

07.05.2001 - Version 0.2 :

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

12.11.2000 - Version 0.1 :

- Dernière mise à jour de la version française.

13: Création de Fenêtres et Applets

Une directive de conception fondamentale est « rendre les choses simples faciles, et les choses difficiles possibles ».

L'objectif initial de la conception de la bibliothèque d'interface utilisateur graphique [*graphical user interface (GUI)*] en Java 1.0 était de permettre au programmeur de construire une GUI qui a un aspect agréable sur toutes les plateformes. Ce but n'a pas été atteint. L'*Abstract Window Toolkit* (AWT) de Java 1.0 produit au contraire une GUI d'aspect aussi mé diocre sur tous les systèmes. De plus, elle est restrictive : on ne peut utiliser que quatre fontes, et on n'a accès à aucun des éléments de GUI sophistiqués disponibles dans son système d'exploitation. Le modèle de programmation de l'AWT Java 1.0 est aussi maladroit et non orienté objet. Un étudiant d'un de mes séminaires (qui était chez Sun lors de la création de Java) m'a expliqué pourquoi : l'AWT original avait été imaginé, conçu, et implémenté en un mois. Certainement une merveille de productivité, mais aussi un exemple du fait que la conception est importante.

La situation s'est améliorée avec le modèle d'événements de l'AWT de Java 1.1, qui a une approche beaucoup plus claire et orientée objet, avec également l'ajout des JavaBeans, un modèle de programmation par composants qui est tourné vers la création facile d'environnements de programmation visuels. Java 2 termine la transformation depuis l'ancien AWT de Java 1.0 en remplaçant à peu près tout par les *Java Foundation Classes* (JFC), dont la partie GUI est appelée « Swing ». Il s'agit d'un ensemble riche de JavaBeans faciles à utiliser et faciles à comprendre, qui peuvent être glissés et déposés (ou programmés manuellement) pour créer une GUI qui vous donnera (enfin) satisfaction. La règle de la « version 3 » de l'industrie du logiciel (un produit n'est bon qu'à partir de la version 3) semble également se vérifier pour les langages de programmation.

Ce chapitre ne couvre que la bibliothèque moderne Swing de Java 2, et fait l'hypothèse raisonnable que Swing est la bibliothèque finale pour les GUI Java. Si pour une quelconque raison vous devez utiliser le « vieux » AWT d'origine (parce que vous maintenez du code ancien ou que vous avez des limitations dans votre browser), vous pouvez trouver cette présentation dans la première édition de ce livre, téléchargeable à www.BruceEckel.com (également incluse sur le CD ROM fourni avec ce livre).

Dès le début de ce chapitre, vous verrez combien les choses sont différentes selon que vous voulez créer une applet ou une application normale utilisant Swing, et comment créer des programmes qui sont à la fois des applets et des applications, de sorte qu'ils puissent être exécutés soit dans un browser soit depuis la ligne de commande. Presque tous les exemples de GUI seront exécutables aussi bien comme des applets que comme des applications.

Soyez conscients que ceci n'est pas un glossaire complet de tous les composants Swing, ou de toutes les méthodes pour les classes décrites. Ce qui se trouve ici a pour but d'être simple. La bibliothèque Swing est vaste, et le but de ce chapitre est uniquement de vous permettre de démarrer avec les notions essentielles et d'être à l'aise avec les concepts. Si vous avez besoin de plus, alors Swing peut probablement vous donner ce que vous voulez si vous avez la volonté de faire des recherches.

Je suppose ici que vous avez téléchargé et installé les documents (gratuits) de la bibliothèque Java au format HTML depuis *java.sun.com* et que vous parcourrez les classes **javax.swing** dans cette documentation pour voir tous les détails et toutes les méthodes de la bibliothèque Swing. Grâce à la simplicité de la conception de Swing, ceci sera souvent suffisant pour régler votre problème. Il y a de nombreux livres (assez épais) dédiés uniquement à Swing et vous vous y référerez si vous avez besoin d'approfondir, ou si vous voulez modifier le comportement par défaut de Swing.

