

Traducteur : Jérôme QUELIN

28.04.2001 - version 5.3 :

- Mise en forme html.

16.07.2000 - version 5.2

- Corrections apportées par Jean-Pierre Vidal.

14.07.2000 - version 5.1

- Première publication sur eGroups.

9 : Stockage des objets

C'est un programme relativement simple que celui qui ne manipule que des objets dont le nombre et la durée de vie sont connus à l'avance.

Mais en général, vos programmes créeront de nouveaux objets basés sur des informations qui ne seront pas connues avant le lancement du programme. Le nombre voire même le type des objets nécessaires ne seront pas connus avant la phase d'exécution du programme. Pour résoudre le problème considéré, il faut donc être capable de créer un certain nombre d'objets, n'importe quand, n'importe où. Ce qui implique qu'on ne peut se contenter d'une référence nommée pour stocker chacun des objets du programme :

```
MyObject myReference;
```

puisque'on ne connaît pas le nombre exact de références qui seront manipulées.

Pour résoudre ce problème fondamental, Java dispose de plusieurs manières de stocker les objets (ou plus exactement les références sur les objets). Le type interne est le tableau, dont nous avons déjà parlé auparavant. De plus, la bibliothèque des utilitaires de Java propose un ensemble relativement complet de *classes conteneurs* (aussi connues sous le nom de *classes collections*, mais comme les bibliothèques de Java 2 utilisent le nom **Collection** pour un sous-ensemble particulier de cette bibliothèque, j'utiliserai ici le terme plus générique « conteneur »). Les conteneurs fournissent des moyens sophistiqués pour stocker et même manipuler les objets d'un programme.

Les tableaux

Les tableaux ont déjà été introduits dans la dernière section du Chapitre 4, qui montrait comment définir et initialiser un tableau. Ce chapitre traite du stockage des objets, et un tableau n'est ni plus ni moins qu'un moyen de stocker des objets. Mais il existe de nombreuses autres manières de stocker des objets : qu'est-ce qui rend donc les tableaux si spécial ?

Les tableaux se distinguent des autres types de conteneurs sur deux points : l'efficacité et le type. Un tableau constitue la manière la plus efficace que propose Java pour stocker et accéder aléatoirement à une séquence d'objets (en fait, de références sur des objets). Un tableau est une simple séquence linéaire, ce qui rend l'accès aux éléments extrêmement rapide ; mais cette rapidité se paye : la taille d'un tableau est fixée lors de sa création et ne peut être plus être changée pendant toute la durée de sa vie. Une solution est de créer un tableau d'une taille donnée, et, lorsque celui-ci est saturé, en créer un nouveau et déplacer toutes les références de l'ancien

tableau dans le nouveau. C'est précisément ce que fait la classe **ArrayList**, qui sera étudiée plus loin dans ce chapitre. Cependant, du fait du surcoût engendré par la flexibilité apportée au niveau de la taille, une **ArrayList** est beaucoup moins efficace qu'un tableau.

La classe conteneur **vector** en C++ *connaît* le type des objets qu'il stocke, mais il a un inconvénient comparé aux tableaux de Java : l'**opérateur** `[]` des **vector** C++ ne réalise pas de contrôle sur les indices, on peut donc tenter d'accéder à un élément au-delà de la taille du **vector** [44]. En Java, un contrôle d'indices est automatiquement effectué, qu'on utilise un tableau ou un conteneur - une exception **RuntimeException** est générée si les frontières sont dépassées. Comme vous le verrez dans le Chapitre 10, ce type d'exception indique une erreur due au programmeur, et comme telle il ne faut pas la prendre en considération dans le code. Bien entendu, le **vector** C++ n'effectue pas de vérifications à chaque accès pour des raisons d'efficacité - en Java, la vérification continue des frontières implique une dégradation des performances pour les tableaux comme pour les conteneurs.

Les autres classes de conteneurs génériques qui seront étudiés dans ce chapitre, les **Lists**, les **Sets** et les **Maps**, traitent les objets comme s'ils n'avaient pas de type spécifique. C'est à dire qu'ils les traitent comme s'ils étaient des **Objects**, la classe de base de toutes les classes en Java. Ceci est très intéressant d'un certain point de vue : un seul conteneur est nécessaire pour stocker tous les objets Java (excepté les types scalaires - ils peuvent toutefois être stockés dans les conteneurs sous forme de constantes en utilisant les classes Java d'encapsulation des types primitifs, ou sous forme de valeurs modifiables en les encapsulant dans des classes personnelles). C'est le deuxième point où les tableaux se distinguent des conteneurs génériques : lorsqu'un tableau est créé, il faut spécifier le type d'objets qu'il est destiné à stocker. Ce qui implique qu'on va bénéficier d'un contrôle de type lors de la phase compilation, nous empêchant de stocker des objets d'un mauvais type ou de se tromper sur le type de l'objet qu'on extrait. Bien sûr, Java empêchera tout envoi de message inapproprié à un objet, soit lors de la compilation soit lors de l'exécution du programme. Aucune des deux approches n'est donc plus risquée que l'autre, mais c'est tout de même mieux si c'est le compilateur qui signale l'erreur, plus rapide à l'exécution et il y a moins de chances que l'utilisateur final ne soit surpris par une exception.

