

25.04.2001 - version 1.1 [Armel]

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

08.06.2000 - version 1.0 [Philippe BOITE]

- Dernière mise à jour de la version française

## 5 : Cacher l'Implémentation

Une règle principale en conception orientée objet est de "séparer les choses qui changent des choses qui ne changent pas".

Ceci est particulièrement important pour les bibliothèques [*libraries*]. Les utilisateurs (*programmeurs clients*) de cette bibliothèque doivent pouvoir s'appuyer sur la partie qu'ils utilisent, et savoir qu'ils n'auront pas à réécrire du code si une nouvelle version de la bibliothèque sort. Inversement, le créateur de la bibliothèque doit avoir la liberté de faire des modifications et améliorations avec la certitude que le code du programmeur client ne sera pas affecté par ces modifications.

On peut y parvenir à l'aide de conventions. Par exemple le programmeur de bibliothèque doit admettre de ne pas enlever des méthodes existantes quand il modifie une classe dans la bibliothèque, puisque cela casserait le code du programmeur. La situation inverse est plus délicate. Dans le cas d'un membre de données, comment le créateur de bibliothèque peut-il savoir quels membres de données ont été accédés par les programmeurs clients ? C'est également vrai avec les méthodes qui ne sont qu'une partie de l'implémentation d'une classe et qui ne sont pas destinées à être utilisées directement par le programmeur client. Mais que se passe-t-il si le créateur de bibliothèque veut supprimer une ancienne implémentation et en mettre une nouvelle ? Changer un de ces membres pourrait casser le code d'un programmeur client. De ce fait la marge de manoeuvre du créateur de bibliothèque est plutôt étroite et il ne peut plus rien changer.

Pour corriger ce problème, Java fournit des *spécificateurs d'accès* pour permettre au créateur de bibliothèque de dire au programmeur client ce qui est disponible et ce qui ne l'est pas. Les niveaux de contrôles d'accès, depuis le "plus accessible" jusqu'au "moins accessible" sont **public**, **protected** (protégé), "friendly" (amical, qui n'a pas de mot-clé), et **private** (privé). Le paragraphe précédent semble montrer que, en tant que concepteur de bibliothèque, on devrait tout garder aussi "privé" [*private*] que possible, et n'exposer que les méthodes qu'on veut que le programmeur client utilise. C'est bien ce qu'il faut faire, même si c'est souvent non intuitif pour des gens qui programment dans d'autres langages (particulièrement en C) et qui ont l'habitude d'accéder à tout sans restrictions. Avant la fin de ce chapitre vous devriez être convaincus de l'importance du contrôle d'accès en Java.

Le concept d'une bibliothèque de composants et le contrôle de qui peut accéder aux composants de cette bibliothèque n'est cependant pas complet. Reste la question de savoir comment les composants sont liés entre eux dans une unité de bibliothèque cohérente. Ceci est contrôlé par le mot-clé **package** en Java, et les spécificateurs d'accès varient selon que la classe est dans le même package ou dans un autre package. Donc, pour commencer ce chapitre, nous allons apprendre comment les composants de bibliothèques sont placés dans des packages. Nous serons alors capables de comprendre la signification complète des spécificateurs d'accès.

### package : l'unité de bibliothèque

Un package est ce qu'on obtient lorsqu'on utilise le mot-clé **import** pour apporter une bibliothèque complète, tel que

```
import java.util.*;
```

Cette instruction apporte la bibliothèque complète d'utilitaires qui font partie de la distribution Java standard. Comme par exemple la classe **ArrayList** est dans **java.util**, on peut maintenant soit spécifier le nom complet **java.util.ArrayList** (ce qui peut se faire sans l'instruction **import**), ou on peut simplement dire **ArrayList** (grâce à l'instruction **import**).

Si on veut importer une seule classe, on peut la nommer dans l'instruction **import** :

```
import java.util.ArrayList;
```

Maintenant on peut utiliser **ArrayList** sans précision. Cependant, aucune des autres classes de **java.util** n'est disponible.

La raison de tous ces imports est de fournir un mécanisme pour gérer les "espaces de nommage" [*name spaces*]. Les noms de tous les membres de classe sont isolés les uns des autres. Une méthode **f()** dans une classe **A** ne sera pas en conflit avec une **f()** qui a la même signature (liste d'arguments) dans une classe **B**. Mais qu'en est-il des noms des classes ? Que se passe-t-il si on crée une classe **stack** qui est installée sur une machine qui a déjà une classe **stack** écrite par quelqu'un d'autre ? Avec Java sur Internet, ceci peut se passer sans que l'utilisateur le sache, puisque les classes peuvent être téléchargées automatiquement au cours de l'exécution d'un programme Java.

Ce conflit de noms potentiel est la raison pour laquelle il est important d'avoir un contrôle complet sur les espaces de nommage en Java, et d'être capable de créer des noms complètement uniques indépendamment des contraintes d'Internet.

Jusqu'ici, la plupart des exemples de ce livre étaient dans un seul fichier et ont été conçus pour un usage en local, et ne se sont pas occupés de noms de packages (dans ce cas le nom de la classe est placé dans le "package par défaut"). C'est certainement une possibilité, et dans un but de simplicité cette approche sera utilisée autant que possible dans le reste de ce livre. Cependant, si on envisage de créer des bibliothèques ou des programmes amicaux [*friendly*] vis-à-vis d'autres programmes sur la même machine, il faut penser à se prémunir des conflits de noms de classes.

Quand on crée un fichier source pour Java, il est couramment appelé une *unité de compilation* [*compilation unit*] (parfois une *unité de traduction* [*translation unit*]). Chaque unité de compilation doit avoir un nom se terminant par **.java**, et dans l'unité de compilation il peut y avoir une classe **public** qui doit avoir le même nom que le fichier (y compris les majuscules et minuscules, mais sans l'extension **.java**). Il ne peut y avoir qu'une seule classe **public** dans chaque unité de compilation, sinon le compilateur sortira une erreur. Le reste des classes de cette unité de compilation, s'il y en a, sont cachées du monde extérieur parce qu'elles ne sont *pas* **public**, elles sont des classes "support" pour la classe **public** principale.

