

On voit que chaque type de composant ne supporte que certains types d'événements. Il semble assez difficile de rechercher tous les événements supportés par chaque composant. Une approche plus simple consiste à modifier le programme **ShowMethodsClean.java** du chapitre 12 de manière à ce qu'il affiche tous les *event listeners* supportés par tout composant Swing entré.

Le chapitre 12 a introduit la *réflexion* et a utilisé cette fonctionnalité pour rechercher les méthodes d'une classe donnée, soit une liste complète des méthodes, soit un sous-ensemble des méthodes dont le nom contient un mot-clé donné. La magie dans ceci est qu'il peut automatiquement nous montrer *toutes* les méthodes d'une classe sans qu'on soit obligé de parcourir la hiérarchie des héritages en examinant les classes de base à chaque niveau. De ce fait, il fournit un outil précieux permettant de gagner du temps pour la programmation : comme les noms de la plupart des méthodes Java sont parlants et descriptifs, on peut rechercher les noms de méthodes contenant un mot particulier. Lorsqu'on pense avoir trouvé ce qu'on cherchait, il faut alors vérifier la documentation en ligne.

Comme dans le chapitre 12 on n'avait pas encore vu Swing, l'outil de ce chapitre était une application de ligne de commande. En voici une version plus pratique avec interface graphique, spécialisée dans la recherche des méthodes `addListener` dans les composants Swing :

```

//: c13:ShowAddListeners.java

// Affiche les methodes "addXXXListener"
// d'une classe Swing donnee.

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

import javax.swing.*;

import javax.swing.event.*;

import java.awt.*;

import java.awt.event.*;

import java.lang.reflect.*;

import java.io.*;

import com.bruceeckel.swing.*;

import com.bruceeckel.util.*;

public class ShowAddListeners extends JApplet {

    Class cl;

    Method[] m;

    Constructor[] ctor;

    String[] n = new String[0];

    JTextField name = new JTextField(25);

    JTextArea results = new JTextArea(40, 65);

    class NameL implements ActionListener {

```

```

public void actionPerformed(ActionEvent e) {

    String nm = name.getText().trim();

    if(nm.length() == 0) {

        results.setText("No match");

        n = new String[0];

        return;

    }

    try {

        cl = Class.forName("javax.swing." + nm);

    } catch(ClassNotFoundException ex) {

        results.setText("No match");

        return;

    }

    m = cl.getMethods();

    // Conversion en un tableau de Strings :

    n = new String[m.length];

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

        n[i] = m[i].toString();

    reDisplay();

}

void reDisplay() {

    // Creation de l'ensemble des resultats :

    String[] rs = new String[n.length];

    int j = 0;

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

        if(n[i].indexOf("add") != -1 &&

            n[i].indexOf("Listener") != -1)

            rs[j++] =

                n[i].substring(n[i].indexOf("add"));

    results.setText("");

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

        results.append(

            StripQualifiers.strip(rs[i]) + "\n");

```

```

    }

    public void init() {

        name.addActionListener(new NameL());

        JPanel top = new JPanel();

        top.add(new JLabel(

            "Swing class name (press ENTER):"));

        top.add(name);

        Container cp = getContentPane();

        cp.add(BorderLayout.NORTH, top);

        cp.add(new JScrollPane(results));

    }

    public static void main(String[] args) {

        Console.run(new ShowAddListeners(), 500,400);

    }

} ///:~

```

La classe **StripQualifiers** définie au chapitre 12 est réutilisée ici en important la bibliothèque **com.bruceeckel.util**.

L'interface utilisateur graphique contient un **JTextField name** dans lequel on saisit le nom de la classe Swing à rechercher. Les résultats sont affichés dans une **JTextArea**.