En apprenant Swing vous découvrirez que :

1. Swing est un bien meilleur modèle de programmation que ce que vous avez probablement vu dans d'autres langages et environnements de développement. Les JavaBeans (qui seront présentées vers la fin de ce chapitre) constituent l'ossature de cette bibliothèque.
2. Les « GUI builders »(environnements de programmation visuels) sont un *must-have* d'un environnement de développement Java complet. Les JavaBeans et Swing permettent au « builder » d'écrire le code pour vous lorsque vous placez les composants sur des formulaires à l'aide d'outils graphiques. Ceci permet non seulement d'accélérer le développement lors de la création de GUI, mais aussi d'expérimenter plus et de ce fait d'essayer plus de modèles et probablement d'en obtenir un meilleur.
3. La simplicité et la bonne conception de Swing signifie que même si vous utilisez un GUI builder plutôt que du codage manuel, le code résultant sera encore compréhensible ; ceci résout un gros problème qu'avaient les GUI builders, qui généraient souvent du code illisible.

Swing contient tous les composants attendus dans une interface utilisateur moderne, depuis des boutons contenant des images jusqu'à des arborescences et des tables. C'est une grosse bibliothèque mais elle est conçue pour avoir la complexité appropriée pour la tâche à effectuer ; si quelque chose est simple, vous n'avez pas besoin d'écrire beaucoup de code, mais si vous essayez de faire des choses plus compliquées, votre code devient proportionnellement plus complexe. Ceci signifie qu'il y a un point d'entrée facile, mais que vous avez la puissance à votre disposition si vous en avez besoin.

Ce que vous aimerez dans Swing est ce qu'on pourrait appeler l'« orthogonalité d'utilisation ». C'est à dire, une fois que vous avez compris les idées générales de la bibliothèque, vous pouvez les appliquer partout. Principalement grâce aux conventions de nommage standards, en écrivant ces exemples je pouvais la plupart du temps deviner le nom des méthodes et trouver juste du premier coup, sans rechercher quoi que ce soit. C'est certainement la marque d'une bonne conception de la bibliothèque. De plus, on peut en général connecter des composants entre eux et les choses fonctionnent correctement.

Pour des question de rapidité, tous les composants sont « légers », et Swing est écrit entièrement en Java pour la portabilité.

>La navigation au clavier est automatique ; vous pouvez exécuter une application Swing sans utiliser la souris, et ceci ne réclame aucune programmation supplémentaire. Le scrolling se fait sans effort, vous emballez simplement votre composant dans un **JScrollPane** lorsque vous l'ajoutez à votre formulaire. Des fonctionnalités telles que les « infobulles » [tool tips] ne demandent qu'une seule ligne de code pour les utiliser.

Swing possède également une fonctionnalité assez avancée, appelée le « pluggable look and feel », qui signifie que l'apparence de l'interface utilisateur peut être modifiée dynamiquement pour s'adapter aux habitudes des utilisateurs travaillant sur des plateformes et systèmes d'exploitation différents. Il est même possible (quoique difficile) d'inventer son propre « look and feel ».

L'applet de base

Un des buts de la conception Java est de créer des *applets*, qui sont des petits programmes s'exécutant à l'intérieur d'un browser Web. Parce qu'ils doivent être sûrs, les applets sont limitées dans ce qu'ils peuvent accomplir. Toutefois, les applets sont un outil puissant de programmation côté client, un problème majeur pour le Web.

Les restrictions des applets

Programmer dans un applet est tellement restrictif qu'on en parle souvent comme étant « dans le bac à sable », car il y a toujours quelqu'un (le système de sécurité Java lors de l'exécution [*Java run-time security system*]) qui vous surveille.

Cependant, on peut aussi sortir du bac à sable et écrire des applications normales plutôt que des applets ; dans ce cas on accède aux autres fonctionnalités de son OS. Nous avons écrit des applications tout au long de ce livre, mais il s'agissait d'*applications console*, sans aucun composant graphique. Swing peut aussi être utilisé pour construire des interfaces utilisateur graphiques pour des applications normales.