Quand on compile un fichier **.java**, on obtient un fichier de sortie avec exactement le même nom mais avec une extension **.class** pour chaque classe du fichier **.java**. De ce fait on peut obtenir un nombre important de fichiers **.class** à partir d'un petit nombre de fichiers **.java**. Si vous avez programmé avec un langage compilé, vous avez sans doute remarqué que le compilateur génère un fichier de forme intermédiaire (généralement un fichier "obj") qui est ensuite assemblé avec d'autres fichiers de ce type à l'aide d'un éditeur de liens [*linker*] (pour créer un fichier exécutable) ou un "gestionnaire de bibliothèque" [*librarian*] (pour créer une bibliothèque). Ce n'est pas comme cela que Java travaille ; un programme exécutable est un ensemble de fichiers **.class**, qui peuvent être empaquetés [*packaged*] et compressés dans un fichier JAR (en utilisant l'archiveur Java **jar**). L'interpréteur Java est responsable de la recherche, du chargement et de l'interprétation de ces fichiers [\[32\]](#).

Une bibliothèque est aussi un ensemble de ces fichiers classes. Chaque fichier possède une classe qui est **public** (on n'est pas obligé d'avoir une classe **public**, mais c'est ce qu'on fait classiquement), de sorte qu'il y a un composant pour chaque fichier. Si on veut dire que tous ces composants (qui sont dans leurs propres fichiers **.java** et **.class** séparés) sont reliés entre eux, c'est là que le mot-clé **package** intervient.

Quand on dit :

```
package mypackage;
```

au début d'un fichier (si on utilise l'instruction **package**, elle *doit* apparaître à la première ligne du fichier, commentaires mis à part), on déclare que cette unité de compilation fait partie d'une bibliothèque appelée **mypackage**. Ou, dit autrement, cela signifie que le nom de la classe **public** dans cette unité de compilation est sous la couverture du nom **mypackage**, et si quelqu'un veut utiliser ce nom, il doit soit spécifier le nom complet, soit utiliser le mot-clé **import** en combinaison avec **mypackage** (utilisation des choix donnés précédemment). Remarquez que la convention pour les noms de packages Java est de n'utiliser que des lettres minuscules, même pour les mots intermédiaires.

Par exemple, supposons que le nom du fichier est **MyClass.java**. Ceci signifie qu'il peut y avoir une et une seule classe **public** dans ce fichier, et que le nom de cette classe doit être **MyClass** (y compris les majuscules et minuscules) :

```
package mypackage;

public class MyClass {

    // . . .
```

Maintenant, si quelqu'un veut utiliser **MyClass** ou, aussi bien, une des autres classes **public** de **mypackage**, il doit utiliser le mot-clé **import** pour avoir à sa disposition le ou les noms définis dans **mypackage**. L'autre solution est de donner le nom complet :

```
mypackage.MyClass m = new mypackage.MyClass();
```

Le mot-clé **import** peut rendre ceci beaucoup plus propre :

```
import mypackage.*;

// . . .

MyClass m = new MyClass();
```

Il faut garder en tête que ce que permettent les mots-clés **package** et **import**, en tant que concepteur de bibliothèque, c'est de diviser l'espace de nommage global afin d'éviter le conflit de nommages, indépendamment de combien il y a de personnes qui vont sur Internet écrire des classes en Java.

## Créer des noms de packages uniques

On pourrait faire remarquer que, comme un package n'est jamais réellement "empaqueté" [*packaged*] dans un

seul fichier, un package pourrait être fait de nombreux fichiers `.class` et les choses pourraient devenir un peu désordonnées. Pour éviter ceci, une chose logique à faire est de placer tous les fichiers `.class` d'un package donné dans un même répertoire ; c'est-à-dire utiliser à son avantage la structure hiérarchique des fichiers définie par le système d'exploitation. C'est un des moyens par lequel Java traite le problème du désordre ; on verra l'autre moyen plus tard lorsque l'utilitaire **jar** sera présenté.

Réunir les fichiers d'un package dans un même répertoire résoud deux autres problèmes : créer des noms de packages uniques, et trouver ces classes qui pourraient être enfouies quelque part dans la structure d'un répertoire. Ceci est réalisé, comme présenté dans le chapitre 2, en codant le chemin où se trouvent les fichiers `.class` dans le nom du **package**. Le compilateur force cela ; aussi, par convention, la première partie du nom de **package** est le nom de domaine Internet du créateur de la classe, renversé. Comme les noms de domaines Internet sont garantis uniques, si on suit cette convention, il est garanti que notre nom de **package** sera unique et donc qu'il n'y aura jamais de conflit de noms (c'est-à-dire, jusqu'à ce qu'on perde son nom de domaine au profit de quelqu'un qui commencerait à écrire du code Java avec les mêmes noms de répertoires). Bien entendu, si vous n'avez pas votre propre nom de domaine vous devez fabriquer une combinaison improbable (telle que votre prénom et votre nom) pour créer des noms de packages uniques. Si vous avez décidé de publier du code Java, cela vaut la peine d'effectuer l'effort relativement faible d'obtenir un nom de domaine.

La deuxième partie de cette astuce consiste à résoudre le nom de package à l'intérieur d'un répertoire de sa machine, de manière à ce que lorsque le programme Java s'exécute et qu'il a besoin de charger le fichier `.class` (ce qu'il fait dynamiquement, à l'endroit du programme où il doit créer un objet de cette classe particulière, ou la première fois qu'on accède à un membre static de la classe), il puisse localiser le répertoire où se trouve le fichier `.class`.

L'interpréteur Java procède de la manière suivante. D'abord il trouve la variable d'environnement `CLASSPATH` (positionnée à l'aide du système d'exploitation, parfois par le programme d'installation qui installe Java ou un outil Java sur votre machine). `CLASSPATH` contient un ou plusieurs répertoires qui sont utilisés comme racines de recherche pour les fichiers `.class`. En commençant à cette racine, l'interpréteur va prendre le nom de package et remplacer chaque point par un "slash" pour construire un nom de chemin depuis la racine `CLASSPATH` (de sorte que **package foo.bar.baz** devient **foo\bar\baz** ou **foo/bar/baz** ou peut-être quelque chose d'autre, selon votre système d'exploitation). Ceci est ensuite concaténé avec les diverses entrées du `CLASSPATH`. C'est là qu'il recherche le fichier `.class` portant le nom correspondant à la classe qu'on est en train d'essayer de créer (il recherche aussi certains répertoires standards relativement à l'endroit où se trouve l'interpréteur Java).