On remarquera qu'il n'y a pas de boutons ou autres composants pour indiquer qu'on désire lancer la recherche. C'est parce que le **JTextField** est surveillé par un **ActionListener**. Lorsqu'on y fait un changement suivi de ENTER, la liste est immédiatement mise à jour. Si le texte n'est pas vide, il est utilisé dans **Class.forName()** pour rechercher la classe. Si le nom est incorrect, **Class.forName()** va échouer, c'est à dire qu'il va émettre une exception. Celle-ci est interceptée et le **JTextArea** est positionné à "No match". Mais si on tape un nom correct (les majuscules/minuscules comptent), **Class.forName()** réussit et **getMethods()** retourne un tableau d'objets **Method**. Chacun des objets du tableau est transformé en **String** à l'aide de **toString()** (cette méthode fournit la signature complète de la méthode) et ajoutée à **n**, un tableau de **Strings**. Le tableau **n** est un membre de la classe **ShowAddListeners** et est utilisé pour mettre à jour l'affichage chaque fois que **reDisplay()** est appelé.

reDisplay() crée un tableau de **Strings** appelé **rs** (pour "result set" : ensemble de résultats). L'ensemble des résultats est conditionnellement copié depuis les **Strings** de **n** qui contiennent add et Listener. **indexOf()** et **substring()** sont ensuite utilisés pour enlever les qualificatifs tels que public, static, etc. Enfin, **StripQualifiers.strip()** enlève les qualificatifs de noms.

Ce programme est une façon pratique de rechercher les capacités d'un composant Swing. Une fois connus les événements supportés par un composant donné, il n'y a pas besoin de rechercher autre chose pour réagir à cet événement. Il suffit de :

1. Prendre le nom de la classe événement et retirer le mot **Event**. Ajouter le mot **Listener** à ce qui reste. Ceci donne le nom de l'interface *listener* qu'on doit implémenter dans une classe interne.
2. Implémenter l'interface ci-dessus et écrire les méthodes pour les événements qu'on veut intercepter. Par exemple, on peut rechercher les événements de déplacement de la souris, et on écrit donc le code pour la

méthode `mouseMoved()` de l'interface **MouseMotionListener** (il faut également implémenter les autres méthodes, bien sûr, mais il y a souvent un raccourci que nous verrons bientôt).

3. Créer un objet de la classe listener de l'étape 2. L'enregistrer avec le composant avec la méthode dont le nom est fourni en ajoutant **add** au début du nom du *listener*. Par exemple, **addMouseMotionListener()**.

Voici quelques-unes des interfaces *listeners* :

Interface <i>listener</i> et <i>adapter</i>	Méthodes de l'interface
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Ce n'est pas une liste exhaustive, en partie du fait que le modèle d'événements nous permet de créer nos propres types d'événements et *listeners* associés. De ce fait, on rencontrera souvent des bibliothèques qui ont inventé leurs propres événements, et la connaissance acquise dans ce chapitre nous permettra de comprendre l'utilisation de ces événements.

Utilisation de *listener adapters* pour simplifier

Dans le tableau ci-dessus, on peut voir que certaines interfaces *listener* ne possèdent qu'une seule méthode. Celles-ci sont triviales à implémenter puisqu'on ne les implémentera que lorsqu'on désire écrire cette méthode particulière. Par contre, les interfaces *listener* qui ont plusieurs méthodes peuvent être moins agréables à utiliser.

par exemple, quelque chose qu'il faut toujours faire en créant une application est de fournir un **WindowListener** au **JFrame** de manière à pouvoir appeler **System.exit()** pour sortir de l'application lorsqu'on reçoit l'événement **windowClosing()**. Mais comme **WindowListener** est une **interface**, il faut implémenter chacune de ses méthodes même si elles ne font rien. Cela peut être ennuyeux.

