

Traducteur : Armel FORTUN

25.04.2001 - Version 0.2 :

- Mise en forme du code html (titres-hx[verdana], paragraphes-p[Georgia], code-blockquote

11: Le système d'E/S de Java

La création d'un bon système d'entrée/sortie, pour le designer du langage, est l'une des tâches les plus difficiles.

Cette difficulté est mise en évidence par le nombre d'approches différentes. Le défi semblant être dans la couverture de toutes les éventualités. Non seulement il y a de nombreuses sources et de réceptacles d'I/O avec lesquelles vous voudrez communiquer (fichiers, la console, connections réseau), mais vous voudrez converser avec eux dans une grande variété de manières (séquentiel, accès-aléatoire, mis en mémoire tampon, binaire, caractère, par lignes, par mots, etc...).

Les designers de bibliothèque Java ont attaqué ce problème en créant de nombreuses classes. En fait, il y a tellement de classes pour le système d'I/O de Java que cela peut être intimidant au premier abord (ironiquement, le design d'I/O de Java prévient actuellement une explosion de classes). Il y a eu aussi un changement significatif dans la bibliothèque d'I/O après Java 1.0, quand la bibliothèque orientée-**byte** d'origine a été complétée par des classes d'I/O de base Unicode orientées-**char**. La conséquence étant qu'il vous faudra assimiler un bon nombre de classes avant de comprendre suffisamment la représentation de l'I/O Java pour que vous puissiez l'employer correctement. De plus, il est plutôt important de comprendre l'évolution historique de la bibliothèque I/O, même si votre première réaction est « me prenez pas la tête avec l'historique, montrez moi seulement comment l'utiliser ! » Le problème est que sans un point de vue historique vous serez rapidement perdu avec certaines des classes et lorsque vous devrez les utiliser vous ne pourrez et ne les utiliserez pas.

Ce chapitre vous fournira une introduction aux diverses classes d'I/O que comprend la bibliothèque standard de Java et comment les utiliser.

La classe File

Avant d'aborder les classes qui effectivement lisent et écrivent des données depuis des streams (flux), nous allons observer un utilitaire fourni avec la bibliothèque afin de vous assister lors des traitements de répertoire de fichier.

La classe **File** possède un nom décevant — vous pouvez penser qu'elle se réfère à un fichier, mais pas du tout. Elle peut représenter soit le *nom* d'un fichier particulier ou bien les *noms* d'un jeu de fichiers dans un dossier. Si il s'agit d'un jeu de fichiers, vous pouvez faire appel à ce jeu avec la méthode **list()**, et celle-ci renverra un tableau de **String**. Il est de bon sens de renvoyer un tableau plutôt qu'une des mouvantes classes container parce que le nombre d'éléments est fixé, et si vous désirez le listing d'un répertoire différent vous créez seulement un autre objet **File**. En fait, « *CheminDeFichier* ou *FilePath* » aurait été un meilleur nom pour cette classe. Cette partie montre un exemple d'utilisation de cette classe, incluant l'**interface** associée **FilenameFilter**.

Lister un répertoire

Supposons que vous désirez voir un listing de répertoire. L'objet **File** peut être listé de deux manières. Si vous appelez **list()** sans arguments, vous obtiendrez la liste complète du contenu de l'objet **File**. Pourtant, si vous

désirez une liste restreinte — par exemple, si vous voulez tous les fichiers avec une extension **.java** — à ce moment là vous utiliserez un « filtre de répertoire », qui est une classe montrant de quelle manière sélectionner les objets **File** pour la visualisation.

Voici le code pour exemple. Notez que le résultat a été trié sans effort (par ordre alphabétique) en utilisant la méthode **java.util.Array.sort()** et l'**AlphabeticComparator** défini au Chapitre 9 :

```
//: c11:DirList.java
// Affiche le listing d'un répertoire.

import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class DirList {
    public static void main(String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new DirFilter(args[0]));
        Arrays.sort(list,
            new AlphabeticComparator());
        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}

class DirFilter implements FilenameFilter {
    String afn;
    DirFilter(String afn) { this.afn = afn; }
    public boolean accept(File dir, String name) {
        // Information du chemin de répertoire :
        String f = new File(name).getName();
        return f.indexOf(afn) != -1;
    }
}
```

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

La classe **DirFilter** « implémente » l'interface **FilenameFilter**. Il est utile de voir comment est simple l'interface **FilenameFilter** :

```
public interface FilenameFilter {
    boolean accept(File dir, String name);
}
```

Cela signifie : la seule chose dont s'occupe ce type d'objet est de fournir une méthode appelée **accept()**. La finalité derrière la création de cette classe est de fournir la méthode **accept()** à la méthode **list()** de telle manière que **list()** puisse « rappeler » **accept()** pour déterminer quelle noms de fichiers doivent être inclus dans la liste. Ainsi, cette technique fait souvent référence à *un rappel automatique* ou parfois à un [functor] (c'est à dire, **DirFilter** est un functor parce que sa seule fonction est de maintenir une méthode) ou la *Command Pattern* (une entité, ensemble de caractéristiques, de commande). Parce que **list()** prend un objet **FilenameFilter** comme argument, cela veut dire que vous pouvez passer un objet de n'importe quelle classe implémentant **FilenameFilter** afin de choisir (même lors de l'exécution) comment la méthode **list()** devra se comporter. L'objectif du rappel est de fournir une flexibilité dans le comportement du code.

DirFilter montre que parce qu'une **interface** peut contenir qu'un jeu de méthodes, vous n'êtes pas réduit à l'écriture seule de ces méthodes. (Vous devez au moins fournir les définitions pour toutes les méthodes dans une interface, de toutes les manières.) Dans ce cas, le constructeur de **DirFilter** est aussi créé.

La méthode **accept()** doit accepter un objet **File** représentant le répertoire ou un fichier particulier se trouve, et un **String** contenant le nom de ce fichier. Vous pouvez choisir d'utiliser ou ignorer l'un ou l'autre de ces arguments, mais vous utiliserez probablement au moins le nom du fichier. Rappelez vous que la méthode **list()** fait appel à **accept()** pour chacun des noms de fichier de l'objet répertoire pour voir lequel doit être inclus — ceci est indiqué par le résultat **booléen** renvoyé par **accept()**.

Pour être sûr que l'élément avec lequel vous êtes en train de travailler est seulement le nom du fichier et qu'il ne contient pas d'information de chemin, tout ce que vous avez à faire est de prendre l'objet **String** et de créer un objet **File** en dehors de celui-ci, puis d'appeler **getName()**, qui éloigne toutes les informations de chemin (dans une vision d'indépendance vis-à-vis de la plate-forme). Puis **accept()** utilise la méthode **indexOf()** de la classe **String** pour voir si le string recherche **afn** apparaît n'importe où dans le nom du fichier. Si **afn** est trouvé à l'intérieur du string, la valeur retournée sera l'indice de départ d'**afn**, mais si il n'est pas trouvé la valeur retournée sera - 1. Gardez en tête que ce n'est qu'une simple recherche de chaîne de caractères et qui ne possède pas d'expression « globale » de comparaison d'assortiment — comme « fo?.b?r* » — qui est beaucoup plus difficile à réaliser.

La méthode **list()** renvoie un tableau. Vous pouvez interroger ce tableau sur sa longueur et puis vous déplacer d'un bout à l'autre de celui-ci en sélectionnant des éléments du tableau. Cette aptitude de passer facilement un tableau dedans et hors d'une méthode est une amélioration immense supérieure au comportement de C et C++.

Les classes internes anonymes

Cette exemple est idéal pour une ré-écriture en utilisant une classe interne anonyme (décrite au Chapitre 8). Comme première incision, une méthode **filter()** est créée retournant une référence à un **FilenameFilter** :

```
//: c11:DirList2.java

// Utilisation de classes internes anonymes.

import java.io.*;
import java.util.*;
import com.bruceeckel.util.*;

public class DirList2 {
    public static FilenameFilter
    filter(final String afn) {
        // Creation de la classe anonyme interne :
        return new FilenameFilter() {
            String fn = afn;

            public boolean accept(File dir, String n) {
                // Strip path information:
                String f = new File(n).getName();
                return f.indexOf(fn) != -1;
            }
        }; // Fin de la classe anonyme interne.
    }

    public static void main(String[] args) {
        File path = new File(".");
        String[] list;

        if(args.length == 0)
            list = path.list();
        else
            list = path.list(filter(args[0]));

        Arrays.sort(list,
            new AlphabeticComparator());
        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
} ///:~
```

Notez que l'argument de **filter()** doit être **final**. Ceci est requis par la classe interne anonyme pour qu'elle puisse utiliser un objet en dehors de sa portée.

Cette conception est une amélioration parce que la classe **FilenameFilter** est maintenant fortement liée à `DirList2`. Cependant, vous pouvez prendre cette approche en allant plus loin et définir la classe anonyme interne comme un argument de `list()`, en quel cas c'est encore plus léger :

```

//: c11:DirList3.java

// Construction de la classe anonyme interne «sur-place ».

import java.io.*;

import java.util.*;

import com.bruceeckel.util.*;

public class DirList3 {

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

        File path = new File(".");

        String[] list;

        if(args.length == 0)

            list = path.list();

        else

            list = path.list(new FilenameFilter() {

                public boolean

                accept(File dir, String n) {

                    String f = new File(n).getName();

                    return f.indexOf(args[0]) != -1;

                }

            });

        Arrays.sort(list,

            new AlphabeticComparator());

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

            System.out.println(list[i]);

    }

} ///:~

```

L'argument de `main()` est maintenant **final**, puisque la classe anonyme interne utilise directement `args[0]`.

Ceci vous montre comment les classes anonymes internes permettent la création de classes quick-and-dirty pour résoudre des problèmes. Étant donné que tout en Java tourne autour des classes, cela peut être une technique de

code utile. Un avantage étant que cela garde le code résolvant un problème particulier isolé dans un même lieu. D'autre part, cela n'est pas toujours facile à lire, donc vous devrez l'utiliser judicieusement.

Vérification et création de répertoires

La classe **File** est bien plus qu'une représentation d'un fichier ou d'un répertoire existant. Vous pouvez aussi utiliser un objet **File** pour créer un nouveau répertoire ou un chemin complet de répertoire si ils n'existent pas. Vous pouvez également regarder les caractéristiques des fichiers (taille, dernière modification, date, lecture/écriture), voir si un objet **File** représente un fichier ou un répertoire, et supprimer un fichier. Ce programme montre quelques unes des méthodes disponibles avec la classe **File** (voir la documentation HTML à java.sun.com pour le jeu complet) :

```
//: c11:MakeDirectories.java

// Démonstration de l'usage de la classe File pour
// créer des répertoire et manipuler des fichiers.

import java.io.*;

public class MakeDirectories {

    private final static String usage =
        "Usage:MakeDirectories path1 ...\n" +
        "Creates each path\n" +
        "Usage:MakeDirectories -d path1 ...\n" +
        "Deletes each path\n" +
        "Usage:MakeDirectories -r path1 path2\n" +
        "Renames from path1 to path2\n";

    private static void usage() {
        System.err.println(usage);
        System.exit(1);
    }

    private static void fileData(File f) {
        System.out.println(
            "Absolute path: " + f.getAbsolutePath() +
            "\n Can read: " + f.canRead() +
            "\n Can write: " + f.canWrite() +
            "\n getName: " + f.getName() +
            "\n getParent: " + f.getParent() +
            "\n getPath: " + f.getPath() +
```

```
"\n length: " + f.length() +
"\n lastModified: " + f.lastModified());
if(f.isFile())
    System.out.println("it's a file");
else if(f.isDirectory())
    System.out.println("it's a directory");
}

public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();

        File
            old = new File(args[1]),
            rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);

        return; // Sortie de main
    }

    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }

    for( ; count < args.length; count++) {
        File f = new File(args[count]);
        if(f.exists()) {
            System.out.println(f + " exists");

            if(del) {
                System.out.println("deleting..." + f);
                f.delete();
            }
        }
    }
}
```

```

else { // N'existe pas

    if(!del) {

        f.mkdirs();

        System.out.println("created " + f);

    }

}

fileData(f);

}

}

} ///:~

```

Dans **fileData()** vous pourrez voir diverses méthodes d'investigation de fichier employées pour afficher les informations sur le fichier ou sur le chemin du répertoire.

La première méthode pratiquée par **main()** est **renameTo()**, laquelle vous permet de renommer (ou déplacer) un fichier vers un nouveau chemin de répertoire signalé par l'argument, qui est un autre objet **File**. Ceci fonctionne également avec des répertoire de n'importe quelle longueur.

Si vous expérimentez avec le programme ci-dessus, vous découvrirez que vous pouvez créer un chemin de répertoire de n'importe quelle complexité puisque **mkdirs()** s'occupera de tout.

Entrée et sortie

Les bibliothèques d'I/O utilisent souvent l'abstraction de flux [stream], qui représente n'importe quelle source ou réceptacle de données comme un objet capable de produire et de recevoir des morceaux de données. Le flux cache les détails de ce qui arrive aux données à l'intérieur du dispositif actuel.

Les classes de la bibliothèque Java d'I/O sont divisées par entrée et sortie, comme vous pouvez le voir en regardant en ligne la hiérarchie des classes Java avec votre navigateur web. Par héritage, toute dérivées des classes **InputStream** ou **Reader** possèdent des méthodes de base nommées **read()** pour lire un simple byte ou une collection de bytes. De la même manière, toutes les dérivés des classes **OutputStream** ou **Writer** ont des méthodes de base appelées **write()** pour écrire un seul byte ou une série de bytes. Cependant, de manière générale vous n'utiliserez pas ces méthodes; elles existent afin que les autres classes puissent les utiliser — ces autres classes ayant des interfaces plus utiles. Ainsi, vous créerez rarement votre objet stream par l'emploi d'une seule classe, mais au lieu de cela en plaçant les objets ensemble sur plusieurs couches pour arriver à la fonctionnalité désirée. Le fait de créer plus d'un objet pour aboutir à un seul flux est la raison primaire qui rend la bibliothèque de flux Java confuse.

Il est utile de ranger les classes suivant leurs fonctionnalités. Pour Java 1.0, les auteurs de la bibliothèque commencèrent par décider que toutes les classes n'ayant rien à voir avec l'entrée seraient héritées depuis **InputStream** et toutes les classes qui seraient associées avec la sortie seraient héritées depuis **OutputStream**.

Les types d'InputStream

Le boulot d'**InputStream** est de représenter les classes qui produisent l'entrée depuis différentes sources. Ces sources peuvent êtres :

1. Une série de bytes.
2. Un objet **String**.
3. Un fichier.
4. Un tuyau, lequel fonctionne comme un vrai tuyau : vous introduisez des choses à une entrée et elles ressortent de l'autre.
5. Une succession d'autres flux, que vous pouvez ainsi rassembler ensemble dans un seul flux.
6. D'autres sources, comme une connexion internet. (Ceci sera abordé dans un prochain chapitre.)

Chacun d'entre eux possède une sous-classe associée d'**InputStream**. En plus, le **FilterInputStream** est aussi un type d'**InputStream**, fournissant une classe de base pour les classes de « décoration » lesquelles attachent des attributs ou des interfaces utiles aux flux d'entrée. Ceci est abordé plus tard.

Tableau 11-1. Types d'InputStream

Classe	Fonction	Arguments du Constructeur
Mode d'emploi		
ByteArray-InputStream	Autorise un tampon en mémoire pour être utilisé comme InputStream	Le tampon depuis lequel extraire les bytes.
Comme une source de donnée. Connectez le a un objet FilterInputStream pour fournir une interface pratique.		
StringBuffer-InputStream	Convertit un String en un InputStream	Un String . L'implémentation fondamentale utilise actuellement un StringBuffer .
Comme une source de donnée. Connectez le a un objet FilterInputStream pour fournir une interface pratique.		
File-InputStream	Pour lire les information depuis un fichier.	Un String représentant le nom du fichier, ou un objet File ou FileDescriptor .
Comme une source de donnée. Connectez le a un objet FilterInputStream pour fournir une interface pratique.		
Piped-InputStream	Produit la donnée qui sera écrite vers le PipedOutput-Stream associé. Applique le concept de « tuyauterie ».	PipedOutputStream
Comme une source de donnée.		

Connectez le a un objet FilterInputStream pour fournir une interface pratique.		
Sequence-InputStream	Convertit deux ou plusieurs objets InputStream dans seul InputStream .	Deux objets InputStream ou une Énumération pour un récipient d'objets InputStream .
Comme une source de donnée. Connectez le a un objet FilterInputStream pour fournir une interface pratique.		
Filter-InputStream	Classe abstraite qui est une interface pour les décorateurs lesquels fournissent une fonctionnalité profitable aux autres classe InputStream . Voir Tableau 11-3.	Voir Tableau 11-3.
Voir Tableau 11-3.		

Les types d'OutputStream

Cette catégorie contient les classes qui décident de l'endroit où iront vos données de sorties : un tableau de bytes (pas de **String**, cependant; vraisemblablement vous pouvez en créer un en utilisant le tableau de bytes), un fichier, ou un « tuyau. »

En complément, le **FilterOutputStream** fournit une classe de base pour les classes de « décoration » qui attachent des attributs ou des interfaces utiles aux flux de sortie.

Tableau 11-2. Les types d'OutputStream

Classe	Fonction	Arguments du constructeur
Mode d'emploi		
ByteArray-OutputStream	Crée un tampon en mémoire. Toutes les données que vous envoyez vers le flux sont placées dans ce tampon.	En option la taille initiale du tampon.
Pour désigner la destination de vos données. Connectez le à un objet FilterOutputStream pour fournir une interface pratique.		
File-OutputStream	Pour envoyer les informations a un fichier.	Un String représentant le nom d'un fichier, ou un objet File ou FileDescriptor .
Pour désigner la destination de vos données. Connectez le à un objet FilterOutputStream pour fournir une interface pratique.		
Piped-OutputStream	N'importe quelle information que vous écrivez vers celui-ci se termine automatiquement comme une entrée du PipedInput-Stream associé. Applique le concept de « tuyauterie. »	PipedInputStream
Pour indiquer la destination de vos données pour une exécution multiple [multithreading]. Connectez le à un objet FilterOutputStream pour fournir une interface pratique.		
Filter-OutputStream	Classe abstraite qui est une interface pour les décorateurs qui fournissent des fonctionnalités pratiques aux autres classes d' OutputStream . Voir Tableau 11-4.	Voir Tableau 11-4.
Voir Tableau 11-4.		