Pour comprendre ceci, prenons mon nom de domaine, qui est **bruceeckel.com**. En l'inversant, **com.bruceeckel** établit le nom global unique pour mes classes (les extensions `com`, `edu`, `org`, etc... étaient auparavant en majuscules dans les packages Java, mais ceci a été changé en Java 2 de sorte que le nom de package est entièrement en minuscules). Je peux ensuite subdiviser ceci en décidant de créer un répertoire simplement nommé **simple**, de façon à obtenir un nom de package :

```
package com.bruceeckel.simple;
```

Maintenant ce nom de package peut être utilisé comme un espace de nommage de couverture pour les deux fichiers suivants :

```
//: com:bruceeckel:simple:Vector.java  
// Création d'un package.  
package com.bruceeckel.simple;
```

```
public class Vector {  
    public Vector() {  
        System.out.println(  
            "com.bruceeckel.util.Vector");  
    }  
} ///:~
```

Lorsque vous créez vos propres packages, vous allez découvrir que l'instruction package doit être le premier code non-commentaire du fichier. Le deuxième fichier ressemble assez au premier :

```
///: com:bruceeckel:simple:List.java  
// Création d'un package.  
package com.bruceeckel.simple;  
public class List {  
    public List() {  
        System.out.println(  
            "com.bruceeckel.util.List");  
    }  
} ///:~
```

Chacun de ces fichiers est placé dans le sous-répertoire suivant dans mon système :

```
C:\DOC\JavaT\com\bruceeckel\simple
```

Dans ce nom de chemin, on peut voir le nom de package **com.bruceeckel.simple**, mais qu'en est-il de la première partie du chemin ? Elle est prise en compte dans la variable d'environnement CLASSPATH, qui est, sur ma machine :

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

On voit que le CLASSPATH peut contenir plusieurs chemins de recherche.

Il y a toutefois une variante lorsqu'on utilise des fichiers JAR. Il faut mettre également le nom du fichier JAR dans le classpath, et pas seulement le chemin où il se trouve. Donc pour un JAR nommé **grape.jar** le classpath doit contenir :

```
CLASSPATH=.;D:\JAVA\LIB;C:\flavors\grape.jar
```

Une fois le classpath défini correctement, le fichier suivant peut être placé dans n'importe quel répertoire :

```
//: c05:LibTest.java
// Utilise la bibliothèque.
import com.bruceeckel.simple.*;

public class LibTest {

    public static void main(String[] args) {

        Vector v = new Vector();

        List l = new List();

    }

} ///:~
```

Lorsque le compilateur rencontre l'instruction **import**, il commence à rechercher dans les répertoires spécifiés par CLASSPATH, recherchant le sous-répertoire com\bruceeckel\simple, puis recherchant les fichiers compilés de noms appropriés (Vector.class pour Vector et List.class pour List). Remarquez que chacune des classes et méthodes utilisées de **Vector** et **List** doivent être **public**.

Positionner le CLASSPATH a posé tellement de problèmes aux utilisateurs débutants de Java (ça l'a été pour moi quand j'ai démarré) que Sun a rendu le JDK un peu plus intelligent dans Java 2. Vous verrez, quand vous l'installerez, que vous pourrez compiler et exécuter des programmes Java de base même si vous ne positionnez pas de CLASSPATH. Pour compiler et exécuter le package des sources de ce livre (disponible sur le CD ROM livré avec ce livre, ou sur [www.BruceEckel.com](http://www.BruceEckel.com)), vous devrez cependant faire quelques modifications de votre CLASSPATH (celles-ci sont expliquées dans le package de sources).

## Collisions

Que se passe-t-il si deux bibliothèques sont importées à l'aide de \* et qu'elles contiennent les mêmes noms ? Par exemple, supposons qu'un programme fasse ceci :

```
import com.bruceeckel.simple.*;

import java.util.*;
```

Comme **java.util.\*** contient également une classe **Vector**, ceci crée une collision potentielle. Cependant, tant qu'on n'écrit pas le code qui cause effectivement la collision, tout va bien ; c'est une chance, sinon on pourrait se retrouver à écrire une quantité de code importante pour éviter des collisions qui n'arriveraient jamais.

La collision se produit si maintenant on essaye de créer un **Vector** :

```
Vector v = new Vector();
```

A quelle classe **Vector** ceci se réfère-t-il ? Le compilateur ne peut pas le savoir, et le lecteur non plus. Le compilateur se plaint et nous oblige à être explicite. Si je veux le **Vector** Java standard, par exemple, je dois dire :

```
java.util.Vector v =new java.util.Vector();
```

Comme ceci (avec le CLASSPATH) spécifie complètement l'emplacement de ce Vector, il n'y a pas besoin d'instruction **import java.util.\***, à moins que j'utilise autre chose dans **java.util**.

## Une bibliothèque d'outils personnalisée

Avec ces connaissances, vous pouvez maintenant créer vos propres bibliothèques d'outils pour réduire ou éliminer les duplications de code. On peut par exemple créer un alias pour **System.out.println( )** pour réduire la frappe. Ceci peut faire partie d'un package appelé **tools** :

```
//: com:bruceeckel:tools:P.java
// Les raccourcis P.rint & P.rintln
package com.bruceeckel.tools;
public class P {
    public static void rint(String s) {
        System.out.print(s);
    }
    public static void rintln(String s) {
        System.out.println(s);
    }
} ///:~
```

On peut utiliser ce raccourci pour imprimer une **String**, soit avec une nouvelle ligne (**P.rintln( )**), ou sans (**P.rint( )**).

Vous pouvez deviner que l'emplacement de ce fichier doit être dans un répertoire qui commence à un des répertoire du CLASSPATH, puis continue dans **com/bruceeckel/tools**. Après l'avoir compilé, le fichier **P.class** peut être utilisé partout sur votre système avec une instruction **import** :

```
//: c05:ToolTest.java
// Utilise la bibliothèque tools
import com.bruceeckel.tools.*;
public class ToolTest {
    public static void main(String[] args) {
        P.rintln("Available from now on!");
        P.rintln("" + 100); // Le force à être une String
        P.rintln("" + 100L);
    }
}
```

```

        P.println(" " + 3.14159);
    }
} ///:~

```

Remarquez que chacun des objets peut facilement être forcé dans une représentation **String** en les plaçant dans une expression **String** ; dans le cas ci-dessus, le fait de commencer l'expression avec une **String** vide fait l'affaire. Mais ceci amène une observation intéressante. Si on appelle **System.out.println(100)**, cela fonctionne sans le transformer *[cast]* en **String**. Avec un peu de surcharge *[overloading]*, on peut amener la classe **P** à faire ceci également (c'est un exercice à la fin de ce chapitre).

A partir de maintenant, chaque fois que vous avez un nouvel utilitaire intéressant, vous pouvez l'ajouter au répertoire **tools** (ou à votre propre répertoire **util** ou **tools**).

## Utilisation des imports pour modifier le comportement

Une caractéristique manquant à Java est la *compilation conditionnelle* du C, qui permet de changer un commutateur pour obtenir un comportement différent sans rien changer d'autre au code. La raison pour laquelle cette caractéristique n'a pas été retenue en Java est probablement qu'elle est surtout utilisée en C pour résoudre des problèmes de portabilité : des parties du code sont compilées différemment selon la plateforme pour laquelle le code est compilé. Comme le but de Java est d'être automatiquement portable, une telle caractéristique ne devrait pas être nécessaire.