On peut en général répondre à la question de savoir ce qu'une applet peut faire en considérant ce qu'elle est *supposée* faire : étendre les fonctionnalités d'une page Web dans un browser. Comme en tant que surfeur sur le Net on ne sait jamais si une page Web vient d'un site amical ou pas, on veut que tout le code qu'il exécute soit sûr. De ce fait, les restrictions les plus importantes sont probablement :

1. *Une applet ne peut pas toucher au disque local.* Ceci signifie écrire *ou* lire, puisqu'on ne voudrait pas qu'une applet lise et transmette des informations privées sur Internet sans notre permission. Ecrire est interdit, bien sûr car ce serait une invitation ouverte aux virus. Java permet une *signature digitale* des applets. Beaucoup de restrictions des applets sont levées si on autorise des *applets de confiance* [*trusted applets*] (celles qui sont signées par une source dans laquelle on a confiance) à accéder à sa machine.
2. Les applets peuvent mettre du temps à s'afficher, car il faut les télécharger complètement à chaque fois, nécessitant un accès au serveur pour chacune des classes. Le browser peut mettre l'applet dans son cache, mais ce n'est pas garanti. Pour cette raison, on devrait toujours emballer ses applets dans un fichier JAR (Java ARchive) qui rassemble tous les composants de l'applet (y compris d'autres fichiers **.class** ainsi que les images et les sons) en un seul fichier compressé qui peut être téléchargé en une seule transaction avec le serveur. La « signature digitale » existe pour chacun des fichiers contenus dans le fichier JAR.

Les avantages d'une applet

Si on admet ces restrictions, les applets ont des avantages, en particulier pour la création d'applications client/serveur ou réseaux :

1. *Il n'y a pas de problème d'installation.* Une applet est réellement indépendante de la plateforme (y compris pour jouer des fichiers audio, etc.) et donc il n'est pas nécessaire de modifier le code en fonction des plateformes ni d'effectuer des « réglages » à l'installation. En fait, l'installation est automatique chaque fois qu'un utilisateur charge la page Web qui contient les applets, de sorte que les mises à jour se passent en silence et automatiquement. Dans les systèmes client/serveur traditionnels, créer et installer une nouvelle version du logiciel client est souvent un cauchemar.
2. *On ne doit pas craindre un code defectueux affectant le système de l'utilisateur,* grâce au système de sécurité inclus au coeur du langage Java et de la structure de l'applet. Ceci, ainsi que le point précédent, rend Java populaire pour les applications *intranet* client/serveur qui n'existent qu'à l'intérieur d'une société

ou dans une zone d'opérations réduite, où l'environnement utilisateur (le navigateur Web et ses extensions) peut être spécifié et/ou contrôlé.

Comme les applets s'intègrent automatiquement dans le code HTML, on dispose d'un système de documentation intégré et indépendant de la plateforme pour le support de l'applet. C'est une astuce intéressante, car on a plutôt l'habitude d'avoir la partie documentation du programme plutôt que l'inverse.

Les squelettes d'applications

Les bibliothèques sont souvent groupées selon leurs fonctionnalités. Certaines bibliothèques, par exemple, sont utilisées telles quelles. Les classes **String** et **ArrayList** de la bibliothèque standard Java en sont des exemples. D'autres bibliothèques sont conçues comme des briques de construction pour créer d'autres classes. Une certaine catégorie de bibliothèques est le *squelette d'applications* [*application framework*], dont le but est d'aider à la construction d'applications en fournissant une classe ou un ensemble de classes reproduisant le comportement de base dont vous avez besoin dans chaque application d'un type donné. Ensuite, pour particulariser le comportement pour ses besoins propres, on hérite de la classe et on redéfinit les méthodes qui nous intéressent. Un squelette d'application est un bon exemple de la règle « séparer ce qui change de ce qui ne change pas », car on essaie de localiser les parties spécifiques d'un programme dans les méthodes redéfinies [\[62\]](#).

les applets sont construites à l'aide d'un squelette d'application. On hérite de la classe **JApplet** et on redéfinit les méthodes adéquates. Quelques méthodes contrôlent la création et l'exécution d'une applet dans une page Web :

Method	Operation
init()	Appelée automatiquement pour effectuer la première initialisation de l'applet, y compris la disposition des composants. Il faut toujours redéfinir cette méthode.
start()	Appelée chaque fois que l'applet est rendue visible du navigateur Web, pour permettre à l'applet de démarrer ses opérations normales (en particuliers celles qui sont arrêtées par stop()). Appelée également après init() .
stop()	Appelée chaque fois que l'applet redevient invisible du navigateur Web, pour permettre à l'applet d'arrêter les opérations coûteuses. Appelée également juste avant destroy() .
destroy()	Appelée lorsque l'applet est déchargée de la page pour effectuer la libération finale des ressources lorsque l'applet n'est plus utilisée.