Pour résoudre le problème, certaines (mais pas toutes) des interfaces *listener* qui ont plus d'une méthode possèdent des adaptateurs [*adapters*], dont vous pouvez voir les noms dans le tableau ci-dessus. Chaque adaptateur fournit des méthodes vides par défaut pour chacune des méthodes de l'interface. Ensuite il suffit d'hériter de cet adaptateur et de redéfinir uniquement les méthodes qu'on doit modifier. Par exemple, le **WindowListener** qu'on utilisera normalement ressemble à ceci (souvenez-vous qu'il a été encapsulé dans la classe **Console** de **com.bruceeckel.swing**) :

```
class MyWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Le seul but des adaptateurs est de faciliter la création des classes *listener*.

Il y a cependant un désavantage lié aux adaptateurs, sous la forme d'un piège. Supposons qu'on écrive un **WindowAdapter** comme celui ci-dessus :

```
class MyWindowListener extends WindowAdapter {
    public void WindowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Ceci ne marche pas, mais il nous rendra fous à comprendre pourquoi, car tout va compiler et s'exécuter correctement, sauf que la fermeture de la fenêtre ne fera pas sortir du programme. Voyez-vous le problème ? Il est situé dans le nom de la méthode : **WindowClosing()** au lieu de **windowClosing()**. Une simple erreur de majuscule se traduit par l'ajout d'une méthode nouvelle. Ce n'est cependant pas cette méthode qui est appelée lorsque la fenêtre est fermée, de sorte qu'on n'obtient pas le résultat attendu. En dépit de cet inconvénient, une interface garantit que les méthodes sont correctement implémentées.

Surveiller plusieurs événements

Pour nous prouver que ces événements sont bien déclenchés, et en tant qu'expérience intéressante, créons une applet qui surveille les autres comportement d'un **JButton**, autres que le simple fait qu'il soit appuyé ou pas. Cet exemple montre également comment hériter de notre propre objet bouton, car c'est ce qui est utilisé comme cible de tous les événements intéressants. Pour cela, il suffit d'hériter de **JButton** [\[69\]](#).

La classe **MyButton** est une classe interne de **TrackEvent**, de sorte que **MyButton** peut aller dans la fenêtre parent et manipuler ses champs textes, ce qu'il faut pour pouvoir écrire une information d'état dans les champs du parent. Bien sûr ceci est une solution limitée, puisque **MyButton** peut être utilisée uniquement avec **TrackEvent**. Ce genre de code est parfois appelé "fortement couplé" :

```

//: c13:TrackEvent.java

// Montre les evenements lorsqu'ils arrivent.
// <applet code=TrackEvent
//   width=700 height=500></applet>

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import java.util.*;

import com.bruceeckel.swing.*;

public class TrackEvent extends JApplet {

    HashMap h = new HashMap();

    String[] event = {

        "focusGained", "focusLost", "keyPressed",

        "keyReleased", "keyTyped", "mouseClicked",

        "mouseEntered", "mouseExited", "mousePressed",

        "mouseReleased", "mouseDragged", "mouseMoved"

    };

    MyButton

    b1 = new MyButton(Color.blue, "test1"),

    b2 = new MyButton(Color.red, "test2");

    class MyButton extends JButton {

        void report(String field, String msg) {

            ((JTextField)h.get(field)).setText(msg);

        }

        FocusListener fl = new FocusListener() {

            public void focusGained(FocusEvent e) {

                report("focusGained", e paramString());

            }

            public void focusLost(FocusEvent e) {

                report("focusLost", e paramString());

            }

        };

        KeyListener kl = new KeyListener() {

```

```

    public void keyPressed(KeyEvent e) {
        report("keyPressed", e paramString());
    }

    public void keyReleased(KeyEvent e) {
        report("keyReleased", e paramString());
    }

    public void keyTyped(KeyEvent e) {
        report("keyTyped", e paramString());
    }
};

MouseListener ml = new MouseListener() {
    public void mouseClicked(MouseEvent e) {
        report("mouseClicked", e paramString());
    }

    public void mouseEntered(MouseEvent e) {
        report("mouseEntered", e paramString());
    }

    public void mouseExited(MouseEvent e) {
        report("mouseExited", e paramString());
    }

    public void mousePressed(MouseEvent e) {
        report("mousePressed", e paramString());
    }

    public void mouseReleased(MouseEvent e) {
        report("mouseReleased", e paramString());
    }
};

MouseMotionListener mml =
    new MouseMotionListener() {
        public void mouseDragged(MouseEvent e) {
            report("mouseDragged", e paramString());
        }

        public void mouseMoved(MouseEvent e) {
            report("mouseMoved", e paramString());
        }
    }
};