Il y a cependant d'autres utilisations valables de la compilation conditionnelle. Un usage très courant est le debug de code. Les possibilités de debug sont validées pendant le développement, et invalidées dans le produit livré. Allen Holub ([www.holub.com](http://www.holub.com)) a eu l'idée d'utiliser les packages pour imiter la compilation conditionnelle. Il l'a utilisée pour créer une version du très utile *mécanisme d'affirmation* *[assertion mechanism]* du C, dans lequel on peut dire "ceci devrait être vrai" ou "ceci devrait être faux" et si l'instruction n'est pas d'accord avec cette affirmation, on en sera averti. Un tel outil est très utile pendant la phase de debuggage.

Voici une classe qu'on utilisera pour debugger :

```

///: com:bruceeckel:tools:debug:Assert.java

// Outil d'affirmation pour debugger

package com.bruceeckel.tools.debug;

public class Assert {

    private static void perr(String msg) {

        System.err.println(msg);

    }

    public final static void is_true(boolean exp) {

        if(!exp) perr("Assertion failed");

    }

    public final static void is_false(boolean exp){

        if(exp) perr("Assertion failed");

    }

}

```



```

    }

    public final static void
    is_true(boolean exp, String msg) {
        if(!exp) perr("Assertion failed: " + msg);
    }

    public final static void
    is_false(boolean exp, String msg) {
        if(exp) perr("Assertion failed: " + msg);
    }
} ///:~

```

Cette classe encapsule simplement des tests booléens, qui impriment des messages d'erreur en cas d'échec. Dans le chapitre 10, on apprendra à utiliser un outil plus sophistiqué pour traiter les erreurs, appelé *traitement d'exceptions* [*exception handling*], mais la méthode **perr()** conviendra bien en attendant.

La sortie est affichée sur la console dans le flux d'erreur *standard* [*standard error stream*] en écrivant avec **System.err**.

Pour utiliser cette classe, ajoutez cette ligne à votre programme :

```
import com.bruceeckel.tools.debug.*;
```

Pour supprimer les affirmations afin de pouvoir livrer le code, une deuxième classe **Assert** est créée, mais dans un package différent :

```

//: com:bruceeckel:tools:Assert.java
// Suppression de la sortie de l'affirmation
// pour pouvoir livrer le programme.

package com.bruceeckel.tools;

public class Assert {

    public final static void is_true(boolean exp){}

    public final static void is_false(boolean exp){}

    public final static void
    is_true(boolean exp, String msg) {}

    public final static void
    is_false(boolean exp, String msg) {}
} ///:~

```

Maintenant si on change l'instruction **import** précédente en :

```
import com.bruceeckel.tools.*;
```

le programme n'affichera plus d'affirmations. Voici un exemple :

```
///  
// Démonstration de l'outil d'affirmation.  
// Mettre en commentaires ce qui suit, et enlever  
// le commentaire de la ligne suivante  
// pour modifier le comportement de l'affirmation :  
import com.bruceeckel.tools.debug.*;  
// import com.bruceeckel.tools.*;  
public class TestAssert {  
    public static void main(String[] args) {  
        Assert.isTrue((2 + 2) == 5);  
        Assert.isFalse((1 + 1) == 2);  
        Assert.isTrue((2 + 2) == 5, "2 + 2 == 5");  
        Assert.isFalse((1 + 1) == 2, "1 + 1 != 2");  
    }  
} ///  
}
```

En changeant le package qui est importé, on change le code de la version debug à la version de production. Cette technique peut être utilisée pour toutes sortes de code conditionnel.

## Avertissement sur les packages

Il faut se rappeler que chaque fois qu'on crée un package, on spécifie implicitement une structure de répertoire en donnant un nom à un package. Le package *doit* se trouver dans le répertoire indiqué par son nom, qui doit être un répertoire qui peut être trouvé en partant du CLASSPATH. L'utilisation du mot-clé **package** peut être un peu frustrante au début, car à moins d'adhérer à la règle nom-de-package - chemin-de-répertoire, on obtient un tas de messages mystérieux à l'exécution signalant l'impossibilité de trouver une classe donnée, même si la classe est là dans le même répertoire. Si vous obtenez un message de ce type, essayez de mettre en commentaire l'instruction **package**, et si ça tourne vous saurez où se trouve le problème.

## Les spécificateurs d'accès Java

Quand on les utilise, les spécificateurs d'accès Java **public**, **protected**, et **private** sont placés devant la définition de chaque membre de votre classe, qu'il s'agisse d'un champ ou d'une méthode. Chaque spécificateur d'accès contrôle l'accès pour uniquement cette définition particulière. Ceci est différent du C++, où un spécificateur d'accès contrôle toutes les définitions le suivant jusqu'à ce qu'un autre spécificateur d'accès soit

rencontré.

D'une façon ou d'une autre, toute chose a un type d'accès spécifié. Dans les sections suivantes, vous apprendrez à utiliser les différents types d'accès, à commencer par l'accès par défaut.

## "Friendly"

Que se passe-t-il si on ne précise aucun spécificateur d'accès, comme dans tous les exemples avant ce chapitre ? L'accès par défaut n'a pas de mot-clé, mais on l'appelle couramment "friendly" (amical). Cela veut dire que toutes les autres classes du package courant ont accès au membre amical, mais pour toutes les classes hors du package le membre apparaît **private**. Comme une unité de compilation -un fichier- ne peut appartenir qu'à un seul package, toutes les classes d'une unité de compilation sont automatiquement amicales entre elles. De ce fait, on dit aussi que les éléments amicaux ont un *accès de package* [*package access*].

L'accès amical permet de grouper des classes ayant des points communs dans un package, afin qu'elles puissent facilement interagir entre elles. Quand on met des classes ensemble dans un package (leur accordant de ce fait un accès mutuel à leurs membres amicaux, c'est à dire en les rendant "amicaux") on "possède" le code de ce package. Il est logique que seul le code qu'on possède ait un accès amical à un autre code qu'on possède. On pourrait dire que l'accès amical donne une signification ou une raison pour regrouper des classes dans un package. Dans beaucoup de langages l'organisation des définitions dans des fichiers peut être faite tant bien que mal, mais en Java on est astreint à les organiser d'une manière logique. De plus, on exclura probablement les classes qui ne devraient pas avoir accès aux classes définies dans le package courant.