Avec ces informations nous sommes prêts à créer une applet simple :

```

//: c13:Applet1.java
// Très simple applet.

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

public class Applet1 extends JApplet {

    public void init() {

        getContentPane().add(new JLabel("Applet!"));

    }
}

```

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

Remarquons qu'il n'est pas obligatoire qu'une applet ait une méthode **main()**. C'est complètement câblé dans le squelette d'application ; on met tout le code de démarrage dans **init()**.

Dans ce programme, la seule activité consiste à mettre un label de texte sur l'applet, à l'aide de la classe **JLabel** (l'ancien AWT s'était approprié le nom **Label** ainsi que quelques autres noms de composants, de sorte qu'on verra souvent un « **J** » au début des composants Swing). Le constructeur de cette classe reçoit une **String** et l'utilise pour créer le label. Dans le programme ci-dessus ce label est placé dans le formulaire.

La méthode **init()** est responsable du placement de tous les composants du formulaire en utilisant la méthode **add()**. On pourrait penser qu'il devrait être suffisant d'appeler simplement **add()**, et c'est ainsi que cela se passait dans l'ancien AWT. Swing exige quant à lui qu'on ajoute tous les composants sur la « vitre de contenu » [*content pane*] d'un formulaire, et il faut donc appeler **getContentPane()** au cours de l'opération **add()**.

Exécuter des applets dans un navigateur Web

Pour exécuter ce programme, il faut le placer dans une page Web et visualiser cette page à l'aide d'un navigateur Web configuré pour exécuter des applets Java. Pour placer une applet dans une page Web, on place un tag spécial dans le source HTML de cette page [\[63\]](#) pour dire à la page comment charger et exécuter cette applet.

Ce système était très simple lorsque Java lui-même était simple et que tout le monde était dans le même bain et incorporait le même support de Java dans les navigateurs Web. Il suffisait alors d'un petit bout de code HTML dans une page Web, tel que :

```
<applet code=Applet1 width=100 height=50>
</applet>
```

Ensuite vint la guerre des navigateurs et des langages, et nous (programmeurs aussi bien qu'utilisateurs finaux) y avons perdu. Au bout d'un moment, JavaSoft s'est rendu compte qu'on ne pouvait plus supposer que les navigateurs supportaient la version correcte de Java, et que la seule solution était de fournir une sorte d'extension qui se conformerait au mécanisme d'extension des navigateurs. En utilisant ce mécanisme d'extensions (qu'un fournisseur de navigateur ne peut pas supprimer -dans l'espoir d'y gagner un avantage sur la concurrence- sans casser aussi toutes les autres extensions), JavaSoft garantit que Java ne pourra pas être supprimé d'un navigateur Web par un fournisseur qui y serait hostile.

Avec Internet Explorer, le mécanisme d'extensions est le contrôle ActiveX, et avec Netscape c'est le plug-in. Dans le code HTML, il faut fournir les tags qui supportent les deux. Voici à quoi ressemble la page HTML la plus simple pour **Applet1** : [\[64\]](#)

```
///: c13:Applet1.html
<html><head><title>Applet1</title></head><hr>
<OBJECT
  classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
  width="100" height="50" align="baseline" codebase="http://java.sun.com/products/plu
  <PARAM NAME="code" VALUE="Applet1.class">
```

```

<PARAM NAME="codebase" VALUE=".">

<PARAM NAME="type" VALUE="application/x-java-applet;version=1.2.2">

<COMMENT>

  <EMBED type=

    "application/x-java-applet;version=1.2.2"

    width="200" height="200" align="baseline"

    code="Applet1.class" codebase="."

    pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">

  <NOEMBED>

</COMMENT>

  No Java 2 support for APPLETT!!

</NOEMBED>

</EMBED>

</OBJECT>

<hr></body></html>

///:~

```

Certaines de ces lignes étaient trop longues et ont été coupées pour être insérées dans cette page. Le code inclus dans les sources de ce livre (sur le CD ROM de ce livre, et téléchargeable sur www.BruceEckel.com) fonctionne sans qu'on doive s'occuper de corriger les sauts de lignes.

Le contenu de code fournit le nom du fichier **.class** dans lequel se trouve l'applet. Les paramètres **width** et **height** spécifient la taille initiale de l'applet (en pixels, comme avant). Il y a d'autres éléments qu'on peut placer dans le tag applet : un endroit où on peut trouver d'autres fichiers .class sur Internet (**codebase**), des informations d'alignement (**align**), un identificateur spécial permettant à des applets de communiquer entre eux (**name**), et des paramètres des applets pour fournir des informations que l'applet peut récupérer. Les paramètres sont de la forme :

```
<param name="identifiant" value = "information">
```

et il peut y en avoir autant qu'on veut.