Ajouter des attributs et des interfaces utiles

L'emploi d'objets en couches pour ajouter dynamiquement et de manière claire des responsabilités aux objets individuels est mentionné comme un Pattern de *Décoration*. (Les Patterns [\[57\]](#) sont le sujet de *Thinking in Patterns with Java*, téléchargeable à <www.BruceEckel.com>.) Le Pattern de décoration précise que tous les objets qui entourent votre objet initial possèdent la même interface. Ceci rend l'usage basique des décorateurs claire — vous envoyez le même message a un objet qu'il soit décoré ou pas. C'est la raison de l'existence des classes « filter » dans la bibliothèque I/O de Java : la classe abstraite « filter » est la classe de base pour tous les décorateurs. (Un décorateur doit avoir la même interface que l'objet qu'il décore, mais le décorateur peut aussi étendre l'interface, ce qui se produit dans un certain nombre de classes « filter »).

Les décorateurs sont souvent employés quand un simple sous-classement touche un grand nombre de sous-classes pour satisfaire toutes les combinaisons possibles nécessaires — avec tellement de sous-classes que cela devient peu pratique. La bibliothèque I/O de Java demande différentes combinaisons de caractéristiques, c'est pourquoi le Pattern de décoration est employé. Il y a un désavantage au Pattern de décoration, néanmoins les décorateurs vous donnent une plus grande flexibilité pendant l'écriture d'un programme (puisque vous pouvez facilement mélanger et assembler des attributs [attributes]), mais ils ajoutent de la complexité à votre code. La raison pour laquelle la bibliothèque d'I/O de Java n'est pas pratique d'emploi est que vous devez créer beaucoup de classes — le type « noyau » d'I/O plus tous les décorateurs — afin d'obtenir le simple objet I/O désiré.

Les classes qui procurent l'interface de décoration pour contrôler un `InputStream` ou `OutputStream` particulier sont **`FilterInputStream`** et **`FilterOutputStream`** — lesquels n'ont pas des noms très intuitifs.

`FilterInputStream` et **`FilterOutputStream`** sont des classes abstraites qui sont dérivées depuis les classes de base de la bibliothèques d'I/O, **`InputStream`** et **`OutputStream`**, qui est l'exigence clef du décorateur (afin qu'il procure une interface commune à tous les objets qui seront Décorés).

Lire depuis un `InputStream` avec `FilterInputStream`

La classe **`FilterInputStream`** accomplit deux choses différentes significatives. **`DataInputStream`** vous permet de lire différents types de données primitives aussi bien que des objets **`String`**. (Toutes les méthodes commencent avec « read, » comme **`readByte()`**, **`readFloat()`**, etc.) Celui-ci, accompagné par **`DataOutputStream`**, vous permet de déplacer des données primitives d'une place à une autre en passant par un flux. Ces « places » sont déterminées par les classes du Tableau 11-1.

Les classes restantes modifient la manière de se comporter en interne d'un **`InputStream`** : s'il est mis en tampon ou pas, si il garde trace des lignes qu'il lit (vous permettant de demander des numéros de ligne ou de régler le numéro de ligne), et si vous pouvez pousser en arrière un caractère seul. Les deux dernières classes ressemblent beaucoup à une ressource pour construire un compilateur (c'est à dire, elles ont été ajoutées en support pour la construction du compilateur Java), donc vous ne l'utiliserez probablement pas en programmation habituelle.

Vous devrez probablement presque tout le temps mettre en tampon votre entrée, sans prendre en compte l'élément d'I/O auquel vous vous connectez, ainsi il aurait été plus sensé de faire un cas spécial (ou un simple appel de méthode) dans la bibliothèque d'I/O pour l'entrée non mise en tampon plutôt que pour l'entrée mise en tampon.

Tableau 11-3. Les types de `FilterInputStream`

Classe	Fonction	Arguments du constructeur
Mode d'emploi		
Data-InputStream	Employé de concert avec DataOutputStream , afin de lire des primitives (int , char , long , etc.) depuis un flux de manière portable.	InputStream
Contient une interface complète vous permettant de lire les types de primitives.		
Buffered-InputStream	Utilisez ceci pour empêcher une lecture physique chaque fois que vous désirez plus de données. Vous dites « Utiliser un tampon. »	InputStream , avec en option la taille du tampon.
Ceci ne fournit pas une interface en soi, mais une condition permettant d'employer le tampon.		
LineNumber-InputStream	Garde trace des numéros de ligne dans le flux d'entrée; vous pouvez appeler getLineNumber() et setLineNumber(int) .	InputStream
Cela ajoute juste la numérotation des lignes, ainsi vous attacherez certainement un objet interface.		
Pushback-InputStream	Possède un tampon retour-chariot d'un byte permettant de pousser le dernier caractère lu en arrière.	InputStream
Généralement employé dans le scanner pour un compilateur et probablement inclus parce qu'il était nécessaire au compilateur Java. Vous ne l'utiliserez probablement pas.		

Écrire vers un OutputStream avec FilterOutputStream

Le complément à **DataInputStream** est **DataOutputStream**, lequel formate chacun des types de primitive et des objets **String** vers un flux de telle sorte que n'importe quel **DataInputStream**, sur n'importe quelle machine, puisse le lire. Toutes les méthodes commencent par « write », comme **writeByte()**, **writeFloat()**, etc.

L'objectif original de **PrintStream** est d'imprimer tous les types de données primitive et objets **String** dans un format visible. Ce qui est différent de **DataOutputStream**, dont le but est de placer les éléments de données dans un flux de manière que **DataInputStream** puisse de façon portable les reconstruire.

Les deux méthodes importantes dans un **PrintStream** sont **print()** et **println()**, qui sont surchargées [overloaded] pour imprimer tous les types différents. La différence entre **print()** et **println()** est que le dernier ajoute une nouvelle ligne une fois exécuté.

PrintStream peut être problématique car il piège toutes les **IOExceptions** (vous devrez tester explicitement le statut de l'erreur avec **checkError()**, lequel retourne **true** si une erreur c'est produite. Aussi, **PrintStream**

n'effectue pas l'internationalisation proprement et ne traite pas les sauts de ligne de manière indépendante de la plate-forme (ces problèmes sont résolus avec **PrintWriter**).

BufferedOutputStream est un modificateur et dit au flux d'employer le tampon afin de ne pas avoir une écriture physique chaque fois que vous écrivez vers le flux. Vous utiliserez vraisemblablement souvent ceci avec les fichiers, et peut-être la console I/O.

Tableau 11-4. Les types de FilterOutputStream

Classe	Fonction	Arguments du Constructeur
Mode d'emploi		
DataOutputStream	Utilisé en concert avec DataInputStream afin d'écrire des primitives (int, char, long, etc.) vers un flux de manière portable.	OutputStream
Contient une interface complète vous permettant d'écrire les types de primitives.		
PrintStream	Pour produire une sortie formatée. Tant que DataOutputStream manie le <i>stockage</i> de données, le PrintStream manie l' <i>affichage</i> .	OutputStream , avec une option boolean indiquant que le tampon est vidé avec chaque nouvelle ligne.
Doit être l'emballage « final » pour votre objet OutputStream . Vous l'utiliserez probablement beaucoup.		
BufferedOutputStream	Utilisez ceci pour prévenir une écriture physique à chaque fois que vous envoyez un morceau de donnée. En disant « Utilise un tampon. » Vous pouvez appeler <code>flush()</code> pour vider le tampon.	OutputStream , avec en option la taille du tampon.
Ceci ne fournit pas une interface en soi, juste l'exigence qu'un tampon soit utilisé. Attache un objet d'interface.		

Lectures & écritures [*Loaders & Writers*]

Java 1.1 apporte quelques modifications significatives à la bibliothèque fondamentale de flux d'I/O (Java 2, cependant, n'apporte pas de modifications fondamentales). Quand vous voyez les classes **Reader** et **Writer** votre première pensée (comme la mienne) doit être que celles-ci ont pour intention de remplacer les classes **InputStream** et **OutputStream**. Mais ce n'est pas le cas. Quoique certains aspects de la bibliothèque originale de flux sont désapprouvés (si vous les employez vous recevrez un avertissement de la part du compilateur), les classes **InputStream** et **OutputStream** fournissent pourtant de précieuses fonctions dans le sens d'un I/O

orienté **byte**, tandis que les classes **Reader** et **Writer** fournissent un E/S se pliant à l'Unicode et sur la base de caractères. En plus :

1. Java 1.1 a ajouté de nouvelles classes dans la hiérarchie d'**InputStream** et **OutputStream**, donc il est évident que ne sont pas remplacées.
2. Il y a des fois où vous devrez employer les classes de la hiérarchie « byte » *en combinaison* avec les classes de la hiérarchie « caractère ». Afin d'accomplir cela il y a des classes « passerelles » : **InputStreamReader** convertit un **InputStream** en un **Reader** et **OutputStreamWriter** convertit un **OutputStream** en un **Writer**.

L'importance des hiérarchies de **Reader** et **Writer** est pour l'internationalisation. L'ancienne hiérarchie de flux d'E/S ne supporte que des flux de byte de 8-bit et ne traite pas bien les caractères Unicode de 16-bit. Depuis qu'Unicode est employé pour l'internationalisation (et les **char** natifs de Java sont en Unicode 16-bit), les hiérarchies de **Reader** et **Writer** ont été ajoutées pour supporter l'Unicode dans toutes les opérations d'E/S. En plus, les nouvelles bibliothèques sont conçues pour des opérations plus rapides que l'ancienne.

Comme il est de coutume dans ce livre, j'aurais aimé fournir une synthèse des classes, mais en supposant que vous utiliserez la documentation en ligne pour éclaircir les détails, comme pour l'exhaustive liste de méthodes.

Les sources et les réceptacles de données

Presque toutes les classes originales de flux d'E/S Java possèdent des classes **Reader** et **Writer** correspondantes afin de fournir une manipulation native en Unicode. Cependant, il y a certains endroits où les **InputStreams** et les **OutputStreams** orientés-**byte** sont la solution correcte; en particulier, les bibliothèques **java.util.zip** sont orientées-**byte** plutôt qu'orientée-**char**. Donc l'approche la plus sage est d'essayer d'utiliser les classes **Reader** et **Writer** chaque fois que c'est possible, et vous découvrirez des situations où il vous faudra employer les bibliothèques orientées-**byte** parce que votre code ne se compilera pas.

Voici un tableau qui montre les correspondances entre les sources et les réceptacles d'informations (c'est à dire, d'où les données physiques viennent ou vont) dans les deux hiérarchies.

Sources & Récipients: Java 1.0 class	Corresponding Java 1.1 class
InputStream	Reader convertisseur : InputStreamReader
OutputStream	Writer convertisseur : OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
(pas de classe correspondante)	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

En général, vous constaterez que les interfaces pour les deux différentes hiérarchies sont semblables sinon identiques.

Modifier le comportement du flux

Pour les **InputStreams** et **OutputStreams**, les flux sont adaptés pour des usages particuliers par l'emploi des sous-classes « décoratives » de **FilterInputStream** et **FilterOutputStream**. La hiérarchie de classe **Reader** et **Writer** poursuit l'usage de ce concept — mais pas exactement.

Dans le tableau suivant, la correspondance est une approximation plus grossière que dans la table précédente. La différence est la cause de l'organisation de la classe : **BufferedOutputStream** est une sous-classe de **FilterOutputStream**, **BufferedWriter** n'est *pas* une sous-classe de **FilterWriter** (laquelle, bien qu'elle soit **abstract**, n'a pas de sous-classe et donc semble avoir été mise dedans de manière à réserver la place ou simplement de manière à ce que vous ne sachiez pas où elle se trouve). Cependant, les interfaces pour les classes sont plutôt un match clos.

Filtres : classe Java 1.0	Classe correspondante en Java 1.1
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (classe abstract sans sous-classe)
BufferedInputStream	BufferedReader (a aussi readLine())
BufferedOutputStream	BufferedWriter
DataInputStream	Utilise DataInputStream (sauf quand vous voulez utiliser readLine() , quand vous devez utiliser un BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer (utilise un constructeur qui prend un Reader à la place)
PushBackInputStream	PushBackReader

Il y a un sens qui est tout à fait clair : Chaque fois que vous voulez utiliser **readLine()**, vous ne devrez plus le faire avec un **DataInputStream** (ceci recevant un message de résiliation [depreciation] au moment de la compilation), mais utiliser à la place un **BufferedReader**. À part cela, **DataInputStream** est pourtant l'élément « préféré » de la bibliothèque d'E/S.

Pour faire la transition vers l'emploi facile d'un **PrintWriter**, il possède des constructeurs qui prennent n'importe quel objet **OutputStream**, aussi bien que des objets **Writer**. Cependant, **PrintWriter** n'a pas plus de support pour formater que ce que faisait **PrintStream** ; les interfaces sont de fait les mêmes.

Le constructeur de **PrintWriter** possède également une option pour effectuer le vidage automatique de la mémoire [automatic flushing], lequel se produit après chaque **println()** si le drapeau du constructeur est levé.

Les classes inchangées

Certaines classes ont été laissées inchangées entre Java 1.0 et Java 1.1 :

Les classes de Java 1.0 qui n'ont pas de classes correspondantes en Java 1.1
DataOutputStream
File
RandomAccessFile
SequenceInputStream

DataOutputStream, en particulier, est utilisé sans changement, donc pour stocker et retrouver des données dans un format transportable vous utiliserez les hiérarchies **InputStream** et **OutputStream**.

Off by itself: RandomAccessFile

RandomAccessFile est employé pour les fichiers dont la taille de l'enregistrement est connue, de sorte que vous pouvez bouger d'un enregistrement à un autre en utilisant **seek()**, puis lire ou changer les enregistrements. Les enregistrements n'ont pas forcément la même taille; vous devez seulement être capable de déterminer de quelle grandeur ils sont et où ils sont placés dans le fichier.

D'abord il est un peu difficile de croire que **RandomAccessFile** ne fait pas partie de la hiérarchie d'**InputStream** ou d'**OutputStream**. Cependant, il n'y a pas d'association avec ces hiérarchies autre que quand il arrive de mettre en œuvre les interfaces **DataInput** et **DataOutput** (qui sont également mises en œuvre par **DataInputStream** et **DataOutputStream**). Elle n'utilise même pas la fonctionnalité des classes existantes **InputStream** et **OutputStream** – il s'agit d'une classe complètement différente, écrite en partant de zéro, avec toutes ses propres méthodes (pour la plupart native). Une raison à cela pouvant être que **RandomAccessFile** a des comportements essentiellement différents des autres types d'E/S, dès qu'il est possible de ce déplacer en avant et en arrière dans un fichier. De toute façon, elle reste seule, comme un descendant direct d'**Object**.

Essentiellement, un **RandomAccessFile** fonctionne comme un **DataInputStream** collé ensemble avec un **DataOutputStream**, avec les méthodes **getFilePointer()** pour trouver où on se trouve dans le fichier, **seek()** pour se déplacer vers un nouvel emplacement dans le fichier, et **length()** pour déterminer la taille maximum du fichier. En complément, les constructeurs requièrent un deuxième argument (identique à **fopen()** en C) indiquant si vous effectuez de manière aléatoire une lecture («**r**») ou une lecture et écriture («**rw**»). Il n'y a pas de ressource pour les fichiers en lecture seule, ce qui pourrait suggérer que **RandomAccessFile** aurait mieux fonctionné si il se trouvait hérité de **DataInputStream**.

Les méthodes de recherche sont valables seulement dans **RandomAccessFile**, qui fonctionne seulement avec des fichiers. Le **BufferedInputStream** permet de marquer «**mark()**» une position (dont la valeur est tenue dans une seule variable interne) et d'annuler cette position «**reset()**», mais c'est limité et pas très pratique.

L'usage typique des flux d'E/S

Bien que vous pouvez combiner les classes de flux d'E/S de différentes manières, vous utiliserez probablement quelques combinaisons. L'exemple suivant pourra être employé comme une référence de base; il montre la création et l'utilisation de configurations d'E/S typiques. Notez que chaque configuration commence par un commentaire avec numéro et titre qui correspondent aux titres de paragraphes suivant et fournissant l'explication approprié.

```
//: c11:IOStreamDemo.java
// Configurations typiques de flux d'E/S.
import java.io.*;

public class IOStreamDemo {
    // Lance les exeptions vers la console :
    public static void main(String[] args)
        throws IOException {
        // 1. Lecture d'entrée par lignes :
        BufferedReader in =
            new BufferedReader(
                new FileReader("IOStreamDemo.java"));
        String s, s2 = new String();
        while((s = in.readLine()) != null)
            s2 += s + "\n";
        in.close();

        // 1b. Lecture d'entrée standard :
        BufferedReader stdin =
            new BufferedReader(
                new InputStreamReader(System.in));
        System.out.print("Enter a line:");
        System.out.println(stdin.readLine());

        // 2. Entrée depuis la mémoire
        StringReader in2 = new StringReader(s2);
        int c;
        while((c = in2.read()) != -1)
            System.out.print((char)c);

        // 3. Entrée de mémoire formatée
        try {
            DataInputStream in3 =
                new DataInputStream(
```

```
        new ByteArrayInputStream(s2.getBytes()));
while(true)
    System.out.print((char)in3.readByte());
} catch(EOFException e) {
    System.err.println("End of stream");
}
```

// 4. Sortie de fichier

```
try {
    BufferedReader in4 =
        new BufferedReader(
            new StringReader(s2));
    PrintWriter out1 =
        new PrintWriter(
            new BufferedWriter(
                new FileWriter("IODemo.out")));
    int lineCount = 1;
    while((s = in4.readLine()) != null )
        out1.println(lineCount++ + ": " + s);
    out1.close();
} catch(EOFException e) {
    System.err.println("End of stream");
}
```

// 5. Stockage et récupération de donnée

```
try {
    DataOutputStream out2 =
        new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
    out2.writeDouble(3.14159);
    out2.writeChars("That was pi\n");
    out2.writeBytes("That was pi\n");
    out2.close();
}
```

```
DataInputStream in5 =
    new DataInputStream(
        new BufferedInputStream(
            new FileInputStream("Data.txt")));
BufferedReader in5br =
    new BufferedReader(
        new InputStreamReader(in5));

// Doit utiliser DataInputStream pour des données :
System.out.println(in5.readDouble());

// Peut maintenant employer le readLine():
System.out.println(in5br.readLine());

// Mais la ligne ressort bizzarement.
// Celle crée avec writeBytes est OK:
System.out.println(in5br.readLine());
} catch (EOFException e) {
    System.err.println("End of stream");
}

// 6. Lecture/écriture par accès aléatoire aux fichiers [Reading/writing random access
RandomAccessFile rf =
    new RandomAccessFile("rtest.dat", "rw");
for(int i = 0; i < 10; i++)
    rf.writeDouble(i*1.414);
rf.close();

rf =
    new RandomAccessFile("rtest.dat", "rw");
rf.seek(5*8);
rf.writeDouble(47.0001);
rf.close();

rf =
    new RandomAccessFile("rtest.dat", "r");
for(int i = 0; i < 10; i++)
```

```

        System.out.println(
            "Value " + i + ": " +
            rf.readDouble());
    rf.close();
}
} ///:~

```

Voici les descriptions pour les sections numérotées du programme :

Flux d'Entrée

La partie 1 à 4 démontre la création et l'utilisation des flux d'entrée. La partie 4 montre aussi l'emploi simple d'un flux de sortie.