La classe contrôle quel code a accès à ses membres. Il n'y a pas de moyen magique pour "entrer par effraction". Le code d'un autre package ne peut pas dire "Bonjour, je suis un ami de **Bob** !" et voir les membres **protected**, friendly, et **private** de **Bob**. La seule façon d'accorder l'accès à un membre est de :

1. Rendre le membre **public**. Ainsi tout le monde, partout, peut y accéder.
2. Rendre le membre amical en n'utilisant aucun spécificateur d'accès, et mettre les autres classes dans le même package. Ainsi les autres classes peuvent accéder à ce membre.
3. Comme on le verra au Chapitre 6, lorsque l'héritage sera présenté, une classe héritée peut avoir accès à un membre **protected** ou **public** (mais pas à des membres **private**). Elle peut accéder à des membres amicaux seulement si les deux classes sont dans le même package. Mais vous n'avez pas besoin de vous occuper de cela maintenant.
4. Fournir des méthodes "accessor/mutator" (connues aussi sous le nom de méthodes "get/set") qui lisent et changent la valeur. Ceci est l'approche la plus civilisée en termes de programmation orientée objet, et elle est fondamentale pour les JavaBeans, comme on le verra dans le Chapitre 13.

## public : accès d'interface

Lorsqu'on utilise le mot-clé **public**, cela signifie que la déclaration du membre qui suit immédiatement **public** est disponible pour tout le monde, en particulier pour le programmeur client qui utilise la bibliothèque. Supposons qu'on définisse un package **dessert** contenant l'unité de compilation suivante :

```
//: c05:dessert:Cookie.java
// Création d'une bibliothèque.

package c05.dessert;

public class Cookie {

    public Cookie() {
```

```

        System.out.println("Cookie constructor");
    }

    void bite() { System.out.println("bite"); }
} ///:~

```

Souvenez-vous, **Cookie.java** doit se trouver dans un sous-répertoire appelé **dessert**, en dessous de **c05** (qui signifie le Chapitre 5 de ce livre) qui doit être en-dessous d'un des répertoire du CLASSPATH. Ne faites pas l'erreur de croire que Java va toujours considérer le répertoire courant comme l'un des points de départ de la recherche. Si vous n'avez pas un "." comme un des chemins dans votre CLASSPATH, Java n'y regardera pas.

Maintenant si on crée un programme qui utilise **Cookie** :

```

///: c05:Dinner.java
// Utilise la bibliothèque.
import c05.dessert.*;

public class Dinner {
    public Dinner() {
        System.out.println("Dinner constructor");
    }

    public static void main(String[] args) {
        Cookie x = new Cookie();

        ///! x.bite(); // Ne peut y accéder
    }
} ///:~

```

on peut créer un objet **Cookie**, puisque son constructeur est **public** et que la classe est **public**. (Nous allons regarder plus en détail le concept d'une classe **public** plus tard.) Cependant, le membre **bite()** est inaccessible depuis **Dinner.java** car **bite()** est amical uniquement à l'intérieur du package **dessert**.

## Le package par défaut

Vous pourriez être surpris de découvrir que le code suivant compile, bien qu'il semble ne pas suivre les règles :

```

///: c05:Cake.java
// Accède à une classe dans
// une unité de compilation séparée.

class Cake {
    public static void main(String[] args) {

```

```

    Pie x = new Pie();

    x.f();
}
} ///:~

```

Dans un deuxième fichier, dans le même répertoire :

```

///: c05:Pie.java

// L'autre classe.

class Pie {

    void f() { System.out.println("Pie.f()"); }

} ///:~

```

On pourrait à première vue considérer ces fichiers comme complètement étrangers l'un à l'autre, et cependant **Cake** est capable de créer l'objet **Pie** et d'appeler sa méthode **f()** ! (Remarquez qu'il faut avoir "." dans le CLASSPATH pour que ces fichiers puissent être compilés.) On penserait normalement que **Pie** et **f()** sont amicaux et donc non accessibles par **Cake**. Ils *sont* amicaux, cette partie-là est exacte. La raison pour laquelle ils sont accessibles dans **Cake.java** est qu'ils sont dans le même répertoire et qu'ils n'ont pas de nom de package explicite. Java traite de tels fichiers comme faisant partie du "package par défaut" pour ce répertoire, et donc amical pour tous les autres fichiers du répertoire.

## private : ne pas toucher !

Le mot-clé **private** signifie que personne ne peut accéder à ce membre à part la classe en question, dans les méthodes de cette classe. Les autres classes du package ne peuvent accéder à des membres **private**, et c'est donc un peu comme si on protégeait la classe contre soi-même. D'un autre côté il n'est pas impossible qu'un package soit codé par plusieurs personnes, et dans ce cas **private** permet de modifier librement ce membre sans que cela affecte une autre classe dans le même package.

L'accès "amical" par défaut dans un package cache souvent suffisamment les choses ; souvenez-vous, un membre "amical" est inaccessible à l'utilisateur du package. C'est parfait puisque l'accès par défaut est celui qu'on utilise normalement (et c'est celui qu'on obtient si on oublie d'ajouter un contrôle d'accès). De ce fait, on ne pensera normalement qu'aux accès aux membres qu'on veut explicitement rendre **public** pour le programmeur client, et donc on pourrait penser qu'on n'utilise pas souvent le mot-clé **private** puisqu'on peut s'en passer (ceci est une différence par rapport au C++). Cependant, il se fait que l'utilisation cohérente de **private** est très importante, particulièrement lors du multithreading (comme on le verra au Chapitre 14).

Voici un exemple de l'utilisation de **private** :

```

///: c05:IceCream.java

// Démontre le mot-clé "private"

class Sundae {

    private Sundae() {}
}

```

```

    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        //! Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
} ///~

```

Ceci montre un cas où **private** vient à propos : on peut vouloir contrôler comment un objet est créé et empêcher quelqu'un d'accéder à un constructeur en particulier (ou à tous). Dans l'exemple ci-dessus, on ne peut pas créer un objet **Sundae** à l'aide de son constructeur ; il faut plutôt utiliser la méthode **makeASundae()** qui le fera pour nous [\[33\]](#).

Toute méthode dont on est certain qu'elle n'est utile qu'à cette classe peut être rendue **private**, pour s'assurer qu'on ne l'utilisera pas ailleurs dans le package, nous interdisant ainsi de la modifier ou de la supprimer. Rendre une méthode **private** garantit cela.

Ceci est également vrai pour un champ **private** dans une classe. A moins qu'on ne doive exposer une implémentation sous-jacente (ce qui est beaucoup plus rare qu'on ne pourrait penser), on devrait déclarer tous les membres **private**. Cependant, le fait que la référence à un objet est **private** dans une classe ne signifie pas qu'un autre objet ne puisse avoir une référence **public** à cet objet (voir l'annexe A pour les problèmes au sujet de l'aliasing).