Le package de codes source de ce livre fournit une page HTML pour chacune des applets de ce livre, et de ce fait de nombreux exemples du tag applet. Vous pouvez trouver une description complète et à jour des détails concernant le placement d'applets dans des pages web à l'adresse java.sun.com.

Utilisation de Appletviewer

Le JDK de Sun (en téléchargement libre depuis java.sun.com) contient un outil appelé l'*Appletviewer* qui extrait le tag **<applet>** du fichier HTML et exécute les applets sans afficher le texte HTML autour. Comme l'*Appletviewer* ignore tout ce qui n'est pas tags APPLET, on peut mettre ces tags dans le source Java en commentaires :

```
// <applet code=MyApplet width=200 height=100>
// </applet>
```

De cette manière, on peut lancer « **appletviewer MyApplet.java** » et il n'est pas nécessaire de créer de petits fichiers HTML pour lancer des tests. Par exemple, on peut ajouter ces tags HTML en commentaires dans **Applet1.java**:

```
//: c13:Applet1b.java
// Embedding the applet tag for Appletviewer.
// <applet code=Applet1b width=100 height=50>
// </applet>

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

public class Applet1b extends JApplet {>
    public void >init() {
        getContentPane().add(new JLabel("Applet!"));
    }
} ///:~
```

Maintenant on peut invoquer l'applet avec la commande

```
appletviewer Applet1b.java
```

Dans ce livre, cette forme sera utilisée pour un test facile des applets. On verra bientôt une autre méthode de codage qui permettra d'exécuter des applets depuis la ligne de commande sans l'Appletviewer.

Tester les applets

On peut exécuter un test simple sans aucune connexion réseau en lançant son navigateur Web et en ouvrant le fichier HTML contenant le tag applet. Au chargement du fichier HTML, le navigateur découvre le tag applet et part à la recherche du fichier **.class** spécifié par le contenu de **code**. Bien sûr, il utilise le CLASSPATH pour savoir où chercher, et si le fichier **.class** n'est pas dans le CLASSPATH, il émettra un message d'erreur dans sa ligne de status pour signaler qu'il n'a pas pu trouver le fichier **.class**.

Quand on veut essayer ceci sur son site Web les choses sont un peu plus compliquées. Tout d'abord il faut *avoir* un site Web, ce qui pour la plupart des gens signifie avoir un Fournisseur d'Accès à Internet (FAI) [*Internet Service Provider (ISP)*]. Comme l'applet est simplement un fichier ou un ensemble de fichiers, le FAI n'a pas besoin de fournir un support particulier pour Java. Il faut disposer d'un moyen pour copier les fichiers HTML et les fichiers **.class** depuis chez vous vers le bon répertoire sur la machine du FAI. Ceci est normalement fait avec un programme de File Transfer Protocol (FTP), dont il existe beaucoup d'exemples disponibles gratuitement ou

comme sharewares. Il semblerait donc que tout ce qu'il y a à faire est d'envoyer les fichiers sur la machine du FAI à l'aide de FTP, et ensuite de se connecter au site et au fichier HTML en utilisant son navigateur ; si l'applet se charge et fonctionne, alors tout va bien, n'est-ce pas ?

C'est là qu'on peut se faire avoir. Si le navigateur de la machine client ne peut pas localiser le fichier **.class** sur le serveur, il va le rechercher à l'aide du CLASSPATH sur la machine *locale*. De ce fait l'applet pourrait bien ne pas se charger correctement depuis le serveur, mais tout paraît correct lors du test parce que le navigateur le trouve sur la machine locale. Cependant, lorsque quelqu'un d'autre se connecte, son navigateur ne la trouvera pas. Donc lorsque vous testez, assurez vous d'effacer les fichiers **.class** (ou **.jar**) de votre machine locale pour vérifier qu'ils existent au bon endroit sur le serveur.

Un des cas les plus insidieux qui me soit arrivé s'est produit lorsque j'ai innocemment placé une applet dans un package. Après avoir téléchargé sur le serveur le fichier HTML et l'applet, le serveur fut trompé sur le chemin d'accès à l'applet à cause du nom du package. Cependant, mon navigateur l'avait trouvé dans le CLASSPATH local. J'étais donc le seul à pouvoir charger correctement l'applet. J'ai mis un certain temps à découvrir que l'instruction **package** était la coupable. En général il vaut mieux ne pas utiliser l'instruction **package** dans une applet.