1. Entrée en tampon du fichier [buffered input file]

Afin d'ouvrir un fichier pour l'entrée de caractère, on utilise un **FileInputStream** avec un objet **String** ou **File** comme nom de fichier. Pour la vitesse, vous voudrez que le fichier soit mit en mémoire tampon alors vous passerez la référence résultante au constructeur à un **BufferedReader**. Puisque **BufferedReader** fournit aussi la méthode **readLine()**, qui est votre objet final et l'interface depuis laquelle vous lisez. Quand vous cherchez la fin du fichier, **readLine()** renvoie null ce qui est utilisé pour sortir de la boucle **while**.

Le **String s2** est utilisé pour accumuler le contenu entier du fichier (incluant les nouvelles lignes qui doivent être ajoutées puisque **readLine()** les enlève). **s2** est ensuite employé dans la dernière partie de se programme. Enfin, **close()** est appelé pour fermer le fichier. Techniquement, **close()** sera appelé au lancement de **finalize()**, et ceci est supposé se produire (que le garbage collector se mette en route ou pas) lors de la fermeture du programme. Cependant, ceci a été inconséquemment implémenté, c'est pourquoi la seule approche sûre est d'appeler explicitement **close()** pour les fichiers.

La section 1b montre comment envelopper **System.in** afin de lire l'entrée sur la console. **System.in** est un **DataInputStream** et **BufferedReader** nécessite un argument **Reader**, voilà pourquoi **InputStreamReader** est introduit pour effectuer la traduction.

2. Entrée depuis la mémoire

Cette partie prend le **String s2** qui contient maintenant le contenu entier du fichier et l'utilise pour créer un **StringReader**. Puis **read()** est utilisé pour lire chaque caractère un par un et les envoie vers la console. Notez que **read()** renvoie le byte suivant sous la forme d'un **int** et pour cette raison il doit être convertit en **char** afin de s'afficher correctement.

3. Entrée de mémoire formatée

Pour lire une donnée « formatée », vous utiliserez un **DataInputStream**, qui est une classe d'E/S orientée-byte (plutôt qu'orientée-char). Ainsi vous devrez utiliser toutes les classes **InputStream** plutôt que les classes **Reader**. Bien sur, vous pouvez lire n'importe quoi (du genre d'un fichier) comme des bytes en utilisant les classes **InputStream**, mais ici c'est un **String** qui est utilisé. Pour convertir le **String** en un tableau de bytes, ce qui est approprié pour un **ByteArrayInputStream**, **String** possède une méthode **getBytes()** pour faire le

travail. A ce stade, vous avez un **InputStream** adéquat pour porter un **DataInputStream**.

Si on lit les caractères depuis un `DataInputStream` un byte à chaque fois en utilisant `readByte()`, n'importe quelle valeur de byte donne un résultat juste donc la valeur de retour ne peut pas être employée pour détecter la fin de l'entrée. À la place, on peut employer la méthode `available()` pour découvrir combien de caractères sont encore disponibles. Voici un exemple qui montre comment lire un fichier byte par byte :

```
//: c11:TestEOF.java
// Test de fin de fichier
// En lisant un byte a la fois.
import java.io.*;

public class TestEOF {
    // Lance les exeptions vers la console :
    public static void main(String[] args)
        throws IOException {
        DataInputStream in =
            new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("TestEof.java")));
        while(in.available() != 0)
            System.out.print((char)in.readByte());
    }
} ///:~
```

Notons qu'**available()** fonctionne différemment en fonction du type de ressource depuis laquelle on lit; c'est littéralement « le nombre de bytes qui peuvent être lus *sans blocage*. » Avec un fichier cela signifie le fichier entier, mais avec une autre sorte d'entrée cela ne pourra pas être possible, alors employez le judicieusement.

On peut aussi détecter la fin de l'entrée dans des cas comme cela en attrapant une exception. Cependant, l'emploi des exeptions pour le contrôle du flux est considéré comme un mauvais emploi de cette caractéristique.

4. Sortie de Fichier

Cet exemple aussi montre comment écrire des données vers un fichier. Premièrement, un **FileWriter** est crée pour se connecter au fichier. Vous voudrez toujours mettre en tampon la sortie en l'emballant[wrapping it] dans un **BufferedWriter** (essayez de retirer cet emballage pour voir l'impact sur les performances – le tampon tend à accroître dramatiquement les performance des opérations d'E/O). Puis le formatage est changé en un `PrintWriter`. Le fichier de données ainsi crée est lisible comme un fichier texte normal.

Comme les lignes sont écrites vers le fichier, les numéros de lignes sont ajoutés. Notez que **LineNumberInputStream** n'est *pas* utilisé, parce que c'est une classe idiote et que vous n'en avez pas besoin.

Comme il est montré ici, il est superficiel de garder trace de vos propres numéros de lignes.

Quand le flux d'entrée épuisé, **readLine()** renvoie null. Vous verrez **close()** explicite pour **out1**, car si vous ne faites pas appel à **close()** pour tous vos fichiers de sortie, vous pourrez découvrir que les tampons ne seront pas libérés sans qu'ils seront incomplets.

Flux de sortie

Les deux types de flux de sortie sont séparés par la manière dont ils écrivent les données : un les écrit pour une consommation humaine, l'autre les écrit pour une reacquisition par un **DataInputStream**. Le **RandomAccessFile** se tient seul, bien que son format de données soit compatible avec **DataInputStream** et **DataOutputStream**.

5. Stocker et récupérer des données

Un **PrintWriter** formate les données afin qu'elles soient lisibles par un humain. Cependant, pour sortir des données qui puissent être récupérées par un autre flux, on utilise un **DataOutputStream** pour écrire les données et un **DataInputStream** pour récupérer les données. Bien sûr, ces flux pourraient être n'importe quoi, mais ici c'est un fichier qui est employé, mis en mémoire tampon pour à la fois lire et écrire. **DataOutputStream** et **DataInputStream** sont orientés-byte et nécessitent ainsi des **InputStreams** and **OutputStreams**.

Si vous employez un **DataOutputStream** pour écrire les données, alors Java se porte garant de l'exacte récupération des données en employant un **DataInputStream** – sans se soucier du type de plate-forme qui écrit et lit les données. Ce qui est incroyablement valable, comme chacun sait ayant passé du temps à s'inquiéter de la distribution de donnée à des plates-formes spécifiques. Ce problème disparaît si l'on a Java sur les deux plates-formes [\[58\]](#).

Notez que les caractères de la chaîne de caractère sont écrit en utilisant à la fois **writeChars()** et **writeBytes()**. Quand vous exécuterez le programme, vous découvrirez que **writeChars()** donne en sortie des caractères Unicode 16-bit. Lorsque l'on lit la ligne avec **readLine()**, vous remarquerez qu'il y a un espace entre chaque caractère, à cause du byte (ndt : octet ?) supplémentaire inséré par Unicode. Comme il n'y a pas de méthode complémentaire « **readChars** » dans **DataInputStream**, vous êtes coincés à retirer ces caractères un par un avec **readChar()**. Ainsi pour l'ASCII, il est plus facile d'écrire les caractères sous la forme de bytes suivit par un saut de ligne; employez alors **readLine()** pour relire les bytes comme des lignes régulières ASCII.

Le **writeDouble()** stocke les nombres **double** pour le flux et le **readDouble()** complémentaire les récupère (il y a des méthodes similaires pour lire et écrire les autres types). Mais pour que n'importe quelle méthode de lecture fonctionne correctement, vous devrez connaître l'emplacement exact des éléments de donnée dans le flux, puisqu'il serait possible de lire les **double** stockés comme de simple séquences de bytes, ou comme des **chars**, etc. Donc vous devrez soit avoir un format fixé pour les données dans le fichier ou des informations supplémentaires devront être stockés dans le fichier et que vous analyserez pour déterminer l'endroit où les données sont stockées.

6. Accès aléatoire en lecture et écriture aux fichiers

Comme il a été noté précédemment, le **RandomAccessFile** est presque totalement isolé du reste de la hiérarchie d'E/S, protégé par le fait qu'il implémente les interfaces **DataInput** et **DataOutput**. Donc vous ne pouvez l'associer avec un des point des sous-classes **InputStream** et **OutputStream**. Quoiqu'il pourrait sembler raisonnable de traiter un **ByteArrayInputStream** comme un élément d'accès aléatoire, vous pouvez employer un **RandomAccessFile** pour ouvrir simplement un fichier. Vous devez supposer qu'un **RandomAccessFile** est correctement mis en mémoire tampon puisque vous ne pouvez pas ajouter cela.

La seule option disponible est dans le second argument du constructeur : vous pouvez ouvrir un **RandomAccessFile** pour lire (« **r** ») ou lire et écrire (« **rw** »).

Utiliser un **RandomAccessFile** est comme utiliser une combinaison de **DataInputStream** et **DataOutputStream** (parce que cela implémente les interfaces équivalentes). En plus, vous pouvez remarquer que **seek()** est utilisé pour errer dans le fichier et changer une des valeurs.

Un bogue ?

Si vous regardez la partie 5, vous verrez que les données sont écrites *avant* le texte. C'est à cause d'un problème qui a été introduit dans Java 1.1 (et persiste dans Java 2) qui apparaît vraiment comme un bogue pour moi, mais j'en ai rendu compte et les débogueurs de JavaSoft ont dit que c'était la manière dont il était supposé fonctionner (pourtant, le problème n'apparaissait pas dans Java 1.0, ce qui me rend suspicieux). Le problème est montré dans le code suivant :

```
//: c11:IOProblem.java
// Problème dans Java 1.1 et supérieur.

import java.io.*;

public class IOProblem {
    // Lance les exeptions vers la console :
    public static void main(String[] args)
        throws IOException {
        DataOutputStream out =
            new DataOutputStream(
                new BufferedOutputStream(
                    new FileOutputStream("Data.txt")));

        out.writeDouble(3.14159);
        out.writeBytes("C'était la valeur de pi\n");
        out.writeBytes("C'est pi/2:\n");
        out.writeDouble(3.14159/2);
        out.close();

        DataInputStream in =
            new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("Data.txt")));

        BufferedReader inbr =
```



```

        new BufferedReader(
            new InputStreamReader(in));

    // Les doubles écrit AVANT la ligne de texte
    // sont renvoyés correctement :

    System.out.println(in.readDouble());

    // Lit le lignes du texte :

    System.out.println(inbr.readLine());
    System.out.println(inbr.readLine());

    // Tenter de lire les doubles après la ligne
    // produit une eexception de fin de ligne :

    System.out.println(in.readDouble());
}
} ///:~

```

Il apparaît que tout ce que vous écrivez après un appel à **writeBytes()** n'est pas récupérable. La réponse est apparemment la même que la réponse à la vieille blague de vaudeville : « Docteur, cela fait mal quand je fais cela ! » « Ne fait pas cela ! ».

Flux Piped

Les **PipedInputStream**, **PipedOutputStream**, **PipedReader** et **PipedWriter** sont mentionnés de manière brève dans ce chapitre. Ce qui n'insinue pas qu'il ne sont pas utiles, mais leur importance n'est pas évidente jusqu'à ce que vous ayez commencé à comprendre le multithreading, étant donné que les flux piped sont employés pour communiquer entre les threads. Ceci est abordé avec un exemple au chapitre 14.

Standard E/S

Le terme d'*E/S standard* se réfère au concept d'Unix (qui est reproduit sous une certaine forme dans Windows et bien d'autres systèmes d'exploitations) d'un simple flux d'information qui est utilisé par un programme. Toutes les entrées du programme peuvent provenir d'une *entrée standard*, toutes ses sorties peuvent aller vers une *sortie standard*, et tous les messages d'erreur peuvent être envoyés à une *erreur standard*. L'importance de l'E/S standard est que le programme peut être facilement mis en chaîne simultanément et la sortie standard d'un programme peut devenir l'entrée standard pour un autre programme. C'est un outil puissant.

Lire depuis une entrée standard

Suivant le modèle d'E/S standard, Java possède **System.in**, **System.out**, et **System.err**. Tout au long de ce livre vous avez vu comment écrire vers une sortie standard en utilisant **System.out**, qui est déjà pré-enveloppé comme un objet **PrintStream**. **System.err** est semblable à **PrintStream**, mais **System.in** est un **InputStream** brut, sans emballage. Ceci signifie que bien que vous pouvez utiliser **System.out** et **System.err** immédiatement, **System.in** doit être enveloppé avant de pouvoir lire depuis celui-ci.

Typiquement, vous désirerez lire l'entrée une ligne à la fois en utilisant **readLine()**, donc vous devrez

envelopper **System.in** dans un **BufferedReader**. Pour cela, vous devrez convertir **System.in** en un **Reader** par l'usage d'**InputStreamReader**. Voici un exemple qui fait simplement écho de chaque ligne que vous tapez :

```
//: c11:Echo.java
// Comment lire depuis l'entrée standard.

import java.io.*;

public class Echo {

    public static void main(String[] args)

        throws IOException {

        BufferedReader in =

            new BufferedReader(

                new InputStreamReader(System.in));

        String s;

        while((s = in.readLine()).length() != 0)

            System.out.println(s);

        // Une ligne vide met fin au programme.

    }

} ///:~
```

Le sens de l'instruction d'exception est que **readLine()** peut lancer une **IOException**. Notez que pourra généralement être mit en tampon, comme avec la majorité des flux

Modifier System.out en un PrintWriter

System.out est un **PrintStream**, qui est un **OutputStream**. **PrintWriter** a un constructeur qui prend un **OutputStream** comme argument. Ainsi, si vous le désirez vous pouvez convertir **System.out** en un **PrintWriter** en utilisant ce constructeur :

```
//: c11:ChangeSystemOut.java
// Tourne System.out en un PrintWriter.

import java.io.*;

public class ChangeSystemOut {

    public static void main(String[] args) {

        PrintWriter out =

            new PrintWriter(System.out, true);

        out.println("Hello, world");

    }

}
```

```

    }
} ///:~

```

Il est important d'utiliser la version à deux arguments du constructeur **PrintWriter** et de fixer le deuxième argument à **true** afin de permettre un vidage automatique, sinon vous ne verriez pas la sortie.

Réorienter l'E/S standard

La classe Java **System** vous permet de rediriger l'entrée, la sortie, et l'erreur standard des flux d'E/S en employant un simple appel aux méthodes statiques :

```

setIn(InputStream)
setOut(PrintStream)
setErr(PrintStream)

```

Réorienter la sortie est particulièrement utile si vous commencez soudainement à créer une grande quantité de sortie sur l'écran et qu'il défile jusqu'à la fin plus vite que vous ne pouvez le lire. [\[59\]](#) Réorienter l'entrée est précieux pour un programme en ligne de commande dans lequel vous désirez tester un ordre d'entrée-utilisateur particulier à plusieurs reprises. Voici un exemple simple qui montre l'utilisation de ces méthodes :

```

//: c11\Redirecting.java
// Demonstration de reorientation d'E/S standard.

import java.io.*;

class Redirecting {
    // Lance les exeptions vers la console :

    public static void main(String[] args)
        throws IOException {

        BufferedInputStream in =
            new BufferedInputStream(
                new FileInputStream(
                    "Redirecting.java"));

        PrintStream out =
            new PrintStream(
                new BufferedOutputStream(
                    new FileOutputStream("test.out")));

        System.setIn(in);

        System.setOut(out);

        System.setErr(out);
    }
}

```

```
BufferedReader br =  
    new BufferedReader(  
        new InputStreamReader(System.in));  
  
String s;  
  
while((s = br.readLine()) != null)  
    System.out.println(s);  
  
out.close(); // Rappelez-vous de ça !  
  
}  
} ///:~
```

Ce programme attache la sortie standard à un fichier, et redirige la sortie standard et l'erreur standard vers un autre fichier.

La redirection d'E/S manipule les flux de bytes, mais pas les flux de caractères, ainsi **InputStreams** et **OutputStreams** sont plus utilisés que les **Readers** et **Writers**.

Compression

La librairie (ndt : ou bibliothèque) d'E/S Java contient des classes pour supporter la lecture et l'écriture de flux dans des formats compressés. Ceux-ci sont enveloppés autour des classes existantes d'E/S pour fournir des fonctionnalités de compression.