## protected : "sorte d'amical"

Le spécificateur d'accès **protected** demande un effort de compréhension . Tout d'abord, il faut savoir que vous n'avez pas besoin de comprendre cette partie pour continuer ce livre jusqu'à l'héritage (Chapitre 6). Mais pour être complet, voici une brève description et un exemple d'utilisation de **protected**.

Le mot-clé **protected** traite un concept appelé *héritage*, qui prend une classe existante et ajoute des membres à cette classe sans modifier la classe existante, que nous appellerons la *classe de base*. On peut également changer le comportement des membres d'une classe existants. Pour hériter d'une classe existante, on dit que le nouvelle classe **extends** (étend) une classe existante, comme ceci :

```
class Foo extends Bar {
```

Le reste de la définition de la classe est inchangé.

Si on crée un nouveau package et qu'on hérite d'une classe d'un autre package, les seuls membres accessibles sont les membres **public** du package d'origine. (Bien sûr, si on effectue l'héritage dans le *même* package, on a l'accès normal à tous les membres "amicaux" du package.) Parfois le créateur de la classe de base veut, pour un

membre particulier, en accorder l'accès dans les classes dérivées mais pas dans le monde entier en général. C'est ce que **protected** fait. Si on reprend le fichier **Cookie.java**, la classe suivante ne peut pas accéder au membre "amical" :

```
//: c05:ChocolateChip.java
// Ne peut pas accéder à un membre amical
// dans une autre classe.
import c05.dessert.*;

public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println(
            "ChocolateChip constructor");
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        ///! x.bite(); // Ne peut pas accéder à bite
    }
} ///:~
```

Une des particularités intéressantes de l'héritage est que si la méthode **bite()** existe dans la classe **Cookie**, alors elle existe aussi dans toute classe héritée de **Cookie**. Mais comme **bite()** est "amical" dans un autre package, il nous est inaccessible dans celui-ci. Bien sûr, on pourrait le rendre **public**, mais alors tout le monde y aurait accès, ce qui ne serait peut-être pas ce qu'on veut. Si on modifie la classe **Cookie** comme ceci :

```
public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    protected void bite() {
        System.out.println("bite");
    }
}
```

alors **bite()** est toujours d'accès "amical" dans le package **dessert**, mais il est aussi accessible à tous ceux qui héritent de **Cookie**. Cependant, il n'est pas **public**.

# Interface et implémentation

Le contrôle d'accès est souvent appelé *cacher l'implémentation* [*implementation hiding*] . L'enveloppement [*wrapping*] des données et méthodes à l'intérieur des classes combiné au masquage de l'implémentation est souvent appelé *encapsulation* [34]. Le résultat est un type de données avec des caractéristiques et des comportements.

Le contrôle d'accès pose des limites sur un type de données pour deux raisons importantes. La première est de déterminer ce que les programmeurs clients peuvent et ne peuvent pas utiliser. On peut construire les mécanismes internes dans la structure sans se préoccuper du risque que les programmeurs clients prennent ces mécanismes internes comme faisant partie de l'interface qu'ils doivent utiliser.

Ceci amène directement à la deuxième raison, qui est de séparer l'interface de son implémentation. Si la structure est utilisée dans un ensemble de programmes, mais que les programmeurs clients ne peuvent qu'envoyer des messages à l'interface **public**, alors on peut modifier tout ce qui *n'est pas public* (c'est à dire "amical", **protected**, ou **private**) sans que cela nécessite des modifications du code client.

Nous sommes maintenant dans le monde de la programmation orientée objet, dans lequel une **class** est en fait la description d' "une classe d'objets", comme on décrirait une classe des poissons ou une classe des oiseaux. Tout objet appartenant à cette classe partage ces caractéristiques et comportements. La classe est une description de la façon dont les objets de ce type vont nous apparaître et se comporter.

Dans le langage de POO d'origine, Simula-67 , le mot-clé **class** était utilisé pour décrire un nouveau type de données. Le même mot-clé a été repris dans la plupart des langages orientés objet. Ceci est le point central de tout le langage : la création de nouveaux types de données qui sont plus que simplement des boîtes contenant des données et des méthodes.

La classe est le concept de POO fondamental en Java. C'est l'un des mots-clés qui *ne sera pas* mis en gras dans ce livre, ça devient lourd pour un mot aussi souvent répété que "class".

Pour plus de clarté, il est préférable d'utiliser un style de création des classes qui place les membres **public** au début, suivi par les membres **protected**, amicaux et **private**. L'avantage de ceci est que l'utilisateur de la classe peut voir ce qui est important pour lui (les membres **public**, parce qu'on peut y accéder de l'extérieur du fichier), en lisant depuis le début et en s'arrêtant lorsqu'il rencontre les membres non-**public**, qui font partie de l'implémentation interne :

```
public class X {  
    public void pub1( ) { /* . . . */ }  
    public void pub2( ) { /* . . . */ }  
    public void pub3( ) { /* . . . */ }  
    private void priv1( ) { /* . . . */ }  
    private void priv2( ) { /* . . . */ }  
    private void priv3( ) { /* . . . */ }  
    private int i;  
    // . . .  
}
```



```
}
```

Ceci ne la rendra que partiellement plus lisible parce que l'interface et l'implémentation sont encore mélangés. C'est-à-dire qu'on voit toujours le code source (l'implémentation) parce qu'il est là dans la classe. Cependant, la documentation sous forme de commentaires supportée par javadoc (décrite au Chapitre 2) diminue l'importance de la lisibilité du code par le programmeur client. Afficher l'interface au consommateur d'une classe est normalement le travail du *class browser*, un outil dont le travail consiste à inspecter toutes les classes disponibles et de montrer ce qu'on peut en faire (c'est à dire quels membres sont disponibles) de façon pratique. Au moment où vous lisez ceci, les browsers devraient faire partie de tout bon outil de développement Java.

## L'accès aux classes

En Java, les spécificateurs d'accès peuvent aussi être utilisés pour déterminer quelles classes d'une bibliothèque seront accessibles aux utilisateurs de cette bibliothèque. Si on désire qu'une classe soit disponible pour un programmeur client, on place le mot-clé **public** quelque part devant l'accolade ouvrante du corps de la classe. Ceci permet de contrôler le fait même qu'un programmeur client puisse créer un objet de cette classe.