```

```

    }
};

public MyButton(Color color, String label) {
    super(label);
    setBackground(color);
    addFocusListener(fl);
    addKeyListener(kl);
    addMouseListener(ml);
    addMouseMotionListener(mml);
}
}

public void init() {
    Container c = getContentPane();
    c.setLayout(new GridLayout(event.length+1,2));
    for(int i = 0; i < event.length; i++) {
        JTextField t = new JTextField();
        t.setEditable(false);
        c.add(new JLabel(event[i], JLabel.RIGHT));
        c.add(t);
        h.put(event[i], t);
    }
    c.add(b1);
    c.add(b2);
}

public static void main(String[] args) {
    Console.run(new TrackEvent(), 700, 500);
}

} ///:~

```

Dans le constructeur de **MyButton**, la couleur des boutons est positionnée par un appel à **SetBackground()**. Les *listeners* sont tous installés par de simples appels de méthodes.

La classe **TrackEvent** contient une **HashMap** pour contenir les chaînes représentant le type d'événement et les **JTextFields** dans lesquels l'information sur cet événement est conservée. Bien sûr, ceux-ci auraient pu être créés en statique plutôt qu'en les mettant dans une **HashMap**, mais je pense que vous serez d'accord que c'est beaucoup plus facile à utiliser et modifier. En particulier, si on a besoin d'ajouter ou supprimer un nouveau type d'événement dans **TrackEvent**, il suffit d'ajouter ou supprimer une chaîne dans le tableau **event**, et tout le reste est automatique.

Lorsque **report()** est appelé on lui donne le nom de l'événement et la chaîne des paramètres de cet événement. Il utilise le **HashMap h** de la classe externe pour rechercher le **JTextField** associé à l'événement portant ce nom, et place alors la chaîne des paramètres dans ce champ.

Cet exemple est amusant à utiliser car on peut réellement voir ce qui se passe avec les événements dans son programme.

Un catalogue de composants Swing

Maintenant que nous connaissons les *layout managers* et le modèle d'événements, nous sommes prêts pour voir comment utiliser les composants Swing. Cette section est une visite non exhaustive des composants Swing et des fonctionnalités que vous utiliserez probablement la plupart du temps. Chaque exemple est conçu pour être de taille raisonnable de manière à pouvoir facilement récupérer le code dans d'autres programmes.

Vos pouvez facilement voir à quoi ressemble chacun de ces exemples en fonctionnement, en visualisant les pages HTML dans le code source téléchargeable de ce chapitre.

Gardez en tête :

1. La documentation HTML de *java.sun.com* comprend toutes les classes et méthodes de Swing (seules quelques-unes sont montrées ici),
2. Grâce aux conventions de nommage utilisées pour les événements Swing, il est facile de deviner comment écrire et installer un gestionnaire d'un événement de type donné. Utilisez le programme de recherche **ShowAddListeners.java** introduit plus avant dans ce chapitre pour faciliter votre investigation d'un composant particulier.
3. Lorsque les choses deviendront compliquées, passez à un *GUI builder*.