Exécuter des applets depuis la ligne de commande

Parfois on voudrait qu'un programme fenêtré fasse autre chose que se trouver dans une page Web. Peut-être voudrait-on aussi faire certaines des choses qu'une application « normale » peut faire, mais en gardant la glorieuse portabilité instantanée fournie par Java. Dans les chapitres précédents de ce livre, nous avons fait des applications de ligne de commande, mais dans certains environnements (le Macintosh par exemple) il n'y a pas de ligne de commande. Voilà un certain nombre de raisons pour vouloir construire un programme fenêtré n'étant pas une applet. C'est certainement un désir légitime.

La bibliothèque Swing nous permet de construire une application qui conserve le « look and feel » du système d'exploitation sous-jacent. Si vous voulez faire des applications fenêtrées, cela n'a de sens [\[65\]](#) que si vous pouvez utiliser la dernière version de Java et ses outils associés, afin de pouvoir fournir des applications qui ne perturberont pas vos utilisateurs. Si pour une raison ou une autre vous devez utiliser une ancienne version de Java, réfléchissez-y bien avant de vous lancer dans la construction d'une application fenêtrée importante.

On a souvent besoin de créer une classe qui peut être appelée aussi bien comme une fenêtre que comme une applet. C'est particulièrement utile lorsqu'on teste des applets, car il est souvent plus facile et plus simple de lancer l'applet-application depuis la ligne de commande que de lancer un navigateur Web ou l'Appletviewer.

Pour créer une applet qui peut être exécutée depuis la ligne de commande, il suffit d'ajouter un **main()** à l'applet, dans lequel on construit une instance de l'applet dans un **JFrame** [\[66\]](#). En tant qu'exemple simple, observons **Applet1b.java** modifié pour fonctionner aussi bien en tant qu'application qu'en tant qu'applet :

```
///  
//: c13:Applet1c.java  
// Une application et une applet.  
// <applet code=Applet1c width=100 height=50>  
// </applet>  
  
import javax.swing.*;  
  
import java.awt.*;  
  
import com.bruceeckel.swing.*;
```



```

public class Applet1c extends JApplet {
    public void init() {
        getContentPane().add(new JLabel("Applet!"));
    }

    // Un main() pour l'application :
    public static void main(String[] args) {
        JApplet applet = new Applet1c();
        JFrame frame = new JFrame("Applet1c");

        // Pour fermer l'application :
        Console.setupClosing(frame);

        frame.getContentPane().add(applet);

        frame.setSize(100,50);

        applet.init();
        applet.start();

        frame.setVisible(true);
    }
} ///:~

```

main() est le seul élément ajouté à l'applet, et le reste de l'applet n'est pas modifié. L'applet est créée et ajoutée à un `JFrame` pour pouvoir être affichée.

La ligne :

```
Console.setupClosing(frame);
```

permet la fermeture propre de la fenêtre. **Console** vient de **com.bruceeckel.swing** et sera expliqué un peu plus tard.

On peut voir que dans **main()**, l'applet est explicitement initialisée et démarrée, car dans ce cas le navigateur n'est pas là pour le faire. Bien sûr, ceci ne fournit pas le comportement complet du navigateur, qui appelle aussi **stop()** et **destroy()**, mais dans la plupart des cas c'est acceptable. Si cela pose un problème, on peut forcer les appels soi-même [\[67\]](#).

Notez la dernière ligne :

```
frame.setVisible(true);
```

Sans elle, on ne verrait rien à l'écran.