Ces classes ne sont pas dérivée des classes **Reader** et **Writer**, mais à la place font partie des hiérarchies d'**InputStream** et **OutputStream**. Ceci parce que la librairie de compression fonctionne avec des bytes, pas des caractères. Cependant, vous serez parfois forcés de mixer les deux types de fluxs. (Rappelez-vous que vous pouvez utiliser **InputStreamReader** et **OutputStreamWriter** pour fournir une conversion facile entre un type et un autre.)

Classe de compression	
Fonction	
CheckedInputStream	GetChecksum() fait une checksum (vérification du nombre de bits transmis afin de deceler des erreurs de transmission) pour n'importe quel InputStream (non pas une simple décompression).
CheckedOutputStream	GetChecksum() fait une checksum pour n'importe quel OutputStream (non pas une simple compression).
DeflaterOutputStream	Classe de base pour les classes de compression.
ZipOutputStream	Un DeflaterOutputStream qui compresse les données au format Zip.
GZIPOutputStream	Un DeflaterOutputStream qui compresse les données au format GZIP.
InflaterInputStream	Classe de base pour les classes de décompression.
ZipInputStream	Un InflaterInputStream qui décompresse les données qui sont stockées au format Zip.
GZIPInputStream	Un InflaterInputStream qui décompresse les données qui sont stockées au format GZIP.

Bien qu'il y ait de nombreux algorithmes de compression, Zip et GZIP sont peut-être ceux employés le plus couramment. Ainsi vous pouvez facilement manipuler vos données compressées avec les nombreux outils disponibles pour écrire et lire ces formats.

Compression simple avec GZIP

L'interface GZIP est simple et est ainsi la plus appropriée quand vous avez un simple flux de donnée que vous désirez compresser (plutôt qu'un récipient (container) de pièces différentes de données. Voici un exemple qui compresses un simple fichier :

```

//: c11:GZIPcompress.java

// Utilise la compression GZIP pour compresser un fichier
// dont le nom est passé en ligne de commande.

import java.io.*;

import java.util.zip.*;

public class GZIPcompress {

    // Lance les exceptions vers la console :

    public static void main(String[] args)
        throws IOException {

        BufferedReader in =

            new BufferedReader(

                new FileReader(args[0]));

        BufferedOutputStream out =

```

```

        new BufferedOutputStream(
            new GZIPOutputStream(
                new FileOutputStream("test.gz"))));
    System.out.println("Writing file");

    int c;

    while((c = in.read()) != -1)
        out.write(c);

    in.close();
    out.close();

    System.out.println("Reading file");

    BufferedReader in2 =
        new BufferedReader(
            new InputStreamReader(
                new GZIPInputStream(
                    new FileInputStream("test.gz"))));

    String s;

    while((s = in2.readLine()) != null)
        System.out.println(s);
    }
} ///:~

```

L'emploi des classes de compression est simple – vous enveloppez simplement votre flux de sortie dans un **GZIPOutputStream** ou un **ZipOutputStream** et votre flux d'entrée dans un **GZIPInputStream** ou un **ZipInputStream**. Tout le reste étant de l'écriture normale d'entrée et de sortie. C'est un exemple de mélange de flux orientés-**char** avec des flux orientés-**byte** : **in** utilise la classe **Reader**, vu que le constructeur de **GZIPOutputStream** peut seulement accepter un objet **OutputStream**, et non pas un objet **Writer**. Quand le fichier est ouvert, le **GZIPInputStream** est convertit en un **Reader**.

Stockage de fichiers multiples avec Zip

La librairie qui supporte le format Zip est bien plus vaste. Avec elle vous pouvez facilement stocker des fichiers multiples, et il y a même une classe séparée pour amener le procédé de lecture d'un fichier Zip simple. La librairie utilise le format Zip standard de manière à ce qu'il fonctionne avec tous les outils couramment téléchargeables sur l'Internet. L'exemple suivant prend la même forme que l'exemple précédent, mais il manipule autant d'arguments de ligne de commande que vous le désirez. En plus, il met en valeur l'emploi de la classe **Checksum** pour calculer et vérifier la somme de contrôle [checksum] pour le fichier. Il y a deux sortes de **Checksum** : **Adler32** (qui est rapide) et **CRC32** (qui est plus lent mais légèrement plus précis).

```

///: c11:ZipCompress.java

// Emploi de la compression Zip pour compresser n'importe quel

```

```
// nombre de fichiers passés en ligne de commande.

import java.io.*;

import java.util.*;

import java.util.zip.*;

public class ZipCompress {

    // Lance les exeptions vers la console :

    public static void main(String[] args)

    throws IOException {

        FileOutputStream f =

            new FileOutputStream("test.zip");

        CheckedOutputStream csum =

            new CheckedOutputStream(

                f, new Adler32());

        ZipOutputStream out =

            new ZipOutputStream(

                new BufferedOutputStream(csum));

        out.setComment("A test of Java Zipping");

        // Pas de getComment() correspondant, bien que.

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

            System.out.println(

                "Writing file " + args[i]);

            BufferedReader in =

                new BufferedReader(

                    new FileReader(args[i]));

            out.putNextEntry(new ZipEntry(args[i]));

            int c;

            while((c = in.read()) != -1)

                out.write(c);

            in.close();

        }

        out.close();

        // Validation de Checksum seulement après que

        // le fichier est été fermé !
    }
}
```

```
System.out.println("Checksum: " +
    csum.getChecksum().getValue());
// Maintenant extrait les fichiers :
System.out.println("Reading file");
FileInputStream fi =
    new FileInputStream("test.zip");
CheckedInputStream csumi =
    new CheckedInputStream(
        fi, new Adler32());
ZipInputStream in2 =
    new ZipInputStream(
        new BufferedInputStream(csumi));
ZipEntry ze;
while((ze = in2.getNextEntry()) != null) {
    System.out.println("Reading file " + ze);
    int x;
    while((x = in2.read()) != -1)
        System.out.write(x);
}
System.out.println("Checksum: " +
    csumi.getChecksum().getValue());
in2.close();
// Méthode alternative pour ouvrir et lire
// les fichiers zip :
ZipFile zf = new ZipFile("test.zip");
Enumeration e = zf.entries();
while(e.hasMoreElements()) {
    ZipEntry ze2 = (ZipEntry)e.nextElement();
    System.out.println("File: " + ze2);
    // ... et extrait les données comme précédemment.
}
}
} ///:~
```


Pour chaque fichier à ajouter à l'archive, vous devez appeler **putNextEntry()** et lui passer un objet **ZipEntry**. L'objet **ZipEntry** contient une interface extensible qui vous permet d'obtenir et de positionner toutes les données disponibles sur cette entrée précise dans votre fichier Zip : nom, tailles compressé et non-compressé, date, somme de contrôle CRC, données supplémentaires, commentaire, méthode de compression, et si il s'agit d'une entrée de répertoire. Toutefois, même si le format Zip possède une méthode pour établir un mot de passe, il n'est pas supporté dans la librairie Zip de Java. Et bien que **CheckedInputStream** et **CheckedOutputStream** supportent les deux contrôles de somme **Adler32** et **CRC32**, la classe **ZipEntry** supporte seulement une interface pour la CRC (Contrôle de Redondance Cyclique). C'est une restriction sous-jacente du format Zip, mais elle pourrait vous limiter d'utiliser l'**Adler32** plus rapide.

Pour extraire les fichiers, **ZipInputStream** a une méthode **getNextEntry()** qui renvoie la **ZipEntry** suivante si il y en a une. Comme alternative plus succincte, vous pouvez lire le fichier en utilisant un objet **ZipFile**, lequel possède une méthode **entries()** pour renvoyer une **Enumeration** au **ZipEntries**.

Afin de lire la somme de contrôle vous devrez d'une manière ou d'une autre avoir accès à l'objet **Checksum** associé. Ici, une référence vers les objets **CheckedOutputStream** et **CheckedInputStream** est retenue, mais vous pourriez aussi juste vous en tenir à une référence à l'objet **Checksum**.

Une méthode déconcertante dans les flux de Zip est **setComment()**. Comme montré plus haut, vous pouvez établir un commentaire lorsque vous écrivez un fichier, mais il n'y a pas de manière pour récupérer le commentaire dans le **ZipInputStream**. Les commentaires sont apparemment complètement supportés sur une base d'entrée-par-entrée par l'intermédiaire de **ZipEntry**.

Bien sûr, vous n'êtes pas limité aux fichiers lorsque vous utilisez les librairies **GZIP** et **Zip** – vous pouvez compresser n'importe quoi, y compris les données à envoyer en passant par une connexion réseau.

ARchives Java (JARs)

Le format Zip est aussi employé dans le format de fichier JAR (Java ARchive), qui est une manière de rassembler un groupe de fichiers dans un seul fichier compressé, tout à fait comme Zip. Cependant, comme tout le reste en Java, les fichiers JAR sont multi-plate-forme donc vous n'avez pas à vous soucier des distributions de plate-forme. Vous pouvez aussi inclure des fichiers audio et image en plus des fichiers class.

Les fichiers JAR sont particulièrement utile quand on a affaire à l'Internet. Avant les fichiers JAR, votre navigateur Web devait faire des requêtes répétées sur un serveur Web afin de télécharger tous les fichiers qui composaient une applet. En plus, chacun de ces fichiers n'était pas compressé. En combinant tous les fichiers d'une applet précise dans un seul fichier JAR, une seule requête au serveur est nécessaire et le transfert est plus rapide en raison de la compression. Et chaque entrée dans un fichier JAR peut être signée digitalement pour la sécurité (se référer à la documentation de Java pour les détails).

Un JAR consiste en un seul fichier contenant une collection de fichiers zippés ensemble avec un « manifeste » qui en fait la description. (Vous pouvez créer votre propre fichier manifeste; sinon le programme **jar** le fera pour vous.) Vous pouvez trouver plus de précisions sur les manifestes dans la documentation HTML du JDK.

L'utilitaire **jar** qui est fourni avec le JDK de Sun compresses automatiquement les fichiers de votre choix. Vous lui faites appel en ligne de commande :

```
jar [options] destination [manifest] inputfile(s)
```

Les options sont simplement une collection de lettres (aucun trait d'union ou autre indicateur n'est nécessaire).

Les utilisateurs noterons la similitude avec les options **tar**. Celles-ci sont :

c	Crée une archive nouvelle ou vide.
t	Établit la table des matières.
x	Extrait tous les fichiers.
x file	Extrait le fichier nommé.
f	Dit : « Je vais vous donner le nom du fichier. » Si vous n'utilisez pas ceci, jar considère que sont entrée viendra de l'entrée standard, ou , si il crée un fichier, sa sortie ira vers la sortie standard.
m	Dit que le premier argument sera le nom du fichier manifeste crée par l'utilisateur.
v	Génère une sortie « verbose » décrivant ce que jar effectue.
0	Stocke seulement les fichiers; ne compresse pas les fichiers (utilisé pour créer un fichier JAR que l'on peut mettre dans le classpath).
M	Ne crée pas automatiquement un fichier manifeste.

Si un sous-répertoire est inclus dans les fichiers devant être placés dans le fichier JAR, ce sous-répertoire est ajouté automatiquement, incluant tous ces sous-répertoire, etc. Les informations de chemin sont ainsi préservées.

Voici quelques façon typiques d'invoquer **jar** :

```
jar cf myJarFile.jar *.class
```

Ceci crée un fichier JAR appelé **myJarFile.jar** qui contient tous les fichiers class du répertoire courant, avec la génération automatique d'un fichier manifeste.

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

Comme l'exemple précédent, mais ajoute un fichier manifeste crée par l'utilisateur nommé **myManifestFile.mf**.

```
jar tf myJarFile.jar
```

Produit une table des matières des fichiers dans **myJarFile.jar**.

```
jar tvf myJarFile.jar
```

Ajoute le drapeau « verbose » pour donner des informations plus détaillées sur les fichiers dans **myJarFile.jar**.

```
jar cvf myApp.jar audio classes image
```

Supposant que **audio**, **classes**, et **image** sont des sous-répertoires, ceci combine tous les sous-répertoires dans le fichier **myApp.jar**. Le drapeau « verbose » est aussi inclus pour donner contrôle d'information supplémentaire pendant que le programme **jar** travaille.

Si vous créez un fichier JAR en utilisant l'option **-o**, ce fichier pourra être placé dans votre CLASSPATH :

```
CLASSPATH="lib1.jar;lib2.jar;"
```

Ainsi Java pourra chercher dans **lib1.jar** et **lib2.jar** pour trouver des fichiers class.

L'outil **jar** n'est pas aussi utile que l'utilitaire **zip**. Par exemple, vous ne pouvez ajouter ou mettre à jour un fichier JAR existant; vous pouvez créer des fichiers JAR seulement à partir de zéro. Aussi, vous ne pouvez déplacer les fichiers dans un fichier JAR, les effaçant dès qu'ils sont déplacés. Cependant un fichier JAR créé sur une plate-forme sera lisible de manière transparente par l'outil **jar** sur n'importe quelle autre plate-forme (un problème qui apparaît parfois avec l'utilitaire **zip**).

Comme vous le verrez dans le chapitre 13, les fichiers JAR sont aussi utilisés pour emballer les JavaBeans.

La sérialisation objet

La *sérialisation objet* en Java vous permet de prendre n'importe quel objet qui implémente l'interface **Serializable** et le dévie en une séquence de bytes qui pourront ensuite être complètement restaurés pour régénérer l'objet original. C'est même vrai à travers un réseau, ce qui signifie que le mécanisme de sérialisation compense automatiquement des différences dans les systèmes d'exploitation. C'est à dire, vous pouvez créer un objet sur une machine Windows, le sérialiser, et l'envoyer à travers le réseau sur une machine Unix où il sera correctement reconstruit. Vous n'avez pas à vous soucier de la représentation des données sur les différentes machines, l'ordonnancement des bytes, ou tout autres détails.

Par elle-même, la sérialisation objet est intéressante parce qu'elle vous permet de mettre en application la *persistance légère* [lightweight persistence]. Rappelez-vous que la persistance signifie que la durée de vie de l'objet n'est pas déterminée tant qu'un programme s'exécute – l'objet vit *dans l'intervalle* des invocations du programme. En prenant un objet sérialisable et en l'écrivant sur le disque, puis en ressortant cet objet lors de la remise en route du programme, vous êtes alors capable de produire l'effet de persistance. La raison pour laquelle elle est appelée « légère » est que vous pouvez simplement définir un objet en employant un certain type de mot-clé pour la « persistance » et de laisser le système prendre soin des détails (bien que cela peut bien arriver dans le futur). À la place de cela, vous devrez sérialiser et désérialiser explicitement les objets dans votre programme.

La sérialisation objet a été ajoutée au langage pour soutenir deux caractéristiques majeures. La *remote method invocation* (RMI) de Java permet aux objets qui vivent sur d'autres machines de se comporter comme si ils vivaient sur votre machine. Lors de l'envoi de messages aux objets éloignés, la sérialisation d'objet est nécessaire pour transporter les arguments et les valeurs retournées. RMI est abordé au Chapitre 15.

La sérialisation des objets est aussi nécessaire pour les JavaBeans, décrit au Chapitre 13. Quand un Bean est utilisé, son information d'état est généralement configuré au moment de la conception. Cette information d'état doit être stockée et récupérée ultérieurement quand le programme est démarré; la sérialisation objet accomplit cette tâche.

Sérialiser un objet est assez simple, aussi longtemps que l'objet implémente l'interface **Serializable** (cette interface est juste un drapeau et n'a pas de méthode). Quand la sérialisation est ajoutée au langage, de nombreuses classes sont changées pour les rendre sérialisables, y compris tous les enveloppeurs [wrappers] pour les types de primitives, toutes les classes de récipients [container], et bien d'autres. Même les objets **Class** peuvent être sérialisés. (Voir le Chapitre 12 pour ce que cela implique.)

Pour sérialiser un objet, vous créez une sorte d'objet **OutputStream** et l'enveloppez ensuite dans un objet **ObjectOutputStream**. À ce point vous avez seulement besoin d'appeler **writeObject()** et votre objet est sérialisé et envoyé à l'**OutputStream**. Pour inverser le processus, vous enveloppez un **InputStream** dans un **ObjectInputStream** et appelez **readObject()**. Ce qui renvoie, comme d'habitude, une référence à un **Objet** dont on a sur-forcé le type [upcast], ainsi vous devrez sous-forcer pour préparer les objets directement.

Un aspect particulièrement astucieux de la sérialisation objet est qu'elle ne sauve pas uniquement une image de votre objet, mais cela s'occupe aussi de toutes les références contenues dans votre objet et sauve *ces* objets, et poursuit dans toutes les références de ces objets, etc. Ceci est parfois rapporté comme le « Web des objets » auquel un simple objet peut être connecté, et il comprend des tableaux de références aux objets aussi bien que d'objets membres. Si vous devez entretenir votre propre schéma de sérialisation, entretenir le code pour suivre tous ces liens serait un peu un casse-tête. Pourtant, la sérialisation d'objet Java semble s'en sortir sans faute, sans aucun doute en utilisant un algorithme optimisé qui traverse le Web des objets. L'exemple suivant teste le mécanisme de sérialisation en créant un « vers » d'objets liés, chacun d'entre eux ayant un lien jusqu'au prochain segment dans le vers en plus d'un tableau de références aux objets d'une classe différent, **Data** :

```
//: c11:Worm.java
// Demontre la sérialisation des objets.

import java.io.*;

class Data implements Serializable {
    private int i;

    Data(int x) { i = x; }

    public String toString() {
        return Integer.toString(i);
    }
}

public class Worm implements Serializable {
    // Génère une valeur d'int aléatoire :

    private static int r() {
        return (int)(Math.random() * 10);
    }

    private Data[] d = {
        new Data(r()), new Data(r()), new Data(r())
    };

    private Worm next;

    private char c;

    // Value de i == nombre de segments
```

```
Worm(int i, char x) {
    System.out.println(" Worm constructor: " + i);
    c = x;
    if(--i > 0)
        next = new Worm(i, (char)(x + 1));
}

Worm() {
    System.out.println("Default constructor");
}

public String toString() {
    String s = ":" + c + "(";
    for(int i = 0; i < d.length; i++)
        s += d[i].toString();
    s += ")";
    if(next != null)
        s += next.toString();
    return s;
}

// Lance les exeptions vers la console :
public static void main(String[] args)
throws ClassNotFoundException, IOException {
    Worm w = new Worm(6, 'a');
    System.out.println("w = " + w);
    ObjectOutputStream out =
        new ObjectOutputStream(
            new FileOutputStream("worm.out"));
    out.writeObject("Worm storage");
    out.writeObject(w);
    out.close(); // Vide aussi la sortie
    ObjectInputStream in =
        new ObjectInputStream(
            new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
}
```

```

        System.out.println(s + ", w2 = " + w2);

        ByteArrayOutputStream bout =
            new ByteArrayOutputStream();

        ObjectOutputStream out2 =
            new ObjectOutputStream(bout);

        out2.writeObject("Worm storage");
        out2.writeObject(w);

        out2.flush();

        ObjectInputStream in2 =
            new ObjectInputStream(
                new ByteArrayInputStream(
                    bout.toByteArray()));

        s = (String)in2.readObject();

        Worm w3 = (Worm)in2.readObject();

        System.out.println(s + ", w3 = " + w3);
    }
} ///:~

```

Pour rendre les choses intéressantes, les tableaux d'objets **Data** à l'intérieur de **Worm** sont initialisés avec des nombres aléatoires. (De cette manière vous ne pourrez suspecter le compilateur de conserver une quelconque sorte de méta-information.) Chaque segment de **Worm** est étiqueté avec un **char** qui est généré automatiquement dans un processus de génération récursive de la liste liée de **Worms**. Quand vous créez un **Worm**, vous indiquez au constructeur la longueur désirée. Pour créer la référence **next** il fait appel au constructeur de **Worms** avec une longueur de moins un, etc. La référence **next** finale est laissée comme **null**, indiquant la fin du **Worm**.