Pour contrôler l'accès à la classe, le spécificateur doit apparaître avant le mot-clé **class**. Donc on peut dire :

```
public class Widget {
```

Maintenant, si le nom de la bibliothèque est **mylib**, tout programmeur client peut accéder à **Widget** en disant

```
import mylib.Widget;
```

ou

```
import mylib.*;
```

Il y a cependant un ensemble de contraintes supplémentaires :

1. Il ne peut y avoir qu'une seule classe **public** par unité de compilation (fichier). L'idée est que chaque unité de compilation a une seule interface publique représentée par cette classe **public**. Elle peut avoir autant de classes "amicales" de support qu'on veut. Si on a plus d'une classe **public** dans une unité de compilation, le compilateur générera un message d'erreur.
2. Le nom de la classe **public** doit correspondre exactement au nom du fichier contenant l'unité de compilation, y compris les majuscules et minuscules. Par exemple pour **Widget**, le nom du fichier doit être **Widget.java**, et pas **widget.java** ou **WIDGET.java**. Là aussi on obtient des erreurs de compilation s'ils ne correspondent pas.
3. Il est possible, bien que non habituel, d'avoir une unité de compilation sans aucune classe **public**. Dans ce cas, on peut appeler le fichier comme on veut.

Que se passe-t-il si on a une classe dans **mylib** qu'on utilise uniquement pour accomplir les tâches effectuées par **Widget** ou une autre classe public de **mylib** ? On ne veut pas créer de documentation pour un programmeur client, et on pense que peut-être plus tard on modifiera tout et qu'on refera toute la classe en lui en substituant une nouvelle. Pour garder cette possibilité, il faut s'assurer qu'aucun programmeur client ne devienne dépendant

des détails d'implémentation cachés dans **mylib**. Pour réaliser ceci il suffit d'enlever le mot-clé **public** de la classe, qui devient dans ce cas amicale. (Cette classe ne peut être utilisée que dans ce package.)

Remarquez qu'une classe ne peut pas être **private** (cela ne la rendrait accessible à personne d'autre que cette classe), ou **protected** [35]. Il n'y a donc que deux choix pour l'accès aux classes : "amical" ou **public**. Si on ne veut pas que quelqu'un d'autre accède à cette classe, on peut rendre tous les constructeurs **private**, ce qui empêche tout le monde de créer un objet de cette classe, à part soi-même dans un membre static de la classe [36]. Voici un exemple :

```
//: c05:Lunch.java

// Démontre les spécificateurs d'accès de classes.

// Faire une classe effectivement private
// avec des constructeurs private :

class Soup {

    private Soup() {}

    // (1) Permettre la création à l'aide d'une méthode static :

    public static Soup makeSoup() {

        return new Soup();

    }

    // (2) Créer un objet static et
    // retourner une référence à la demande.
    // (le patron "Singleton"):

    private static Soup ps1 = new Soup();

    public static Soup access() {

        return ps1;

    }

    public void f() {}

}

class Sandwich { // Utilise Lunch

    void f() { new Lunch(); }

}

// Une seule classe public autorisée par fichier :

public class Lunch {

    void test() {

        // Ne peut pas faire ceci ! Constructeur privé :
```

```

    !! Soup priv1 = new Soup();

    Soup priv2 = Soup.makeSoup();

    Sandwich f1 = new Sandwich();

    Soup.access().f();

}

!~

```

Jusqu'ici, la plupart des méthodes retournaient soit **void** soit un type primitif, ce qui fait que la définition :

```

public static Soup access() {

    return ps1;

}

```

pourrait paraître un peu confuse. Le mot devant le nom de la méthode (**access**) dit ce que retourne la méthode. Jusqu'ici cela a été le plus souvent **void**, ce qui signifie qu'elle ne retourne rien. Mais on peut aussi retourner la référence à un objet, comme c'est le cas ici. Cette méthode retourne une référence à un objet de la classe **Soup**.

La classe **Soup** montre comment empêcher la création directe d'une classe en rendant tous les constructeurs **private**. Souvenez-vous que si vous ne créez pas explicitement au moins un constructeur, le constructeur par défaut (un constructeur sans arguments) sera créé pour vous. En écrivant ce constructeur par défaut, il ne sera pas créé automatiquement. En le rendant **private**, personne ne pourra créer un objet de cette classe. Mais alors comment utilise-t-on cette classe ? L'exemple ci-dessus montre deux possibilités. Premièrement, une méthode **static** est créée, elle crée un nouveau **Soup** et en retourne la référence. Ceci peut être utile si on veut faire des opérations supplémentaires sur **Soup** avant de le retourner, ou si on veut garder un compteur du nombre d'objets **Soup** créés (peut-être pour restreindre leur population).

La seconde possibilité utilise ce qu'on appelle un *patron de conception [design pattern]*, qui est expliqué dans *Thinking in Patterns with Java*, téléchargeable sur [www.BruceEckel.com](http://www.BruceEckel.com). Ce patron particulier est appelé un "singleton" parce qu'il n'autorise la création que d'un seul objet. L'objet de classe **Soup** est créé comme un membre **static private** de **Soup**, ce qui fait qu'il n'y en a qu'un seul, et on ne peut y accéder qu'à travers la méthode **public access()**.

Comme mentionné précédemment, si on ne met pas de spécificateur d'accès il est "amical" par défaut. Ceci signifie qu'un objet de cette classe peut être créé par toute autre classe du package, mais pas en dehors du package (souvenez-vous que tous les fichiers dans le même répertoire qui n'ont pas de déclaration **package** explicite font implicitement partie du package par défaut pour ce répertoire). Cependant, si un membre **static** de cette classe est **public**, le programmeur client peut encore accéder à ce membre **static** même s'il ne peut pas créer un objet de cette classe.

## Résumé

Dans toute relation il est important d'avoir des limites respectées par toutes les parties concernées. Lorsqu'on crée une bibliothèque, on établit une relation avec l'utilisateur de cette bibliothèque (le programmeur client) qui est un autre programmeur, mais qui construit une application ou qui utilise la bibliothèque pour créer une bibliothèque plus grande.

Sans règles, les programmeurs clients peuvent faire tout ce qu'ils veulent des membres de la classe, même si vous préféreriez qu'ils ne manipulent pas directement certains des membres. Tout est à découvert dans le monde entier.

Ce chapitre a décrit comment les classes sont construites pour former des bibliothèques ; d'abord, comment un groupe de classes est empaqueté [*packaged*] dans une bibliothèque, et ensuite comment la classe contrôle l'accès à ses membres.

On estime qu'un projet programmé en C commence à s'écrouler lorsqu'il atteint 50K à 100K de lignes de code, parce que le C a un seul "espace de nommage", et donc les noms commencent à entrer en conflit, provoquant un surcroît de gestion. En Java, le mot-clé **package**, le principe de nommage des packages et le mot-clé **import** donnent un contrôle complet sur les noms, et le problème de conflit de nommage est facilement évité.

