
                dlg.show();
            }
        });

        getContentPane().add(b1);
    }
}

```

```

    public static void main(String[] args) {
        Console.run(new Dialogs(), 125, 75);
    }
} ///:~

```

Une fois le `JDialog` créé, la méthode **show()** doit être appelée pour l'afficher et l'activer. Pour que le dialogue se ferme, il faut appeler **dispose()**.

On remarquera que tout ce qui sort d'une applet, y compris les boîtes de dialogue, n'est pas digne de confiance. C'est à dire qu'on obtient un avertissement dans la fenêtre qui apparaît. Ceci est dû au fait qu'en théorie il serait possible de tromper l'utilisateur et lui faire croire qu'il a à faire avec une de ses applications normales et de le faire taper son numéro de carte de crédit, qui partirait alors sur le Web. Une applet est toujours attachée à une page Web et visible dans un navigateur, tandis qu'une boîte de dialogue est détachée, et tout ceci est donc possible en théorie. Le résultat est qu'il n'est pas fréquent de voir une applet qui utilise une boîte de dialogue.

L'exemple suivant est plus complexe; la boîte de dialogue est composée d'une grille (en utilisant **GridLayout**) d'un type de bouton particulier qui est défini ici comme la classe **ToeButton**. Ce bouton dessine un cadre autour de lui et, selon son état, un blanc, un x ou un o au milieu. Il démarre en blanc, et ensuite, selon à qui c'est le tour, se modifie en x ou en o. Cependant, il transformera le x en o et vice versa lorsqu'on clique sur le bouton (ceci rend le principe du tic-tac-toe seulement un peu plus ennuyeux qu'il ne l'est déjà). De plus, la boîte de dialogue peut être définie avec un nombre quelconque de rangées et de colonnes dans la fenêtre principale de l'application.

```

///: c13:TicTacToe.java
// Démonstration de boîtes de dialogue
// et création de vos propres composants.
// <applet code=TicTacToe
//   width=200 height=100></applet>

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

public class TicTacToe extends JApplet {
    JTextField
        rows = new JTextField("3"),
        cols = new JTextField("3");

    static final int BLANK = 0, XX = 1, OO = 2;

    class ToeDialog extends JDialog {
        int turn = XX; // Démarre avec x a jouer

        // w = nombre de cellules en largeur
    }
}

```

```

// h = nombre de cellules en hauteur
public ToeDialog(int w, int h) {
    setTitle("The game itself");
    Container cp = getContentPane();
    cp.setLayout(new GridLayout(w, h));
    for(int i = 0; i < w * h; i++)
        cp.add(new ToeButton());
    setSize(w * 50, h * 50);
    // fermeture du dialogue en JDK 1.3 :
    // #setDefaultCloseOperation(
    // #    DISPOSE_ON_CLOSE);
    // fermeture du dialogue en JDK 1.2 :
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e){
            dispose();
        }
    });
}

class ToeButton extends JPanel {
    int state = BLANK;

    public ToeButton() {
        addMouseListener(new ML());
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        int x1 = 0;
        int y1 = 0;
        int x2 = getSize().width - 1;
        int y2 = getSize().height - 1;
        g.drawRect(x1, y1, x2, y2);
        x1 = x2/4;
        y1 = y2/4;
        int wide = x2/2;
        int high = y2/2;
    }
}

```



```

        if(state == XX) {
            g.drawLine(x1, y1,
                x1 + wide, y1 + high);
            g.drawLine(x1, y1 + high,
                x1 + wide, y1);
        }
        if(state == OO) {
            g.drawOval(x1, y1,
                x1 + wide/2, y1 + high/2);
        }
    }
}

class ML extends MouseAdapter {
    public void mousePressed(MouseEvent e) {
        if(state == BLANK) {
            state = turn;
            turn = (turn == XX ? OO : XX);
        }
        else
            state = (state == XX ? OO : XX);
        repaint();
    }
}

}

class BL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JDialog d = new ToeDialog(
            Integer.parseInt(rows.getText()),
            Integer.parseInt(cols.getText()));
        d.setVisible(true);
    }
}

public void init() {
    JPanel p = new JPanel();

```

```

        p.setLayout(new GridLayout(2,2));
        p.add(new JLabel("Rows", JLabel.CENTER));
        p.add(rows);
        p.add(new JLabel("Columns", JLabel.CENTER));
        p.add(cols);

        Container cp = getContentPane();
        cp.add(p, BorderLayout.NORTH);

        JButton b = new JButton("go");
        b.addActionListener(new BL());
        cp.add(b, BorderLayout.SOUTH);
    }

    public static void main(String[] args) {
        Console.run(new TicTacToe(), 200, 100);
    }
} ///:~

```

Comme les **statics** peuvent être uniquement au niveau le plus extérieur de la classe, les classes internes ne peuvent pas avoir de données static ni de classes internes **static**.