L'essentiel de tout cela est de rendre quelque chose raisonnablement complexe qui ne puisse pas facilement être sérialisé. L'action de sérialiser, cependant, est plutôt simple. Une fois que l'**ObjectOutputStream** est créé depuis un autre flux, **writeObject()** sérialise l'objet. Notez aussi l'appel de **writeObject()** pour un **String**. Vous pouvez aussi écrire tous les types de données primitives utilisant les même méthodes qu'**DataOutputStream** (ils partagent la même interface).

Il y a deux portions de code séparées qui ont une apparence similaire. La première écrit et lit et la seconde, pour varier, écrit et lit un **ByteArray**. Vous pouvez lire et écrire un objet en utilisant la sérialisation vers n'importe quel **DataInputStream** ou **DataOutputStream** incluant, comme vous le verrez dans le Chapitre 15, un réseau. La sortie d'une exécution donne :

```

Worm constructor: 6
Worm constructor: 5
Worm constructor: 4
Worm constructor: 3

```

```

Worm constructor: 2
Worm constructor: 1
w = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w2 = :a(262):b(100):c(396):d(480):e(316):f(398)
Worm storage, w3 = :a(262):b(100):c(396):d(480):e(316):f(398)

```

Vous pouvez voir que l'objet désérialisé contient vraiment tous les liens qui étaient dans l'objet original.

Notons qu'aucun constructeur, même pas le constructeur par défaut, n'est appelé dans le processus de désérialisation d'un objet **Serializable**. L'objet entier est restauré par récupération des données depuis l'**InputStream**.

La sérialisation objet est orientée-**byte**, et ainsi emploie les hiérarchies d'**InputStream** et d'**OutputStream**.

Trouver la classe

Vous devez vous demander ce qui est nécessaire pour qu'un objet soit récupéré depuis son état sérialisé. Par exemple, supposons que vous sérialisez un objet et que vous l'envoyez comme un fichier à travers un réseau vers une autre machine. Un programme sur l'autre machine pourra-t-il reconstruire l'objet en utilisant seulement le contenu du fichier ?

La meilleure manière de répondre à cette question est (comme d'habitude) en accomplissant une expérience. Le fichier suivant file dans le sous-répertoire pour ce chapitre :

```

//: c11:Alien.java
// Une classe sérializable.

import java.io.*;

public class Alien implements Serializable {
} ///:~

```

Le fichier qui crée et sérialise un objet **Alien** va dans le même répertoire :

```

//: c11:FreezeAlien.java
// Crée un fichier de sortie sérialisé.

import java.io.*;

public class FreezeAlien {
    // Lance les exeptions vers la console:

    public static void main(String[] args)

```

```

throws IOException {

    ObjectOutputStream out =

        new ObjectOutputStream(

            new FileOutputStream("X.file"));

    Alien zorcon = new Alien();

    out.writeObject(zorcon);

}

} ///:~

```

Plutôt que de saisir et de traiter les exeptions, ce programme prend une approche rapide et sale qui passe les exeptions en dehors de **main()**, ainsi elle seront reportés en ligne de commande.

Une fois que le programme est compilé et exécuté, copiez le **X.file** résultant dans un sous répertoire appelé **xfiles**, où va le code suivant :

```

///: c11:xfiles:ThawAlien.java

// Essaye de récupérer un fichier sérialisé sans

// la classe de l'objet qui est stocké dans ce fichier.

import java.io.*;

public class ThawAlien {

    public static void main(String[] args)

    throws IOException, ClassNotFoundException {

        ObjectInputStream in =

            new ObjectInputStream(

                new FileInputStream("X.file"));

        Object mystery = in.readObject();

        System.out.println(mystery.getClass());

    }

} ///:~

```

Ce programme ouvre le fichier et lit dans l'objet **mystery** avec succès. Pourtant, dès que vous essayez de trouver quelque chose à propos de l'objet – qui nécessite l'objet **Class** pour **Alien** – la Machine Virtuelle Java (JVM) ne peut pas trouver **Alien.class** (à moins qu'il arrive qu'il soit dans le Classpath, ce qui n'est pas le cas dans cet exemple). Vous obtiendrez un **ClassNotFoundException**. (Une fois encore, toute les preuves de l'existence de vie alien disparaît avant que la preuve de son existence soit vérifiée !)

Si vous espérez en faire plus après avoir récupéré un objet qui a été sérialisé, vous devrez vous assurer que la JVM puisse trouver les fichiers **.class** soit dans le chemin de class local ou quelque part sur l'Internet.

Contrôler la sérialisation

Comme vous pouvez le voir, le mécanisme de sérialisation par défaut est d'un usage trivial. Mais que faire si vous avez des besoins spéciaux ? Peut-être que vous avez des problèmes de sécurité spéciaux et que vous ne voulez pas sérialiser des parties de votre objet, ou peut-être que cela n'a pas de sens pour un sous-objet d'être sérialisé si cette partie doit être de nouveau créée quand l'objet est récupéré.

Vous pouvez contrôler le processus de sérialisation en implémentant l'interface **Externalizable** à la place de l'interface **Serializable**. L'interface **Externalizable** étend l'interface **Serializable** et ajoute deux méthodes, **writeExternal()** et **readExternal()**, qui sont automatiquement appelées pour votre objet pendant la sérialisation et la désérialisation afin que vous puissiez exécuter vos opérations spéciales.

L'exemple suivant montre des implémentations simple des méthodes de l'interface **Externalizable**. Notez que **Blip1** et **Blip2** sont presque identiques à l'exception d'une subtile différence (voyez si vous pouvez la découvrir en regardant le code) :

```
///  
// c11:Blips.java  
  
// Emploi simple d'Externalizable & un piège.  
  
import java.io.*;  
import java.util.*;  
  
class Blip1 implements Externalizable {  
    public Blip1() {  
        System.out.println("Blip1 Constructor");  
    }  
  
    public void writeExternal(ObjectOutput out)  
        throws IOException {  
        System.out.println("Blip1.writeExternal");  
    }  
  
    public void readExternal(ObjectInput in)  
        throws IOException, ClassNotFoundException {  
        System.out.println("Blip1.readExternal");  
    }  
}  
  
class Blip2 implements Externalizable {  
    Blip2() {  
        System.out.println("Blip2 Constructor");  
    }  
}
```

```
}

public void writeExternal(ObjectOutput out)
    throws IOException {
    System.out.println("Blip2.writeExternal");
}

public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException {
    System.out.println("Blip2.readExternal");
}
}

public class Blips {
    // Lance les exeptions vers la console :
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        System.out.println("Constructing objects:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        ObjectOutputStream o =
            new ObjectOutputStream(
                new FileOutputStream("Blips.out"));
        System.out.println("Saving objects:");
        o.writeObject(b1);
        o.writeObject(b2);
        o.close();
        //Maintenant faites les revenir :
        ObjectInputStream in =
            new ObjectInputStream(
                new FileInputStream("Blips.out"));
        System.out.println("Recovering b1:");
        b1 = (Blip1)in.readObject();
        // OOPS! Lance une exeption :
        //! System.out.println("Recovering b2:");
        //! b2 = (Blip2)in.readObject();
    }
}
```

```

    }
} ///:~

```

La sortie pour ce programme est :

```

Constructing objects:

Blip1 Constructor
Blip2 Constructor

Saving objects:

Blip1.writeExternal
Blip2.writeExternal

Recovering bl:

Blip1 Constructor
Blip1.readExternal

```

La raison pour laquelle l'objet **Blip2** n'est pas récupéré est que le fait d'essayer provoque une exception. Vous pouvez voir la différence entre **Blip1** et **Blip2** ? Le constructeur de **Blip1** est **public**, tandis que le constructeur de **Blip2** ne l'est pas, et cela lance une exception au recouvrement. Essayez de rendre le constructeur de **Blip2** **public** et retirez les commentaires **///** pour voir les résultats correct.

Quand **b1** est récupéré, le constructeur par défaut **Blip1** est appelé. Ceci est différent de récupérer un objet **Serializable**, dans lequel l'objet est construit entièrement de ses bits enregistrés, sans appel au constructeur. Avec un objet **Externalizable**, tous les comportements de construction par défaut se produisent (incluant les initialisations à ce point du champ de définition), et *alors* **readExternal()** est appelé. Vous devez prendre en compte ceci en particulier, le fait que toute la construction par défaut a toujours lieu pour produire le comportement correct dans vos objets **Externalizable**.

Voici un exemple qui montre ce que vous devez faire pour complètement stocker et retrouver un objet **Externalizable** :

```

///: c11:Blip3.java

// Reconstruction d'un objet externalizable.

import java.io.*;

import java.util.*;

class Blip3 implements Externalizable {

    int i;

    String s; // Aucune initialisation

    public Blip3() {

        System.out.println("Blip3 Constructor");
    }
}

```

```
    // s, i n'est pas initialisé
}

public Blip3(String x, int a) {
    System.out.println("Blip3(String x, int a)");

    s = x;
    i = a;

    // s & i initialisé seulement dans le non
    // constructeur par défaut.
}

public String toString() { return s + i; }
public void writeExternal(ObjectOutput out)
throws IOException {
    System.out.println("Blip3.writeExternal");
    // Vous devez faire ceci :
    out.writeObject(s);
    out.writeInt(i);
}

public void readExternal(ObjectInput in)
throws IOException, ClassNotFoundException {
    System.out.println("Blip3.readExternal");
    // Vous devez faire ceci :
    s = (String)in.readObject();
    i = in.readInt();
}

public static void main(String[] args)
throws IOException, ClassNotFoundException {
    System.out.println("Constructing objects:");
    Blip3 b3 = new Blip3("A String ", 47);
    System.out.println(b3);
    ObjectOutputStream o =
        new ObjectOutputStream(
            new FileOutputStream("Blip3.out"));
    System.out.println("Saving object:");
    o.writeObject(b3);
}
```

```

        o.close();

        // Maintenant faites le revenir :

        ObjectInputStream in =

            new ObjectInputStream(

                new FileInputStream("Blip3.out"));

        System.out.println("Recovering b3:");

        b3 = (Blip3)in.readObject();

        System.out.println(b3);

    }

} ///:~

```

Les champs **s** et **i** sont initialisés seulement dans le second constructeur, mais pas dans le constructeur par défaut. Ceci signifie que si vous n'initialisez pas **s** et **i** dans **readExternal()**, il sera alors **null** (parce que le stockage pour l'objet arrive nettoyé à zéro dans la première étape de la création de l'objet). Si vous enlevez les commentaires sur les deux lignes suivant les phrases « Vous devez faire ceci » et lancez le programme, vous verrez que lorsque l'objet est récupéré, **s** est **null** et **i** est zéro.

Si vous l'héritage se fait depuis un objet **Externalizable**, vous appellerez typiquement les versions classe-de-base de **writeExternal()** et **readExternal()** pour fournir un stockage et une récupération propre des composants de classe-de-base.

Ainsi pour faire fonctionner correctement les choses vous ne devrez pas seulement écrire les données importantes depuis l'objet pendant la méthode **writeExternal()** (il n'y a pas de comportement par défaut qui écrit n'importe quels objets pour un objet **Externalizable** object), mais vous devrez aussi récupérer ces données dans la méthode **readExternal()**. Ceci peut être un petit peu confus au premier abord parce que le constructeur par défaut du comportement pour un objet **Externalizable** peut le faire ressembler à une sorte de stockage et de récupération ayant lieu automatiquement. Ce n'est pas le cas.

Le mot-clé « transient »

Quand vous contrôlez la sérialisation, il peut y avoir un sous-objet précis pour qui vous ne voulez pas que le mécanisme de sérialisation java sauve et restaure automatiquement. C'est communément le cas si le sous-objet représente des informations sensibles que vous ne désirez pas sérialiser, comme un mot de passe. Même si cette information est **private** dans l'objet, une fois qu'elle est sérialisée il est possible pour quelqu'un d'y accéder en lisant un fichier ou en interceptant une transmission réseau.

Une manière de prévenir les parties sensibles de votre objet d'être sérialisé est d'implémenter votre classe comme **Externalizable**, comme montré précédemment. Ainsi rien n'est sérialisé automatiquement et vous pouvez sérialiser explicitement seulement les parties nécessaires dans **writeExternal()**.

Si vous travaillez avec un objet **Serializable**, néanmoins, toutes les sérialisation arrivent de façon automatique. Pour contrôler ceci, vous pouvez fermer la sérialisation sur une base de champ-par-champ en utilisant le mot **transient**, lequel dit « Ne t'embarrasse pas à sauver ou restaurer ceci – Je me charge de ça. »

Par exemple, considérons un objet **Login** qui conserve les informations à propos d'un login de session particulier. Supposez que, dès que vous vérifiez le login, vous désirez stocker les données, mais sans le mot de

passé. La manière la plus simple pour réaliser ceci est en d'implémentant **Serializable** et en marquant le champ **password** comme **transient**. Voici ce à quoi cela ressemble :

```
//: c11:Logon.java
// Explique le mot « transient. »

import java.io.*;
import java.util.*;

class Logon implements Serializable {
    private Date date = new Date();
    private String username;
    private transient String password;
    Logon(String name, String pwd) {
        username = name;
        password = pwd;
    }
    public String toString() {
        String pwd =
            (password == null) ? "(n/a)" : password;
        return "logon info: \n    " +
            "username: " + username +
            "\n    date: " + date +
            "\n    password: " + pwd;
    }
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        Logon a = new Logon("Hulk", "myLittlePony");
        System.out.println( "logon a = " + a);
        ObjectOutputStream o =
            new ObjectOutputStream(
                new FileOutputStream("Logon.out"));
        o.writeObject(a);
        o.close();
        // Délai :
        int seconds = 5;
    }
}
```

```

    long t = System.currentTimeMillis()
        + seconds * 1000;

    while(System.currentTimeMillis() < t)
        ;

    // Maintenant faites les revenir :

    ObjectInputStream in =
        new ObjectInputStream(
            new FileInputStream("Logon.out"));

    System.out.println(
        "Recovering object at " + new Date());

    a = (Logon)in.readObject();

    System.out.println( "logon a = " + a);
}
} ///:~

```

Vous pouvez voir que les champs **date** et **username** sont normaux (pas **transient**), et ils sont ainsi sérialisés automatiquement. Pourtant, **password** est **transient**, et donc n'est pas stocké sur le disque; aussi le mécanisme de sérialisation ne fait aucune tentative pour le récupérer. La sortie donne :

```

logon a = logon info:

    username: Hulk

    date: Sun Mar 23 18:25:53 PST 1997

    password: myLittlePony

Recovering object at Sun Mar 23 18:25:59 PST 1997

logon a = logon info:

    username: Hulk

    date: Sun Mar 23 18:25:53 PST 1997

    password: (n/a)

```

Lorsque l'objet est récupéré, le champ de **password** est **null**. Notez que **toString()** est obligé de contrôler la valeur **null** de **password** parcequ'il essaye d'assembler un objet String utilisant l'opérateur surchargé **+**, et que cet opérateur est confronté à une référence de type **null**, on aurait donc un **NullPointerException**. (Les nouvelles versions de Java contiendront peut être du code pour résoudre ce problème.)

Vous pouvez aussi voir que le champ **date** est stocké sur et récupéré depuis le disque et n'en génère pas une nouvelle.

Comme les objets **Externalizable** ne stockent pas tous leurs champs par défaut, le mot-clé **transient** est a

employer avec les objets **Serializable** seulement.

Une alternative à Externalizable

Si vous n'êtes pas enthousiasmé par l'implémentation de l'interface **Externalizable**, il y a une autre approche. Vous pouvez implémenter l'interface **Serializable** et *ajouter* (notez que je dit « ajouter » et non pas « imposer » ou « implémenter ») des méthodes appelées **writeObject()** et **readObject()** qui seront automatiquement appelées quand l'objet est sérialisé et désérialisé, respectivement. C'est à dire, si vous fournissez ces deux méthodes elles seront employées à la place de la sérialisation par défaut.