Il y a deux raisons pour contrôler l'accès aux membres. La première est d'écarter les utilisateurs des outils qu'ils ne doivent pas utiliser ; outils qui sont nécessaires pour les traitements internes des types de données, mais qui ne font pas partie de l'interface dont les utilisateurs ont besoin pour résoudre leurs problèmes particuliers. Donc créer des méthodes et des champs **private** est un service rendu aux utilisateurs parce qu'ils peuvent facilement voir ce qui est important pour eux et ce qu'ils peuvent ignorer. Cela leur simplifie la compréhension de la classe.

La deuxième raison, et la plus importante, pour le contrôle d'accès, est de permettre au concepteur de bibliothèque de modifier les fonctionnements internes de la classe sans se soucier de la façon dont cela peut affecter le programmeur client. On peut construire une classe d'une façon, et ensuite découvrir qu'une restructuration du code améliorera grandement sa vitesse. Si l'interface et l'implémentation sont clairement séparées et protégées, on peut y arriver sans forcer l'utilisateur à réécrire son code.

Les spécificateurs d'accès en Java donnent un contrôle précieux au créateur d'une **class**. Les utilisateurs de la **class** peuvent voir clairement et exactement ce qu'ils peuvent utiliser et ce qu'ils peuvent ignorer. Encore plus important, on a la possibilité de garantir qu'aucun utilisateur ne deviendra dépendant d'une quelconque partie de l'implémentation d'une **class**. Sachant cela en tant que créateur d'une **class**, on peut en modifier l'implémentation en sachant qu'aucun programmeur client ne sera affecté par les modifications car il ne peut accéder à cette partie de la **class**.

Lorsqu'on a la possibilité de modifier l'implémentation, on peut non seulement améliorer la conception plus tard, mais on a aussi le droit de faire des erreurs. Quelles que soient les soins que vous apportez à votre planning et à votre conception, vous ferez des erreurs. Savoir qu'il est relativement peu dangereux de faire ces erreurs veut dire que vous ferez plus d'expériences, vous apprendrez plus vite, et vous finirez plus vite votre projet.

L'interface publique d'une classe est ce que l'utilisateur *voit*, donc c'est la partie qui doit être "correcte" lors de l'analyse et la conception. Et même ici vous avez encore quelques possibilités de modifications. Si vous n'avez pas la bonne interface du premier coup, vous pouvez *ajouter* des méthodes, pour autant que vous ne supprimiez pas celles que les programmeurs clients ont déjà utilisé dans leur code.

## Exercices

Les solutions des exercices sélectionnés sont disponibles dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible à un prix raisonnable sur [www.BruceEckel.com](http://www.BruceEckel.com).

1. Ecrivez un programme qui crée un objet ArrayList sans importer explicitement **java.util.\***.
2. Dans la section nommée "package : l'unité de bibliothèque", transformez les fragments de code concernant **mypackage** en un ensemble de fichiers Java qui peuvent être compilés et qui tournent.

3. Dans la section appelée "Collisions", prenez les fragments de code et transformez les en programme, et vérifiez qu'en fait les collisions arrivent.
4. Généralisez la classe `P` définie dans ce chapitre en ajoutant toutes les versions surchargées de `rint()` et `rintln()` nécessaires pour gérer tous les types Java de base.
5. Modifiez l'instruction `import` de `TestAssert.java` pour autoriser ou inhiber le mécanisme d'assertion.
6. Créez une classe avec **public**, **private**, **protected**, et des membres de données et des méthodes "amicaux". Faites attention au fait que des classes dans un même répertoire font partie d'un package "par défaut".
7. Créez une classe avec des données **protected**. Créez une deuxième classe dans le même fichier, qui a une méthode qui manipule les données **protected** de la première classe.
8. Modifiez la classe `Cookie` comme spécifié dans la section "**protected : sorte d'amical**". Vérifiez que `bite()` n'est pas **public**.
9. Dans la section nommée "Accès aux classes" vous trouverez des fragments de code décrivant **mylib** et **Widget**. Créez cette bibliothèque, et ensuite créez un **Widget** dans une classe qui ne fait pas partie du package **mylib**.
10. Créez un nouveau répertoire et modifiez votre `CLASSPATH` pour inclure ce nouveau répertoire. Copiez le fichier `P.class` (produit par la compilation de `com.bruceeckel.tools.P.java`) dans votre nouveau répertoire et changez ensuite le nom du fichier, la classe `P` à l'intérieur, et les noms de méthodes (vous pouvez aussi ajouter des sorties supplémentaires pour observer comment cela fonctionne). Créez un autre programme dans un autre répertoire, qui utilise votre nouvelle classe.
11. En suivant la forme de l'exemple `Lunch.java`, créez une classe appelée **ConnectionManager** qui gère un tableau fixe d'objets **Connection**. Le programmeur client ne doit pas pouvoir créer explicitement des objets **Connection**, mais doit seulement pouvoir les obtenir à l'aide d'une méthode static dans **ConnectionManager**. Lorsque le **ConnectionManager** tombe à court d'objets, il retourne une référence **null**. Testez la classe dans `main()`.
12. Créez le fichier suivant dans le répertoire `c05/local` (supposé être dans votre `CLASSPATH`) :
- 13.

```

///c05:local:PackagedClass.java

package c05.local;

class PackagedClass {

    public PackagedClass() {

        System.out.println(

            "Creating a packaged class");

    }

}///::~

```

Créez ensuite le fichier suivant dans un répertoire autre que `c05` :

```

///c05:foreign:Foreign.java

package c05.foreign;

import c05.local.*;

public class Foreign {

    public static void main (String[] args) {

        PackagedClass pc = new PackagedClass();
    }
}

```

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

Expliquez pourquoi le compilateur génère une erreur. Le fait de mettre la classe **Foreign** dans le package **c05.local** changerait-il quelque chose ?

---

[32] Rien en Java n'oblige à utiliser un interpréteur. Il existe des compilateurs Java de code natif qui génèrent un seul fichier exécutable.

[33] Il y a un autre effet dans ce cas. Comme le constructeur par défaut est le seul défini, et qu'il est **private**, il empêchera l'héritage de cette classe. (Un sujet qui sera présenté dans le Chapitre 6.)

[34] Cependant, on parle souvent aussi d'encapsulation pour le seul fait de cacher l'implémentation.

[35] En fait, une *classe interne* [*inner class*] peut être **private** ou **protected**, mais il s'agit d'un cas particulier. Ceux-ci seront présentés au Chapitre 7.

[36] On peut aussi le faire en héritant (Chapitre 6) de cette classe.