Boutons

Swing comprend un certain nombre de boutons de différents types. Tous les boutons, boîtes à cocher [*checkboxes*], boutons radio [*radio buttons*], et même les éléments de menus [*menu items*] héritent de **AbstractButton** (qui, vu qu'ils comprennent les éléments de menus, auraient probablement été mieux nommés **AbstractChooser** ou quelque chose du genre). Nous verrons l'utilisation des éléments de menus bientôt, mais l'exemple suivant montre les différents types de boutons existants :

```

//: c13:Buttons.java

// Divers boutons Swing.
// <applet code=Buttons
// width=350 height=100></applet>

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

```

```

public class Buttons extends JApplet {

    JButton jtb = new JButton("JButton");

    BasicArrowButton

    up = new BasicArrowButton(
        BasicArrowButton.NORTH),
    down = new BasicArrowButton(
        BasicArrowButton.SOUTH),
    right = new BasicArrowButton(
        BasicArrowButton.EAST),
    left = new BasicArrowButton(
        BasicArrowButton.WEST);

    public void init() {

        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(jtb);
        cp.add(new JToggleButton("JToggleButton"));
        cp.add(new JCheckBox("JCheckBox"));
        cp.add(new JRadioButton("JRadioButton"));
        JPanel jp = new JPanel();
        jp.setBorder(new TitledBorder("Directions"));
        jp.add(up);
        jp.add(down);
        jp.add(left);
        jp.add(right);
        cp.add(jp);
    }

    public static void main(String[] args) {

        Console.run(new Buttons(), 350, 100);
    }

} ///:~

```

On commence par le **BasicArrowButton** de **javax.swing.plaf.basic**, puis on continue avec les divers types de boutons. Si vous exécutez cet exemple, vous verrez que le *toggle button* (bouton inverseur) garde sa dernière position, enfoncé ou relâché. Mais les boîtes à cocher et les boutons radio se comportent de manière identique, on les clique pour les (dé)sélectionner (ils sont hérités de **JToggleButton**).

Groupes de boutons

Si on désire des boutons radio qui se comportent selon un "ou exclusif", il faut les ajouter à un groupe de boutons. Mais, comme l'exemple ci-dessous le montre, tout **AbstractButton** peut être ajouté à un **ButtonGroup**.

Pour éviter de répéter beaucoup de code, cet exemple utilise la réflexion pour générer les groupes de différents types de boutons. Ceci peut se voir dans **makeBPanel()**, qui crée un groupe de boutons et un **JPanel**. Le second argument de **makeBPanel()** est un tableau de **String**. Pour chaque **String**, un bouton de la classe désignée par le premier argument est ajouté au **JPanel** :

```

//: c13:ButtonGroups.java

// Utilise la reflexion pour creer des groupes
// de differents types de AbstractButton.
// <applet code=ButtonGroups
// width=500 height=300></applet>

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

public class ButtonGroups extends JApplet {
    static String[] ids = {
        "June", "Ward", "Beaver",
        "Wally", "Eddie", "Lumpy",
    };

    static JPanel
    makeBPanel(Class bClass, String[] ids) {
        ButtonGroup bg = new ButtonGroup();
        JPanel jp = new JPanel();
        String title = bClass.getName();
        title = title.substring(
            title.lastIndexOf('.') + 1);
        jp.setBorder(new TitledBorder(title));
        for(int i = 0; i < ids.length; i++) {

```

```

AbstractButton ab = new JButton("failed");

try {
    // Obtient la méthode de construction dynamique
    // qui demande un argument String :
    Constructor ctor = bClass.getConstructor(
        new Class[] { String.class });
    // Creation d'un nouve objet :
    ab = (AbstractButton)ctor.newInstance(
        new Object[]{ids[i]});
} catch(Exception ex) {
    System.err.println("can't create " +
        bClass);
}

bg.add(ab);
jp.add(ab);
}

return jp;
}

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    cp.add(makeBPanel(JButton.class, ids));
    cp.add(makeBPanel(JToggleButton.class, ids));
    cp.add(makeBPanel(JCheckBox.class, ids));
    cp.add(makeBPanel(JRadioButton.class, ids));
}

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

} ///:~

```

Le titre de la bordure est construit à partir du nom de la classe, en enlevant tout ce qui concerne son chemin. L'**AbstractButton** est initialisé comme un **JButton** qui a le label "failed" de sorte que si on ignore un message d'exception le problème se verra à l'écran. La méthode **getConstructor()** fournit un objet **Constructor** qui prend un tableau d'arguments des types du tableau **Class** passé à **getConstructor()**. Ensuite il suffit d'appeler **newInstance()**, en lui passant un tableau d'**Objects** contenant les arguments, dans ce cas il s'agit des **Strings** du tableau **ids**.