Ces méthodes devront avoir ces signatures exactes :

```
private void
    writeObject(ObjectOutputStream stream)
        throws IOException;

private void
    readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException
```

D'un point de vue conceptuel, les choses sont vraiment mystérieuses ici. Pour commencer, vous allez penser que parce que ces méthodes ne font pas partie des classes de base ou de l'interface **Serializable**, elles devraient être définies dans leur propre interface(s). Mais notez qu'elles sont définies comme **private**, ce qui veut dire qu'elles doivent être appelées seulement par les autres membres de cette classe. Cependant, vous ne faites pas réellement appel à elles depuis les autres membres de cette classe, mais à la place des méthodes **writeObject()** et **readObject()** des objets **ObjectOutputStream** et **ObjectInputStream** appellent vos objets **writeObject()** et les méthodes **readObject()**. (Remarquez mon énorme retenue de ne pas me lancer dans une longue diatribe au sujet de l'emploi des même noms de méthodes ici. En un mot : embrouillant.) Vous pourrez vous demander comment les objets **ObjectOutputStream** et **ObjectInputStream** ont accès aux méthodes **private** de votre class. Nous pouvons seulement supposer que cela fait partie de la magie de la sérialisation.

De toute façon, tout ce qui est défini dans une **interface** est automatiquement **public** donc **writeObject()** et **readObject()** doivent être **private**, à ce moment là ils feront partie d'une **interface**. Puisque vous devez suivre exactement les signatures, l'effet est le même que si vous implémentiez une **interface**.

Il apparaîtra que lorsque vous appelez **ObjectOutputStream.writeObject()**, l'objet **Serializable** que vous lui transmettez est interrogé (utilisant la réflexion, pas de doute) pour voir si il implémente son propre **writeObject()**. Si c'est le cas, le processus normale de sérialisation est omis et est le **writeObject()** appelé. Le même type de situation existe pour **readObject()**.

Il y a une autre entorse. À l'intérieur de votre **writeObject()**, vous pouvez choisir d'exécuter l'action **writeObject()** par défaut en appelant **defaultWriteObject()**. Également, dans **readObject()** vous pouvez appeler **defaultReadObject()**. Voici un exemple simple qui démontre comment vous pouvez contrôler le stockage et la récupération d'un objet **Serializable** :

```
//: c11:SerialCtl.java
```



```
// Contrôler la sérialisation en ajoutant vos propres
// méthodes writeObject() et readObject().

import java.io.*;

public class SerialCtl implements Serializable {

    String a;

    transient String b;

    public SerialCtl(String aa, String bb) {

        a = "Not Transient: " + aa;

        b = "Transient: " + bb;

    }

    public String toString() {

        return a + "\n" + b;

    }

    private void

        writeObject(ObjectOutputStream stream)

            throws IOException {

        stream.defaultWriteObject();

        stream.writeObject(b);

    }

    private void

        readObject(ObjectInputStream stream)

            throws IOException, ClassNotFoundException {

        stream.defaultReadObject();

        b = (String)stream.readObject();

    }

    public static void main(String[] args)

        throws IOException, ClassNotFoundException {

        SerialCtl sc =

            new SerialCtl("Test1", "Test2");

        System.out.println("Before:\n" + sc);

        ByteArrayOutputStream buf =

            new ByteArrayOutputStream();

        ObjectOutputStream o =
```

```

        new ObjectOutputStream(buf);

o.writeObject(sc);

// Maintenant faites les revenir :

ObjectInputStream in =

    new ObjectInputStream(

        new ByteArrayInputStream(

            buf.toByteArray()));

SerialCtl sc2 = (SerialCtl)in.readObject();

System.out.println("After:\n" + sc2);

    }

} ///:~

```

Dans cet exemple, un champ **String** est normal et le second est **transient**, pour prouver que le champ non-**transient** est sauvé par la méthode **defaultWriteObject()** et le que le champ **transient** est sauvé et récupéré explicitement. Les champs sont initialisés dans le constructeur plutôt qu'au point de définition pour prouver qu'ils n'ont pas été initialisés par un certain mécanisme automatique durant la sérialisation.

Si vous employez le mécanisme par défaut pour écrire les parties non-**transient** de votre objet, vous devrez appeler **defaultWriteObject()** comme la première action dans **writeObject()** et **defaultReadObject()** comme la première action dans **readObject()**. Ce sont d'étranges appels à des méthodes. Il apparaît, par exemple, que vous appelez **defaultWriteObject()** pour un **ObjectOutputStream** et ne lui passez aucun argument, mais il tourne d'une manière ou d'une autre autour et connaît la référence à votre objet et comment écrire toutes les parties non-**transient**. Étrange.

Le stockage et la récupération des objets **transient** utilisent un code plus familier. Et cependant, pensez à ce qu'il se passe ici. Dans **main()**, un objet **SerialCtl** est créé, puis est sérialisé en un **ObjectOutputStream**. (Notez dans ce cas qu'un tampon est utilisé à la place d'un fichier – c'est exactement pareil pour tout l'**ObjectOutputStream**.) La sérialisation survient à la ligne :

```
o.writeObject(sc);
```

La méthode **writeObject()** doit examiner **sc** pour voir si il possède sa propre méthode **writeObject()**. (Non pas en contrôlant l'interface – il n'y en a pas – ou le type de classe, mais en recherchant en fait la méthode en utilisant la réflexion.) Si c'est le cas, elle l'utilise. Une approche similaire garde true pour **readObject()**. Peut-être que c'est la seule réelle manière dont ils peuvent résoudre le problème, mais c'est assurément étrange.

Versioning

Il est possible que vous désiriez changer la version d'une classe sérialisable (les objets de la classe original peuvent être stockés dans un base de donnée, par exemple). Ceci est supporté mais vous devrez probablement le faire seulement dans les cas spéciaux, et cela requiert un profondeur supplémentaire de compréhension qui ne sera pas tenté d'atteindre ici. Les documents HTML du JDK téléchargeables depuis *java.sun.com* couvrent ce sujet de manière très approfondie.

Vous pourrez aussi noter dans la documentation HTML du JDK que de nombreux commentaires commencent par :

Attention : *Les objets sérialisés de cette classe ne seront pas compatibles avec les futures versions de Swing. Le support actuel de la sérialisation est approprié pour le stockage à court terme ou le RMI entre les applications.*
...

Ceci parce que le mécanisme de versionning est trop simple pour fonctionner de manière fiable dans toutes les situations, surtout avec les JavaBeans. Ils travaillent sur une correction de la conception, et c'est le propos de l'avertissement.

Utiliser la persistance

Il est plutôt attrayant de faire appel à la technologie de la sérialisation pour stocker certains états de votre programme afin que vous puissiez facilement récupérer le programme dans l'état actuel plus tard. Mais avant de pouvoir faire cela, il faut répondre à certaines questions. Qu'arrive-t-il si vous sérialisez deux objets qui ont tous les deux une référence à un troisième objet ? Quand vous récupérez ces deux objets depuis leur état sérialisé, aurez-vous une seule occurrence du troisième objet ? Que se passe-t-il si vous sérialisez vos deux objets pour séparer les fichiers et les désérialisez dans différentes parties de votre code ?

Voici un exemple qui montre le problème :

```
//: c11:MyWorld.java

import java.io.*;

import java.util.*;

class House implements Serializable {}

class Animal implements Serializable {

    String name;

    House preferredHouse;

    Animal(String nm, House h) {

        name = nm;

        preferredHouse = h;

    }

    public String toString() {

        return name + "[" + super.toString() +

            "], " + preferredHouse + "\n";

    }

}
```

```
public class MyWorld {

    public static void main(String[] args)
        throws IOException, ClassNotFoundException {

        House house = new House();

        ArrayList animals = new ArrayList();

        animals.add(
            new Animal("Bosco the dog", house));

        animals.add(
            new Animal("Ralph the hamster", house));

        animals.add(
            new Animal("Fronk the cat", house));

        System.out.println("animals: " + animals);

        ByteArrayOutputStream buf1 =
            new ByteArrayOutputStream();

        ObjectOutputStream o1 =
            new ObjectOutputStream(buf1);

        o1.writeObject(animals);
        o1.writeObject(animals); // Écrit un 2ème jeu
        // Écrit vers un flux différent :

        ByteArrayOutputStream buf2 =
            new ByteArrayOutputStream();

        ObjectOutputStream o2 =
            new ObjectOutputStream(buf2);

        o2.writeObject(animals);

        // Now get them back:

        ObjectInputStream in1 =
            new ObjectInputStream(
                new ByteArrayInputStream(
                    buf1.toByteArray()));

        ObjectInputStream in2 =
            new ObjectInputStream(
                new ByteArrayInputStream(
                    buf2.toByteArray()));
```

```

        ArrayList animals1 =
            (ArrayList)in1.readObject();
    ArrayList animals2 =
        (ArrayList)in1.readObject();
    ArrayList animals3 =
        (ArrayList)in2.readObject();

    System.out.println("animals1: " + animals1);
    System.out.println("animals2: " + animals2);
    System.out.println("animals3: " + animals3);
}
} ///:~

```

Une chose intéressante ici est qu'il est possible d'utiliser la sérialisation d'objet depuis et vers un tableau de bytes comme une manière de faire une « copie en profondeur » de n'importe quel objet qui est **Serializable**. (une copie en profondeur veut dire que l'on copie la structure complète des objets, plutôt que seulement l'objet de base et ses références.) La copie est abordée en profondeur dans l'Annexe A.

Les objets **Animal** contiennent des champs de type **House**. Dans **main()**, une **ArrayList** de ces **Animals** est créée et est sérialisée deux fois vers un flux et ensuite vers un flux distinct. Quand ceci est désérialisé et affiché, on obtient les résultats suivant pour une exécution (les objets seront dans des emplacements mémoire différents à chaque exécution) :

```

animals: [Bosco the dog[Animal@1cc76c], House@1cc769
, Ralph the hamster[Animal@1cc76d], House@1cc769
, Fronk the cat[Animal@1cc76e], House@1cc769
]
animals1: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals2: [Bosco the dog[Animal@1cca0c], House@1cca16
, Ralph the hamster[Animal@1cca17], House@1cca16
, Fronk the cat[Animal@1cca1b], House@1cca16
]
animals3: [Bosco the dog[Animal@1cca52], House@1cca5c
, Ralph the hamster[Animal@1cca5d], House@1cca5c
, Fronk the cat[Animal@1cca61], House@1cca5c
]

```

Bien sûr vous vous attendez à ce que les objets désérialisés aient des adresses différentes des originaux. Mais notez que dans **animals1** et **animals2** les mêmes adresses apparaissent, incluant les références à l'objet **House** que tous les deux partagent. D'un autre côté, quand **animals3** est récupéré le système n'a pas de moyen de savoir que les objets de l'autre flux sont des alias des objets du premier flux, donc il crée un réseau d'objets complètement différent.

Aussi longtemps que vos sérialisez tout dans un flux unique, vous pourrez récupérer le même réseau d'objets que vous avez écrits, sans aucune duplication accidentelle d'objets. Bien sûr, vous pouvez modifier l'état de vos objets entre la période d'écriture du premier et du dernier, mais c'est de votre responsabilité – les objets seront écrit dans l'état où ils sont quel qu'il soit (et avec les connections quelles qu'elles soient qu'ils ont avec les autres objets) au moment où vous les sérialiserez.

La chose la plus sûre à faire si vous désirez sauver l'état d'un système est de sérialiser comme une opération « atomique. » Si vous sérialisez quelque chose, faites une autre action, et en sérialisez une autre en plus, etc., alors vous ne stockerez pas le système sûrement. Au lieu de cela, mettez tous les objets qui comprennent l'état de votre système dans un simple conteneur et écrivez simplement ce conteneur à l'extérieur en une seule opération. Ensuite vous pourrez aussi bien le récupérer avec un simple appel à une méthode.

L'exemple suivant est un système imaginaire de conception assistée par ordinateur (CAD) qui démontre cette approche. En plus, il projette dans le sujet des champs **static** – si vous regardez la documentation vous verrez que **Class** est **Serializable**, donc il sera facile de stocker les champs **static** en sérialisant simplement l'objet **Class**. Cela semble comme une approche sensible, en tous cas.

```

//: c11:CADState.java

// Sauve et récupère l'état de la
// simulation d'un système de CAD.

import java.io.*;

import java.util.*;

abstract class Shape implements Serializable {

    public static final int

        RED = 1, BLUE = 2, GREEN = 3;

    private int xPos, yPos, dimension;

    private static Random r = new Random();

    private static int counter = 0;

    abstract public void setColor(int newColor);

    abstract public int getColor();

    public Shape(int xVal, int yVal, int dim) {

        xPos = xVal;

        yPos = yVal;

        dimension = dim;
    }

```

```
}

public String toString() {
    return getClass() +
        " color[" + getColor() +
        "]" xPos[" + xPos +
        "]" yPos[" + yPos +
        "]" dim[" + dimension + "]\n";
}

public static Shape randomFactory() {
    int xVal = r.nextInt() % 100;
    int yVal = r.nextInt() % 100;
    int dim = r.nextInt() % 100;
    switch(counter++ % 3) {
        default:
            case 0: return new Circle(xVal, yVal, dim);
            case 1: return new Square(xVal, yVal, dim);
            case 2: return new Line(xVal, yVal, dim);
    }
}
}
```

```
class Circle extends Shape {
    private static int color = RED;

    public Circle(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }

    public void setColor(int newColor) {
        color = newColor;
    }

    public int getColor() {
        return color;
    }
}
```

```
class Square extends Shape {  
    private static int color;  
    public Square(int xVal, int yVal, int dim) {  
        super(xVal, yVal, dim);  
        color = RED;  
    }  
    public void setColor(int newColor) {  
        color = newColor;  
    }  
    public int getColor() {  
        return color;  
    }  
}
```

```
class Line extends Shape {  
    private static int color = RED;  
    public static void  
    serializeStaticState(ObjectOutputStream os)  
        throws IOException {  
        os.writeInt(color);  
    }  
    public static void  
    deserializeStaticState(ObjectInputStream os)  
        throws IOException {  
        color = os.readInt();  
    }  
    public Line(int xVal, int yVal, int dim) {  
        super(xVal, yVal, dim);  
    }  
    public void setColor(int newColor) {  
        color = newColor;  
    }  
    public int getColor() {  
        return color;  
    }  
}
```



```
    }  
}  
  
public class CADState {  
    public static void main(String[] args)  
        throws Exception {  
        ArrayList shapeTypes, shapes;  
        if(args.length == 0) {  
            shapeTypes = new ArrayList();  
            shapes = new ArrayList();  
            // Ajoute des références aux objets de class :  
            shapeTypes.add(Circle.class);  
            shapeTypes.add(Square.class);  
            shapeTypes.add(Line.class);  
            // Fait quelques formes :  
            for(int i = 0; i < 10; i++)  
                shapes.add(Shape.randomFactory());  
            // Établit toutes les couleurs statiques en GREEN:  
            for(int i = 0; i < 10; i++)  
                ((Shape)shapes.get(i))  
                    .setColor(Shape.GREEN);  
            // Sauve le vecteur d'état :  
            ObjectOutputStream out =  
                new ObjectOutputStream(  
                    new FileOutputStream("CADState.out"));  
            out.writeObject(shapeTypes);  
            Line.serializeStaticState(out);  
            out.writeObject(shapes);  
        } else { // C'est un argument de ligne de commande  
            ObjectInputStream in =  
                new ObjectInputStream(  
                    new FileInputStream(args[0]));  
            // Read in the same order they were written:  
            shapeTypes = (ArrayList)in.readObject();  
        }  
    }  
}
```

```

        Line.deserializeStaticState(in);

        shapes = (ArrayList)in.readObject();
    }

    // Affiche les formes :

    System.out.println(shapes);

}

} ///:~

```

La classe **Shape** implémente **Serializable**, donc tout ce qui est hérité de **Shape** est aussi automatiquement **Serializable**. Chaque **Shape** contient des données, et chaque classe **Shape** dérivée contient un champ **static** qui détermine la couleur de tous ces types de **Shapes**. (Placer un champ **static** dans la classe de base ne donnera qu'un seul champ, puisque les champs **static** ne sont pas reproduits dans les classes dérivées.) Les méthodes dans les classes de base peuvent être surpassées [*overridden*] pour établir les couleurs des types variables (les méthodes **static** ne sont pas dynamiquement délimitées, donc ce sont des méthodes normales). La méthode **randomFactory()** crée un **Shape** différent chaque fois que vous y faites appel, utilisant des valeurs aléatoires pour les données du **Shape**.

Circle and **Square** sont des extensions directes de **Shape** ; la seule différence est que **Circle** initialise **color** au point de définition et que **Square** l'initialise dans le constructeur. Nous laisserons la discussion sur **Line** pour plus tard.

Dans **main()**, une **ArrayList** est employée pour tenir les objets de **Class** et les autres pour tenir les formes. Si vous ne fournissez pas un argument en ligne de commande la **shapeTypes ArrayList** est créé et les objets **Class** sont ajoutés, et ensuite l'**ArrayList** de **shapes** est créé et les objets **Shape** sont ajoutés. Puis, toutes les valeurs **static** de couleur (**color**) sont établies à **GREEN**, et tout est sérialisé vers le fichier **CADState.out**.

Si l'on fournit un argument en ligne de commande (vraisemblablement **CADState.out**), ce fichier est ouvert et utilisé pour restituer l'état du programme. Dans les deux situations, l'**ArrayList** de **Shapes** est affichée. Le résultat d'une exécution donne :

```

>java CADState

[class Circle color[3] xPos[-51] yPos[-99] dim[38]
, class Square color[3] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[3] xPos[-70] yPos[1] dim[16]
, class Square color[3] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[3] xPos[-75] yPos[-43] dim[22]
, class Square color[3] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[3] xPos[17] yPos[90] dim[-76]

```

```

]

>java CADState CADState.out

[class Circle color[1] xPos[-51] yPos[-99] dim[38]
, class Square color[0] xPos[2] yPos[61] dim[-46]
, class Line color[3] xPos[51] yPos[73] dim[64]
, class Circle color[1] xPos[-70] yPos[1] dim[16]
, class Square color[0] xPos[3] yPos[94] dim[-36]
, class Line color[3] xPos[-84] yPos[-21] dim[-35]
, class Circle color[1] xPos[-75] yPos[-43] dim[22]
, class Square color[0] xPos[81] yPos[30] dim[-45]
, class Line color[3] xPos[-29] yPos[92] dim[17]
, class Circle color[1] xPos[17] yPos[90] dim[-76]
]

```

Vous pouvez voir que les valeurs d'**xPos**, **yPos**, et **dim** sont toutes stockées et récupérées avec succès, mais qu'il y a quelque chose d'anormal dans la récupération d'informations **static**. Les « 3 » rentrent bien, mais ne sortent pas de la même manière. Les **Circles** ont une valeur de 1 (**RED**, ce qui est la définition), et les **Squares** ont une valeur de 0 (rappelez-vous, ils sont initialisés dans le constructeur). C'est comme si les **statics** n'étaient pas sérialisées du tout ! Ce qui est correct – malgré que la class **Class** est **Serializable**, cela n'agit pas comme on pouvait l'espérer. Donc si vous désirez sérialiser des **statics**, vous devrez le faire vous-même.