Un squelette d'affichage

Bien que le code qui transforme des programmes en applets et applications produise des résultats corrects, il devient perturbant et gaspille du papier s'il est utilisé partout. Au lieu de cela, le squelette d'affichage ci-après sera utilisé pour les exemples Swing du reste de ce livre :

```
//: com:bruceeckel:swing:Console.java

// Outil pour exécuter des démos Swing depuis
// la console, aussi bien applets que JFrames.

package com.bruceeckel.swing;

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

public class Console {

    // Crée une chaîne de titre à partir du nom de la classe :
    public static String title(Object o) {
        String t = o.getClass().toString();

        // Enlever le mot "class":
        if(t.indexOf("class") != -1)
            t = t.substring(6);

        return t;
    }

    public static void setupClosing(JFrame frame) {
        // La solution JDK 1.2 Solution avec une
        // classe interne anonyme :
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        // La solution améliorée en JDK 1.3 :
        // frame.setDefaultCloseOperation(
        //     EXIT_ON_CLOSE);
    }

    public static void
```

```

run(JFrame frame, int width, int height) {
    setupClosing(frame);

    frame.setSize(width, height);

    frame.setVisible(true);
}

public static void
run(JApplet applet, int width, int height) {
    JFrame frame = new JFrame(title(applet));

    setupClosing(frame);

    frame.getContentPane().add(applet);

    frame.setSize(width, height);

    applet.init();

    applet.start();

    frame.setVisible(true);
}

public static void
run(JPanel panel, int width, int height) {
    JFrame frame = new JFrame(title(panel));

    setupClosing(frame);

    frame.getContentPane().add(panel);

    frame.setSize(width, height);

    frame.setVisible(true);
}
} ///:~

```

Comme c'est un outil que vous pouvez utiliser vous-mêmes, il est placé dans la bibliothèque **com.bruceeckel.swing**. La classe **Console** contient uniquement des méthodes **static**. La première est utilisée pour extraire le nom de la classe (en utilisant RTTI) depuis n'importe quel objet, et pour enlever le mot « class », qui est ajouté normalement au début du nom par **getClass()**. On utilise les méthodes de **String** : **indexOf()** pour déterminer si le mot « class » est présent, et **substring()** pour générer la nouvelle chaîne sans « class » ou le blanc de fin. Ce nom est utilisé pour étiqueter la fenêtre qui est affichée par les méthodes **run()**.

setupClosing() est utilisé pour cacher le code qui provoque la sortie du programme lorsque la **JFrame** est fermée. Le comportement par défaut est de ne rien faire, donc si on n'appelle pas **setupClosing()** ou un code équivalent pour le **JFrame**, l'application ne se ferme pas. Une des raisons pour laquelle ce code est caché plutôt que d'être placé directement dans les méthodes **run()** est que cela nous permet d'utiliser la méthode en elle-même lorsque ce qu'on veut faire est plus complexe que ce que fournit **run()**. Mais il isole aussi un facteur de changement : Java 2 possède deux manières de provoquer la fermeture de certains types de fenêtres. En JDK

1.2, la solution est de créer une nouvelle classe **WindowAdapter** et d'implémenter **windowClosing()**, comme vu plus haut (la signification de ceci sera expliquée en détails plus tard dans ce chapitre). Cependant lors de la création du JDK 1.3, les concepteurs de la librairie ont observé qu'on a normalement besoin de fermer des fenêtres chaque fois qu'on crée un programme qui n'est pas une applet, et ils ont ajouté **setDefaultCloseOperation()** à **JFrame** et **JDialog**. Du point de vue de l'écriture du code, la nouvelle méthode est plus agréable à utiliser, mais ce livre a été écrit alors qu'il n'y avait pas de JDK 1.3 implémenté sur Linux et d'autres plateformes, et donc dans l'intérêt de la portabilité toutes versions, la modification a été isolée dans **setupClosing()**.

La méthode **run()** est surchargée pour fonctionner avec les **JApplets**, les **JPanels**, et les **JFrames**. Remarquez que **init()** et **start()** ne sont appelées que s'il s'agit d'une **JApplet**.

Maintenant toute applet peut être lancée de la console en créant un **main()** contenant une ligne comme celle-ci :

```
Console.run(new MyClass(), 500, 300);
```

dans laquelle les deux derniers arguments sont la largeur et la hauteur de l'affichage. Voici **Applet1c.java** modifié pour utiliser **Console** :

```
///  
c13:Applet1d.java  
// Console exécute des applets depuis la ligne de commande.  
// <applet code=Applet1d width=100 height=50>  
// </applet>  
  
import javax.swing.*;  
import java.awt.*;  
import com.bruceeckel.swing.*;  
  
public class Applet1d extends JApplet {  
    public void init() {  
        getContentPane().add(new JLabel("Applet!"));  
    }  
  
    public static void main(String[] args) {  
        Console.run(new Applet1d(), 100, 50);  
    }  
} ///  
:~
```

Ceci permet l'élimination de code répétitif tout en fournissant la plus grande flexibilité pour lancer les exemples.