Ceci ajoute un peu de complexité à ce qui est un processus simple. Pour obtenir un comportement de "OU exclusif" avec des boutons, on crée un groupe de boutons et on ajoute au groupe chaque bouton pour lequel on désire ce comportement. Lorsqu'on exécute le programme, on voit que tous les boutons, à l'exception de **JButton**, montrent ce comportement de "OU exclusif".

Icones

On peut utiliser un **Icon** dans un **JLabel** ou tout ce qui hérite de **AbstractButton** (y compris **JButton**, **JCheckBox**, **JRadioButton**, et les différents types de **JMenuItem**). L'utilisation d'**Icons** avec des **JLabels** est assez directe (on verra un exemple plus tard). L'exemple suivant explore toutes les façons d'utiliser des **Icons** avec des boutons et leurs descendants.

Vous pouvez utiliser les fichiers gif que vous voulez, mais ceux utilisés dans cet exemple font partie de la livraison du code de ce livre, disponible à www.BruceEckel.com. Pour ouvrir un fichier et utiliser l'image, il suffit de créer un **ImageIcon** et de lui fournir le nom du fichier. A partir de là on peut utiliser l'**Icon** obtenu dans le programme.

Remarquez que l'information de chemin est codée en dur dans cet exemple; vous devrez changer ce chemin pour qu'il corresponde à l'emplacement des fichiers des images.

```

//: c13:Faces.java

// Comportement des Icones dans des Jbuttons.

// <applet code=Faces
// width=250 height=100></applet>

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

public class Faces extends JApplet {

    // L'information de chemin suivante est nécessaire
    // pour l'exécution via une applet directement depuis le disque :

    static String path =

        "C:/aaa-TIJ2-distribution/code/c13/";

    static Icon[] faces = {

        new ImageIcon(path + "face0.gif"),

        new ImageIcon(path + "face1.gif"),

        new ImageIcon(path + "face2.gif"),

        new ImageIcon(path + "face3.gif"),

        new ImageIcon(path + "face4.gif"),
    }

```

```

};

JButton

    jb = new JButton("JButton", faces[3]),
    jb2 = new JButton("Disable");

boolean mad = false;

public void init() {
    Container cp = getContentPane();
    cp.setLayout(new FlowLayout());
    jb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            if(mad) {
                jb.setIcon(faces[3]);
                mad = false;
            } else {
                jb.setIcon(faces[0]);
                mad = true;
            }
            jb.setVerticalAlignment(JButton.TOP);
            jb.setHorizontalAlignment(JButton.LEFT);
        }
    });
    jb.setRolloverEnabled(true);
    jb.setRolloverIcon(faces[1]);
    jb.setPressedIcon(faces[2]);
    jb.setDisabledIcon(faces[4]);
    jb.setToolTipText("Yow!");
    cp.add(jb);
    jb2.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e){
            if(jb.isEnabled()) {
                jb.setEnabled(false);
                jb2.setText("Enable");
            } else {
                jb.setEnabled(true);
            }
        }
    });
}

```

```

        jb2.setText("Disable");
    }
}
});
cp.add(jb2);
}

public static void main(String[] args) {
    Console.run(new Faces(), 400, 200);
}
} ///:~

```

Un **Icon** peut être utilisé dans de nombreux constructeurs, mais on peut aussi utiliser **setIcon()** pour ajouter ou changer un **Icon**. Cet exemple montre également comment un **JButton** (ou un quelconque **AbstractButton**) peut positionner les différentes sortes d'icônes qui apparaissent lorsque des choses se passent sur ce bouton : lorsqu'il est enfoncé, invalidé, ou lorsqu'on roule par dessus *[rolled over]* (la souris passe au-dessus sans cliquer). On verra que ceci donne au bouton une sensation d'animation agréable.