C'est à quoi les méthodes **static serializeStaticState()** et **deserializeStaticState()** dans **Line** servent. Vous pouvez voir qu'elles sont appelées explicitement comme une partie du processus de stockage et de récupération. (Notez que l'ordre d'écriture vers le fichier sérialisé et de sa relecture doit être conservé.) Ainsi pour que **CADState.java** s'exécute correctement vous devez :

1. Ajouter un **serializeStaticState()** et un **deserializeStaticState()** aux figures [shapes].
2. Enlever **ArrayList shapeTypes** et tout le code s'y rattachant.
3. Ajouter des appels aux nouvelles méthodes statiques de sérialisation et de désérialisation dans les figures [shapes].

Un autre sujet auquel vous devez penser est la sécurité, vu que la sérialisation sauve aussi les données **private**. Si vous vous avez orientation de sécurité, ces champs doivent être marqués comme **transient**. Mais alors vous devez concevoir une manière sûre pour stocker cette information de sorte que quand vous faites une restauration vous pouvez remettre à l'état initial [reset] ces variables **privates**.

Tokenizer l'entrée

Tokenizing est le processus de casser une séquence de caractères en un séquence de « tokens, » qui sont des morceaux de texte délimités par quoi que vous vous choisissiez. Par exemple, vos jetons [tokens] peuvent être des mots, et ensuite ils pourront être délimités par un blanc et de la ponctuation. Il y a deux classes fournies

dans la librairie standard de Java qui peuvent être employées pour la tokenisation : **StreamTokenizer** and **StringTokenizer**.

StreamTokenizer

Bien que **StreamTokenizer** ne soit pas dérivé d'**InputStream** ou **OutputStream**, il ne fonctionne qu'avec les objets **InputStreams**, donc il appartient à juste titre à la partie d'E/S de la librairie.

Considérons un programme qui compte les occurrences de mots dans un fichier texte :

```
///  
// c11:WordCount.java  
  
// Compte les mots dans un fichier, produit  
// les résultats dans un formulaire classé.  
  
import java.io.*;  
import java.util.*;  
  
class Counter {  
    private int i = 1;  
    int read() { return i; }  
    void increment() { i++; }  
}  
  
public class WordCount {  
    private FileReader file;  
    private StreamTokenizer st;  
  
    // Un TreeMap conserve les clés dans un ordre classé :  
    private TreeMap counts = new TreeMap();  
  
    WordCount(String filename)  
        throws FileNotFoundException {  
        try {  
            file = new FileReader(filename);  
            st = new StreamTokenizer(  
                new BufferedReader(file));  
            st.ordinaryChar('.');  
            st.ordinaryChar('-');  
        } catch (FileNotFoundException e) {  
            System.err.println(  

```

```
        "Could not open " + filename);
    throw e;
}
}

void cleanup() {
    try {
        file.close();
    } catch(IOException e) {
        System.err.println(
            "file.close() unsuccessful");
    }
}

void countWords() {
    try {
        while(st.nextToken() !=
            StreamTokenizer.TT_EOF) {
            String s;
            switch(st.ttype) {
                case StreamTokenizer.TT_EOL:
                    s = new String("EOL");
                    break;
                case StreamTokenizer.TT_NUMBER:
                    s = Double.toString(st.nval);
                    break;
                case StreamTokenizer.TT_WORD:
                    s = st.sval; // Déjà un String
                    break;
                default: // un seul caractère dans type
                    s = String.valueOf((char)st.ttype);
            }
            if(counts.containsKey(s))
                ((Counter)counts.get(s)).increment();
            else
                counts.put(s, new Counter());
        }
    }
}
```

```

    }

    } catch(IOException e) {

        System.err.println(

            "st.nextToken() unsuccessful");

    }

}

Collection values() {

    return counts.values();

}

Set keySet() { return counts.keySet(); }

Counter getCounter(String s) {

    return (Counter)counts.get(s);

}

public static void main(String[] args)

throws FileNotFoundException {

    WordCount wc =

        new WordCount(args[0]);

    wc.countWords();

    Iterator keys = wc.keySet().iterator();

    while(keys.hasNext()) {

        String key = (String)keys.next();

        System.out.println(key + ": "

            + wc.getCounter(key).read());

    }

    wc.cleanup();

}

} ///:~

```

Présenter les mots sous une forme classée est facile à faire en stockant les données dans un `TreeMap`, qui organise automatiquement ces clés dans un ordre classé (voir le Chapitre 9). Quand vous avez un jeu de clés en utilisant `keySet()`, elles seront automatiquement rangées dans l'ordre.

Pour ouvrir le fichier, un **FileReader** est utilisé, et pour changer le fichier en mots un **StreamTokenizer** est créé depuis le **FileReader** enveloppé dans un **BufferedReader**. Dans **StreamTokenizer**, il y a une liste par défaut de séparateurs, et vous pouvez en ajouter d'autres avec un jeu de méthodes. Ici, `ordinaryChar()` est utilisé pour dire « Ce caractère n'a aucune signification pour que je m'y intéresse, » donc l'analyseur ne les incluras pas comme des parties de tous les mots qu'il va créer. Par exemple, dire `st.ordinaryChar('.')` veut dire

que les points ne seront pas inclus comme des parties des mots qui seront analysés. Vous pourrez trouver plus d'information dans la documentation HTML du JDK à [<java.sun.com>](http://java.sun.com).

Dans **countWords()**, les tokens sont tirés un à un depuis le stream, et l'information **ttype** est utilisée pour déterminer ce qu'il faut faire avec chaque token, vu qu'un token peut être une fin de ligne, un nombre, une chaîne de caractère [string], ou un simple caractère.

Une fois qu'un token est trouvé, le **counts** de **TreeMap** est appelé pour voir si il contient déjà le token sous la forme d'une clé. Si c'est le cas, l'objet **Counter** correspondant est incrémenté pour indiquer qu'une autre instance de ce mots a été trouvée. Si ce n'est pas le cas, un nouveau **Counter** est créé – vu que le constructeur de **Counter** initialise sa valeur à un, ceci agit aussi pour compter le mot.

WordCount n'est pas un type de **TreeMap**, donc il n'est pas hérité. Il accomplit un type de fonctionnalité spécifique, ainsi bien que les méthodes **keys()** et **values()** doivent être re-exposées, cela ne veut pas forcément dire que l'héritage doit être utilisé vu qu'un bon nombre de méthodes de **TreeMap** sont ici inappropriés. En plus, les autres méthodes comme **getCounter()**, qui prend le **Counter** pour un **String** particulier, et **sortedKeys()**, qui produit un **Iterator**, terminant le changement dans la forme d'interface de **WordCount**..

Dans **main()** vous pouvez voir l'emploi d'un **WordCount** pour ouvrir et compter les mots dans un fichier – cela ne prend que deux lignes de code. Puis un **Iterator** est extrait vers une liste triée de clés (mots), et est employé pour retirer chaque clé et **Count** associés. L'appel à **cleanup()** est nécessaire pour s'assurer que le fichier est fermé.

StringTokenizer

Bien qu'il ne fasse pas partie de la bibliothèque d'E/S, le **StringTokenizer** possède des fonctions assez similaires a **StreamTokenizer** comme il sera décrit ici.

Le **StringTokenizer** renvoie les tokens dans une chaîne de caractère un par un. Ces tokens sont des caractères consécutifs délimités par des tabulations, des espaces, et des nouvelles lignes. Ainsi, les tokens de la chaîne de caractère « Où est mon chat ? » sont « Où, » « est, » « mon, » « chat, » « ?. » Comme le **StreamTokenizer**, vous pouvez appeler le **StringTokenizer** pour casser l'entrée de n'importe quelle manière, mais avec **StringTokenizer** vous effectuez cela en passant un second argument au constructeur, qui est un **String** des délimiteurs que vous utiliserez. En général, si vous désirez plus de sophistication, employez un **StreamTokenizer**.

Vous appelez un objet **StringTokenizer** pour le prochain token dans la chaîne de caractère en utilisant la méthode **nextToken()**, qui renvoie soit un token ou une chaîne de caractère vide pour indiquer qu'il ne reste pas de tokens.

Comme exemple, le programme suivant exécute une analyse limitée d'une phrase, cherchant des phrases clés pour indiquer si la joie ou la tristesse sont sous-entendues.

```
/// c11:AnalyzeSentence.java
// Recherche des séries particulières dans les phrases.
import java.util.*;

public class AnalyzeSentence {
```

```
public static void main(String[] args) {  
    analyze("I am happy about this");  
    analyze("I am not happy about this");  
    analyze("I am not! I am happy");  
    analyze("I am sad about this");  
    analyze("I am not sad about this");  
    analyze("I am not! I am sad");  
    analyze("Are you happy about this?");  
    analyze("Are you sad about this?");  
    analyze("It's you! I am happy");  
    analyze("It's you! I am sad");  
}  
  
static StringTokenizer st;  
  
static void analyze(String s) {  
    prt("\nnew sentence >> " + s);  
  
    boolean sad = false;  
  
    st = new StringTokenizer(s);  
    while (st.hasMoreTokens()) {  
        String token = next();  
  
        // Cherche jusqu'à ce l'on trouve un des  
        // deux tokens de départ :  
  
        if(!token.equals("I") &&  
            !token.equals("Are"))  
            continue; // Haut de la boucle while  
  
        if(token.equals("I")) {  
            String tk2 = next();  
  
            if(!tk2.equals("am")) // Doit être après I  
                break; // Sortie de la boucle while  
  
            else {  
                String tk3 = next();  
  
                if(tk3.equals("sad")) {  
                    sad = true;  
  
                    break; // Sortie de la boucle while  
                }  
            }  
        }  
    }  
}
```



```
        if (tk3.equals("not")) {
            String tk4 = next();
            if(tk4.equals("sad"))
                break; // Laisse sad faux
            if(tk4.equals("happy")) {
                sad = true;
                break;
            }
        }
    }
}

if(token.equals("Are")) {
    String tk2 = next();
    if(!tk2.equals("you"))
        break; // Doit être après Are
    String tk3 = next();
    if(tk3.equals("sad"))
        sad = true;
    break; // Sortie de la boucle while
}
}

if(sad) prt("Sad detected");
}

static String next() {
    if(st.hasMoreTokens()) {
        String s = st.nextToken();
        prt(s);
        return s;
    }
    else
        return "";
}

static void prt(String s) {
    System.out.println(s);
}
```

```

    }
} ///:~

```

Pour que chaque chaîne de caractère soit analysée, une boucle **while** est entrée et les tokens sont poussés hors de la chaîne de caractères. Notez la première déclaration de **if**, qui dit de **continuer** (retourne au début de la boucle et recommence encore) si le token est ni un « I » ou un « Are. » Ceci signifie qu'il attrapera les tokens que si un « I » ou un « Are » est trouvé. Vous pourriez penser utiliser le `==` à la place de la méthode `equals()`, mais cela ne fonctionne pas correctement, comme `==` compare les références de valeur tandis que `equals()` compare les contenus.

La logique du reste de la méthode `analyze()` est que le pattern (motif) dont on recherche la présence est « I am sad, » « I am not happy, » or « Are you sad? » Sans la déclaration **break**, le code réalisant ça serait bien plus confus qu'il ne l'est. Vous devez être conscient qu'un parseur typique (ceci en est un exemple primitif) possède normalement une table de ces tokens et un morceau de code qui bouge de l'un à l'autre des statuts dans la table quand les nouveaux tokens sont lus.

Vous pourrez voir le **StringTokenizer** seulement comme un raccourci pour un type de **StreamTokenizer** simple et spécifique. Cependant, si vous avez un **String** que vous désirez tokenizer et que **StringTokenizer** est trop limité, tout ce que vous avez à faire est de le retourner dans un flux avec **StringBufferInputStream** puis de l'utiliser pour créer un **StreamTokenizer** beaucoup plus puissant.

Vérifier le style de capitalization

Dans cette partie on observera un exemple un peut plus complet d'utilisation du Java E/S. Ce projet est directement utile puisqu'il accomplit une vérification de style pour être sur que la capitalisation se conforme au style Java comme décrit sur java.sun.com/docs/codeconv/index.html. Il ouvre chaque fichier **.java** dans le répertoire courant et extrait tous les noms de classe et d'identifiants, puis nous indique si affiche l'un d'entre eux ne correspond pas au style Java.

Afin que le programme s'exécute correctement, on devra d'abord construire un dépôt de noms de classes pour conserver tous les noms de classes dans la bibliothèque standard de Java. On réalise ceci en se déplaçant dans tous les sous-répertoires du code source de la bibliothèque standard Java et en exécutant **ClassScanner** dans chaque sous répertoire. Donnant comme arguments le nom du fichier dépositaire (utilisant le même chemin et nom chaque fois) et l'option -lign de commande **-a** pour indiquer que les noms de classes devront être ajoutés au dépôt.

Afin d'utiliser le programme pour vérifier votre code, indiquez lui le chemin et le nom du dépôt à employer. Il vérifiera toutes les classes et les identifiants du répertoire courant et vous vous dira lesquels ne suivent pas le style de capitalisation Java.

Vous devrez être conscient que ce programme n'est pas parfait ; il y a quelques fois où il signalera ce qui lui semble être un problème mais en regardant le code vous verrez qu'il n'y a rien à changer. C'est quelque peut agaçant, mais c'est tellement plus simple que d'essayer de trouver tous ces cas en vérifiant de vos yeux votre code.

```

///: c11:ClassScanner.java

// Scanne tous les fichiers dans le répertoire pour les classes

// et les identifiants, pour vérifier la capitalisation.

// Présume des listes de code correctement compilées.

```

```
// Ne fait pas tout bien, mais c'est
// une aide utile.

import java.io.*;
import java.util.*;

class MultiStringMap extends HashMap {
    public void add(String key, String value) {
        if(!containsKey(key))
            put(key, new ArrayList());

        ((ArrayList)get(key)).add(value);
    }
    public ArrayList getArrayList(String key) {
        if(!containsKey(key)) {
            System.err.println(
                "ERROR: can't find key: " + key);
            System.exit(1);
        }
        return (ArrayList)get(key);
    }
    public void printValues(PrintStream p) {
        Iterator k = keySet().iterator();
        while(k.hasNext()) {
            String oneKey = (String)k.next();
            ArrayList val = getArrayList(oneKey);
            for(int i = 0; i < val.size(); i++)
                p.println((String)val.get(i));
        }
    }
}

public class ClassScanner {
    private File path;
    private String[] fileList;
```

```
private Properties classes = new Properties();

private MultiStringMap
    classMap = new MultiStringMap(),
    identMap = new MultiStringMap();

private StreamTokenizer in;

public ClassScanner() throws IOException {
    path = new File(".");
    fileList = path.list(new JavaFilter());
    for(int i = 0; i < fileList.length; i++) {
        System.out.println(fileList[i]);
        try {
            scanListing(fileList[i]);
        } catch(FileNotFoundException e) {
            System.err.println("Could not open " +
                fileList[i]);
        }
    }
}

void scanListing(String fname)
throws IOException {
    in = new StreamTokenizer(
        new BufferedReader(
            new FileReader(fname)));
    // Ne semble pas fonctionner :
    // in.slashStarComments(true);
    // in.slashSlashComments(true);
    in.ordinaryChar('/');
    in.ordinaryChar('.');
    in.wordChars('_', '_');
    in.eolIsSignificant(true);
    while(in.nextToken() !=
        StreamTokenizer.TT_EOF) {
        if(in.ttype == '/')
            eatComments();
    }
}
```

```

        else if(in.ttype ==
                StreamTokenizer.TT_WORD) {
            if(in.sval.equals("class") ||
                in.sval.equals("interface")) {
                // Prend le nom de la classe :
                while(in.nextToken() !=
                    StreamTokenizer.TT_EOF
                    && in.ttype !=
                    StreamTokenizer.TT_WORD)
                    ;

                classes.put(in.sval, in.sval);
                classMap.add(fname, in.sval);
            }
            if(in.sval.equals("import") ||
                in.sval.equals("package"))
                discardLine();
            else // C'est un identifiant ou un mot-clé
                identMap.add(fname, in.sval);
        }
    }
}

void discardLine() throws IOException {
    while(in.nextToken() !=
        StreamTokenizer.TT_EOF
        && in.ttype !=
        StreamTokenizer.TT_EOL)
        ; // Renvoie les tokens en fin de ligne
}

// Le déplacement des commentaire de StreamTokenizer semble
// être cassé. Ceci les extrait :

void eatComments() throws IOException {
    if(in.nextToken() !=
        StreamTokenizer.TT_EOF) {
        if(in.ttype == '/')

```

```
        discardLine();
    else if(in.ttype != '*')
        in.pushBack();
    else
        while(true) {
            if(in.nextToken() ==
                StreamTokenizer.TT_EOF)
                break;
            if(in.ttype == '*')
                if(in.nextToken() !=
                    StreamTokenizer.TT_EOF
                    && in.ttype == '/')
                    break;
        }
    }
}

public String[] classNames() {
    String[] result = new String[classes.size()];
    Iterator e = classes.keySet().iterator();
    int i = 0;
    while(e.hasNext())
        result[i++] = (String)e.next();
    return result;
}

public void checkClassNames() {
    Iterator files = classMap.keySet().iterator();
    while(files.hasNext()) {
        String file = (String)files.next();
        ArrayList cls = classMap.getArrayList(file);
        for(int i = 0; i < cls.size(); i++) {
            String className = (String)cls.get(i);
            if(Character.isLowerCase(
                className.charAt(0)))
                System.out.println(
```

```
        "class capitalization error, file: "
        + file + ", class: "
        + className);
    }
}

public void checkIdentNames() {
    Iterator files = identMap.keySet().iterator();
    ArrayList reportSet = new ArrayList();
    while(files.hasNext()) {
        String file = (String)files.next();
        ArrayList ids = identMap.getArrayList(file);
        for(int i = 0; i < ids.size(); i++) {
            String id = (String)ids.get(i);
            if(!classes.contains(id)) {
                // Ignore le identifiants de longueur 3
                // qui sont majuscule sur tout leur longueur
                // (probablement des valeurs finales statiques) :
                if(id.length() >= 3 &&
                    id.equals(
                        id.toUpperCase()))
                    continue;

                // Vérifie pour voir si le premier caractère est majuscule :
                if(Character.isUpperCase(id.charAt(0))){
                    if(reportSet.indexOf(file + id)
                        == -1){ // Ne rend pas encore compte
                        reportSet.add(file + id);
                        System.out.println(
                            "Ident capitalization error in:"
                            + file + ", ident: " + id);
                    }
                }
            }
        }
    }
}
```

```
    }
}

static final String usage =
    "Usage: \n" +
    "ClassScanner classnames -a\n" +
    "\tAdds all the class names in this \n" +
    "\tdirectory to the repository file \n" +
    "\tcalled 'classnames'\n" +
    "ClassScanner classnames\n" +
    "\tChecks all the java files in this \n" +
    "\tdirectory for capitalization errors, \n" +
    "\tusing the repository file 'classnames'";

private static void usage() {
    System.err.println(usage);
    System.exit(1);
}

public static void main(String[] args)
throws IOException {
    if(args.length < 1 || args.length > 2)
        usage();

    ClassScanner c = new ClassScanner();
    File old = new File(args[0]);
    if(old.exists()) {
        try {
            // Essaye d'ouvrir un fichier de
            // propriété existant :

            InputStream oldlist =
                new BufferedInputStream(
                    new FileInputStream(old));

            c.classes.load(oldlist);
            oldlist.close();
        } catch(IOException e) {
            System.err.println("Could not open "
                + old + " for reading");
        }
    }
}
```



```

        System.exit(1);
    }
}

if(args.length == 1) {
    c.checkClassNames();
    c.checkIdentNames();
}

// Écrit les noms de classe dans un dépôt :
if(args.length == 2) {
    if(!args[1].equals("-a"))
        usage();

    try {
        BufferedOutputStream out =
            new BufferedOutputStream(
                new FileOutputStream(args[0]));

        c.classes.store(out,
            "Classes found by ClassScanner.java");

        out.close();
    } catch(IOException e) {
        System.err.println(
            "Could not write " + args[0]);

        System.exit(1);
    }
}
}

class JavaFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        // Retire les information de chemin :
        String f = new File(name).getName();

        return f.trim().endsWith(".java");
    }
}
//::~~

```

La classe **MultiStringMap** est un outil qui nous permet d'organiser un groupe de chaînes de caractères sur chaque entrée de clé. Il utilise un **HashMap** (cette fois avec héritage) avec la clé comme simple chaîne de caractère qui est organisée sur la valeur de l'**ArrayList**. La méthode **add()** vérifie simplement si il y a déjà une clé dans le **HashMap**, si ce n'est pas le cas elle en ajoute une à cet endroit. La méthode **getArrayList()** produit une **ArrayList** pour une clé particulière, et **printValues()**, qui est essentiellement utile pour le débogage, affiche toutes les valeurs **ArrayList** par **ArrayList**.

Pour avoir rester simple, les noms de classe des bibliothèques standard de Java sont toutes placées dans un objet **Properties** (de la bibliothèque standard de Java). Rappelons qu'un objet **Properties** est un **HashMap** qui ne garde que des objets **String** pour à la fois la clé et les valeurs d'entrée. Cependant, il peut être sauvé vers le disque et restauré depuis le disque par un appel de méthode, donc c'est l'idéal pour le dépôt des noms. Actuellement, on ne désire qu'une liste de noms, et un **HashMap** ne peut pas accepter une **null** valeur à la fois pour ces clés ou ces valeurs d'entrée.

Pour les classes et identifiants qui sont découverts pour les fichiers d'un certain répertoire, deux **MultiStringMaps** sont utilisés : **classMap** et **identMap**. Aussi, quand le programme démarre il charge le dépôt de nom de classe standard dans l'objet **Properties** appelé **classes**, et quand un nouveau nom de classe est découvert dans le répertoire local il est aussi ajouté à **classes** en plus de **classMap**. De cette manière, **classMap** peut être employé pour se déplacer à travers toutes les classes du répertoire local, et **classes** peut être utilisé pour voir si le token courant est un nom de classe (qui indique la définition d'un objet ou le début d'une méthode, donc saisit le token suivant – jusqu'au point virgule – et le place dans **identMap**).

Le constructeur par défaut pour **ClassScanner** crée une liste de noms de fichier, en utilisant **JavaFilter** implémentation du **FilenameFilter**, montré à la fin du fichier. Ensuite il appelle **scanListing()** pour chaque nom de fichier.

Dans **scanListing()** le fichier de code source est ouvert et transformé en un **StreamTokenizer**. Dans la documentation, passer **true** de **slashStarComments()** à **slashSlashComments()** est supposé enlever ces commentaires, mais cela semble être quelque peu défectueux, car cela semble ne pas fonctionner. Au lieu de cela, ces lignes sont marquées en tant que commentaires et les commentaires sont extraits par une autre méthode. Pour cela, le « / » doit être capturé comme un caractère ordinaire plutôt que de laisser le **StreamTokenizer** l'absorber comme une partie de commentaire, et la méthode **ordinaryChar()** dit au **StreamTokenizer** de faire cela. C'est aussi vrai pour les points (« . »), vu que nous voulons que l'appel de la méthode soit séparé en deux en identifiants individuels. Pourtant, l'underscore (soulignement), qui est ordinairement traité par **StreamTokenizer** comme un caractère individuel, doit être laissé comme une partie des identifiants puisqu'il apparaît dans des valeurs **static final** comme **TT_EOF**, etc., employé dans tant de nombreux programmes. La méthode **wordChars()** prend une rangée de caractères que l'on désire ajouter à ceux qui sont laissés dans un token qui a été analysé comme un mot. Finalement, lorsqu'on fait l'analyse pour une ligne de commentaire ou que l'on met de côté une ligne on veut savoir quand arrive la fin de ligne, c'est pourquoi en appelant **eolIsSignificant(true)** l'EOL (*End Of Line* : fin de ligne) apparaîtra plutôt que sera absorbé par le **StreamTokenizer**.

Le reste du **scanListing()** lit et réagit aux tokens jusqu'à la fin du fichier, annonçant quand **nextToken()** renvoie la valeur **final static StreamTokenizer.TT_EOF**.

Si le token est un « / » il est potentiellement un commentaire, donc **eatComments()** est appelé pour voir avec lui. La seule autre situation qui nous intéresse est quand il s'agit d'un mot, sur lequel il y a des cas spéciaux.

Si le mot est **class** ou **interface** alors le prochain token représente une classe ou une interface, et il est placé dans **classes** et **classMap**. Si le mot est **import** ou **package**, alors on n'a plus besoin du reste de la ligne. Tout le reste doit être un identifiant (auquel nous sommes intéressés) ou un mot-clé (qui ne nous intéresse pas, mais ils

sont tous en minuscule de toutes manières donc cela n'abîmera rien de les placer dedans). Ceux-ci sont ajoutés à **identMap**.

La méthode **discardLine()** est un outil simple qui cherche la fin de ligne. Notons que chaque fois que l'on trouve un nouveau token, l'on doit effectuer le contrôle de la fin de ligne.

La méthode **eatComments()** est appelée toutes les fois qu'un slash avant est rencontré dans la boucle d'analyse principale. Cependant, cela ne veut pas forcément dire qu'un commentaire ai été trouvé, donc le prochain token doit être extrait pour voir si c'est un autre slash avant (dans ce cas la ligne est rejetée) ou un astérisque. Mais si c'est n'est pas l'un d'entre eux, cela signifie que le token qui vient d'être sorti est nécessaire dans la boucle d'analyse principale ! Heureusement, la **pushBack()** méthode nous permet de « pousser en arrière » le token courant dans le flux d'entrée pour que quand la boucle d'analyse principale appelle **nextToken()** elle prendra celui que l'on viens tout juste de pousser en arrière.

Par commodité, la méthode produit un tableau de tous les noms dans le récipient **classes**. Cette méthode n'est pas utilisée dans le programme mais est utiles pour le débogage.

Les deux prochaines méthodes sont celles dans lesquelles prend place la réelle vérification. Dans **checkClassNames()**, les noms des classe sont extraits depuis le **classMap** (qui, rappelons le, contient seulement les noms dans ce répertoire, organisés par nom de fichier de manière à ce que le nom de fichier puisse être imprimé avec le nom de classe errant). Ceci est réalisé en poussant chaque **ArrayList** associé et en allant plus loin que ça, en regardant pour voir si le premier caractère est en minuscule. Si c'est le cas, le message approprié est imprimé.

Dans **checkIdentNames()**, une approche similaire est prise : chaque nom d'identifiant est extrait depuis **identMap**. Si le nom n'est pas dans la liste **classes**, il est supposé être un identifiant ou un mot-clé. Un cas spécial est vérifié : si la longueur de l'identifiant est trois et tout les caractères sont en majuscule, cette identifiant est ignoré puisqu'il est probablement une valeur **static final** comme **TT_EOF**. Bien sur, ce n'est pas un algorithme parfait, mais il assume que l'on avertira par la suite de tous les identifiants complètement-majuscules qui sont hors sujet.

Au lieu de reporter tout les identifiants qui commencent par un caractère majuscule, cette méthode garde trace de celles qui ont déjà été rapportées dans **ArrayList** appelées **reportSet()**. Ceci traite l'**ArrayList** comme un « jeu » qui vous signale lorsqu'un élément se trouve déjà dans le jeu. L'élément est produit en ajoutant au nom de fichier l'identifiant. Si l'élément n'est pas dans le jeu, il est ajouté puis un le rapport est effectué.

Le reste du listing est composé de **main()**, qui s'en occupe lui même en manipulant les arguments de ligne de commande et prenant en compte de l'endroit où l'on a construit le dépôt de noms de classe de la bibliothèque standard Java ou en vérifiant la validité du code que l'on a écrit. Dans les deux cas il fait un objet **ClassScanner**.

Si l'on construit ou utilisons un dépôt, on doit essayer d'ouvrir les dépôts existants. En créant un objet **File** et en testant son existence, on peut décider que l'on ouvre le fichier et **load()** la liste de **classes Properties** dans **ClassScanner**. (Les classes du dépôt s'ajoutent, ou plutôt recouvrent, les classes trouvées par le constructeur **ClassScanner**.) Si l'on fournit seulement un seul argument en ligne de commande cela signifie que l'on désire accomplir une vérification des noms de classes et d'identifiants, mais si l'on fournit deux arguments (le second étant un « -a ») on construit un dépôt de noms de classes. Dans ce cas, un fichier de sortie est ouvert et la méthode **Properties.save()** est utilisée pour écrire la liste dans un fichier, avec une chaîne de caractère fournissant l'information d'en tête du fichier.

Résumé

La bibliothèque Java de flux d'E/S satisfait les exigences de base : on peut faire la lecture et l'écriture avec la console, un fichier, un bloc de mémoire, ou même à travers l'Internet (comme on pourra le voir au Chapitre 15). Avec l'héritage, on peut créer de nouveaux types d'objets d'entrée et de sortie. Et l'on peut même ajouter une simple extension vers les types d'objets qu'un flux pourrait accepter en redéfinissant la méthode **toString()** qui est automatiquement appelée lorsque l'on passe un objet à une méthode qui attendait un **String** (« conversion automatique de type » limitée à Java).

Il y a des questions laissées sans réponses par la documentation et la conception de la bibliothèque de flux. Par exemple, il aurait été agréable de pouvoir dire que l'on désire lancer une exception si l'on essaye d'écrire par dessus un fichier en l'ouvrant pour la sortie – certains systèmes de programmation permettant de spécifier que l'on désire ouvrir un fichier de sortie, mais seulement si il n'existe pas déjà. En Java, il apparaît que l'on est supposé utiliser un objet **File** pour déterminer qu'un fichier existe, puisque si on l'ouvre comme un **FileOutputStream** ou un **FileWriter** il sera toujours écrasé.

La bibliothèque de flux d'E/S amène des sentiments mélangés ; elle fait plus de travail et est portable. Mais si vous ne comprenez pas encore le decorator pattern, la conception n'est pas intuitive, il y a donc des frais supplémentaires pour l'apprendre et l'enseigner. Elle est aussi incomplète : il n'y a pas de support pour le genre de format de sortie que presque chaque paquetage d'E/S d'autres langages supportent.

Cependant, dès que vous *pourrez* comprendre le decorator pattern et commencerez à utiliser la bibliothèque pour des situations demandant sa flexibilité, vous pourrez commencer à bénéficier de cette conception, à tel point que ce que ça vous coûtera en lignes de codes supplémentaires ne vous ennuiera pas beaucoup.

Si vous n'avez pas trouvé ce que vous cherchiez dans ce chapitre (qui a seulement été une introduction, et n'est pas censée être complète), vous pourrez trouver une couverture détaillée dans *Java E/S*, de Elliotte Rusty Harold (O'Reilly, 1999).

Exercices

Les solutions aux exercices choisis peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour un faible coût sur www.BruceEckel.com.

1. Ouvrez un fichier de manière à pouvoir lire le fichier une ligne à la fois. Lisez chaque ligne comme un **String** et placez cet objet **String** dans un **LinkedList**. Affichez toutes les lignes dans le **LinkedList** en ordre inverse.
2. Modifiez l'exercice 1 pour que le nom du fichier à lire soit entré comme argument de ligne de commande.
3. Modifiez l'exercice 2 pour ouvrir aussi un fichier texte dans lequel vous pourrez écrire. Écrivez les lignes dans l'**ArrayList**, avec les numéros de ligne (ne pas essayer d'utiliser la classe « *LineNumber* »), vers le fichier.
4. Modifiez l'exercice 2 pour imposer les majuscules à toutes les lignes de l'**ArrayList** et envoyer les résultats à **System.out**.
5. Modifiez l'exercice 2 pour qu'il prenne un argument de ligne de commande supplémentaire des mots à trouver dans le fichier. Affichez toutes les lignes dans lesquelles on trouve le mot.
6. Modifier **DirList.java** pour que le **FilenameFilter** ouvre en fait chaque fichier et reçoive le fichier sur la base de n'importe quel argument tiré de la ligne de commande existant dans ce fichier.
7. Créez une classe nommée **SortedDirList** avec un constructeur qui prend l'information de chemin de

fichier et construit une liste triée du répertoire à partir des fichiers du répertoire. Créez deux méthodes **list()** surchargées avec un constructeur qui présenterons l'une ou l'autre la liste complète ou un sous-ensemble de la liste basée sur un argument. Ajoutez une méthode **size()** qui prendra un nom de fichier et présentera la taille de ce fichier.

8. Modifiez **WordCount.java** pour qu'il produise à la place un classement alphabétique, utilisant l'outil du Chapitre 9.
9. Modifiez **WordCount.java** pour qu'il utilise une classe contenant un **String** et une valeur de numérotation pour stocker chaque différent mot, et un **Set** de ces objets pour maintenir la liste de ces mots.
10. Modifiez **IOStreamDemo.java** afin qu'il utilise **LineNumberInputStream** pour garder trace du compte de ligne. Notez qu'il est plus facile de garder seulement les traces de manière programmatique.
11. En commençant avec la partie 4 de **IOStreamDemo.java**, écrivez un programme qui compare les performances à l'écriture d'un fichier en utilisant une E/S mise en mémoire tampon et l'autre non.
12. Modifiez la partie 5 d'**IOStreamDemo.java** pour éliminer les espaces dans la ligne produite par le premier appel à **in5br.readLine()**. Faites cela en employant une boucle **while** et **readChar()**.
13. Réparez le programme **CADState.java** comme décrit dans le texte.
14. Dans **Blips.java**, copiez le fichier et renommez le en **BlipCheck.java** et renommez la classe **Blip2** en **BlipCheck** (rendez la **public** et enlevez la portée publique de la classe **Blips** dans le processus). Enlevez les marques **//!** dans le fichier et lancez le programme incluant les mauvaises lignes. Ensuite, enlevez les commentaires du constructeur par défaut de **BlipCheck**. Lancez le et expliquez pourquoi il fonctionne. Notez qu'après la compilation, vous devrez exécuter le programme avec « **java Blips** » parce que la méthode **main()** est encore dans **Blips**.
15. Dans **Blip3.java**, enlevez les commentaires des deux lignes après les phrases « Vous devez faire ceci : » et lancez le programme. Expliquez le résultat et pourquoi il diffère de quand les deux lignes sont dans le programme.
16. (Intermédiaire) Dans le chapitre 8, repérez l'exemple **GreenhouseControls.java**, qui se compose de trois fichiers. Dans **GreenhouseControls.java**, la classe interne **Restart()** contient un jeu d'événements codé durement. Changez le programme pour qu'il lise les événements et leurs heures relatives depuis un fichier texte. (Défi : Utilisez une *méthode de fabrique* d'un design pattern pour construire les événements – voir *Thinking in Patterns with Java*, téléchargeable à www.BruceEckel.com.)

[57] *Design Patterns*, Erich Gamma *et al.*, Addison-Wesley 1995.

[58] XML est un autre moyen de résoudre le problème de déplacer les données entre différentes plates-formes informatiques, et ne dépend pas du fait d'avoir Java sur toutes les plates-formes. Cependant, des outils Java existent qui supportent XML.

[59] Le chapitre 13 montre une solution même plus commode pour cela : un programme GUI avec une zone de texte avec ascenseur.