Infobulles *[Tool tips]*

L'exemple précédent ajoutait un *tool tip* au bouton. La plupart des classes qu'on utilisera pour créer une interface utilisateur sont dérivées de **JComponent**, qui contient une méthode appelée **setToolTipText(String)**. Donc pratiquement pour tout ce qu'on place sur un formulaire, il suffit de dire (pour un objet **jc** de toute classe dérivée de **JComponent**) :

```
jc.setToolTipText("My tip");
```

et lorsque la souris reste au-dessus de ce **JComponent** pour un temps prédéterminé, une petite boîte contenant le texte va apparaître à côté de la souris.

Champs de texte *[Text Fields]*

Cet exemple montre le comportement supplémentaire dont sont capables les **JTextFields** :

```

///: c13:TextFields.java
// Champs de texte et événements Java.
// <applet code=TextFields width=375
// height=125></applet>

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

```

```

public class TextFields extends JApplet {

    JButton

    b1 = new JButton("Get Text"),

    b2 = new JButton("Set Text");

    JTextField

    t1 = new JTextField(30),

    t2 = new JTextField(30),

    t3 = new JTextField(30);

    String s = new String();

    UpperCaseDocument

    ucd = new UpperCaseDocument();

    public void init() {

        t1.setDocument(ucd);

        ucd.addDocumentListener(new T1());

        b1.addActionListener(new B1());

        b2.addActionListener(new B2());

        DocumentListener dl = new T1();

        t1.addActionListener(new T1A());

        Container cp = getContentPane();

        cp.setLayout(new FlowLayout());

        cp.add(b1);

        cp.add(b2);

        cp.add(t1);

        cp.add(t2);

        cp.add(t3);

    }

    class T1 implements DocumentListener {

        public void changedUpdate(DocumentEvent e){}

        public void insertUpdate(DocumentEvent e){

            t2.setText(t1.getText());

            t3.setText("Text: " + t1.getText());

        }

        public void removeUpdate(DocumentEvent e){

```



```

        t2.setText(t1.getText());
    }
}

class T1A implements ActionListener {
    private int count = 0;
    public void actionPerformed(ActionEvent e) {
        t3.setText("t1 Action Event " + count++);
    }
}

class B1 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        if(t1.getSelectedText() == null)
            s = t1.getText();
        else
            s = t1.getSelectedText();
        t1.setEditable(true);
    }
}

class B2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        ucd.setUpperCase(false);
        t1.setText("Inserted by Button 2: " + s);
        ucd.setUpperCase(true);
        t1.setEditable(false);
    }
}

public static void main(String[] args) {
    Console.run(new TextFields(), 375, 125);
}

class UpperCaseDocument extends PlainDocument {
    boolean upperCase = true;
    public void setUpperCase(boolean flag) {

```

```

        upperCase = flag;
    }

    public void insertString(int offset,
        String string, AttributeSet attributeSet)
        throws BadLocationException {
        if(upperCase)
            string = string.toUpperCase();
        super.insertString(offset,
            string, attributeSet);
    }
} ///:~

```

Le **JTextField t3** est inclus pour servir d'emplacement pour signaler lorsque l'*action listener* du **JTextField t1** est lancé. On verra que l'*action listener* d'un **JTextField** n'est lancé que lorsqu'on appuie sur la touche enter.

Le **JTextField t1** a plusieurs *listeners* attachés. le *listener* T1 est un **Document Listener** qui répond à tout changement dans le document (le contenu du **JTextField**, dans ce cas). Il copie automatiquement tout le texte de **t1** dans **t2**. De plus, le document **t1** est positionné à une classe dérivée de **PlainDocument**, appelée **UpperCaseDocument**, qui force tous les caractères en majuscules. Il détecte automatiquement les retours en arrière [*backspaces*] et effectue l'effacement, tout en ajustant le curseur et gérant tout de la manière attendue.