Du fait de l'efficacité et du contrôle de type, il est toujours préférable d'utiliser un tableau si c'est possible. Cependant, les tableaux peuvent se révéler trop restrictifs pour résoudre certains problèmes. Après un examen des tableaux, le reste de ce chapitre sera consacré aux classes conteneurs proposées par Java.

Les tableaux sont des objets

Indépendamment du type de tableau qu'on utilise, un identifiant de tableau est en fait une référence sur un vrai objet créé dans le segment. C'est l'objet qui stocke les références sur les autres objets, et il peut être créé soit implicitement grâce à la syntaxe d'initialisation de tableau, soit explicitement avec une expression **new**. Une partie de l'objet tableau (en fait, la seule méthode ou champ auquel on peut accéder) est le membre en lecture seule **length** qui indique combien d'éléments peuvent être stockés dans l'objet. La syntaxe «`[]`» est le seul autre accès disponible pour les objets tableaux.

L'exemple suivant montre les différentes façons d'initialiser un tableau, et comment les références sur un tableau peuvent être assignées à différents objets tableau. Il montre aussi que les tableaux d'objets et les tableaux de scalaires sont quasi identiques dans leur utilisation. La seule différence est qu'un tableau d'objets stocke des références, alors qu'un tableau de scalaires stocke les valeurs directement.

```
/// c09:ArraySize.java  
  
// Initialisation & ré-assignation des tableaux.
```

```
class Weeble {} // Une petite créature mythique

public class ArraySize {
    public static void main(String[] args) {
        // Tableaux d'objets :

        Weeble[] a; // Référence Null.

        Weeble[] b = new Weeble[5]; // Références Null
        Weeble[] c = new Weeble[4];

        for(int i = 0; i < c.length; i++)
            c[i] = new Weeble();

        // Initialisation par agrégat :

        Weeble[] d = {
            new Weeble(), new Weeble(), new Weeble()
        };

        // Initialisation dynamique par agrégat :

        a = new Weeble[] {
            new Weeble(), new Weeble()
        };

        System.out.println("a.length=" + a.length);
        System.out.println("b.length = " + b.length);
        // Les références à l'intérieur du tableau sont
        // automatiquement initialisées à null :

        for(int i = 0; i < b.length; i++)
            System.out.println("b[" + i + "]= " + b[i]);

        System.out.println("c.length = " + c.length);
        System.out.println("d.length = " + d.length);

        a = d;

        System.out.println("a.length = " + a.length);

        // Tableaux de scalaires :

        int[] e; // Référence Null

        int[] f = new int[5];

        int[] g = new int[4];

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

```
        g[i] = i*i;
    int[] h = { 11, 47, 93 };
    // Erreur de compilation : variable e non initialisée :
    //!System.out.println("e.length=" + e.length);
    System.out.println("f.length = " + f.length);
    // Les scalaires dans le tableau sont
    // automatiquement initialisées à zéro :
    for(int i = 0; i < f.length; i++)
        System.out.println("f[" + i + "]= " + f[i]);
    System.out.println("g.length = " + g.length);
    System.out.println("h.length = " + h.length);
    e = h;
    System.out.println("e.length = " + e.length);
    e = new int[] { 1, 2 };
    System.out.println("e.length = " + e.length);
}
} ///:~
```

Voici la sortie du programme :

```
a.length = 2
b.length = 5
b[0]=null
b[1]=null
b[2]=null
b[3]=null
b[4]=null
c.length = 4
d.length = 3
a.length = 3
f.length = 5
f[0]=0
f[1]=0
f[2]=0
f[3]=0
```

```
f[4]=0
g.length = 4
h.length = 3
e.length = 3
e.length = 2
```

Le tableau **a** n'est initialement qu'une référence **null**, et le compilateur interdit d'utiliser cette référence tant qu'elle n'est pas correctement initialisée. Le tableau **b** est initialisé afin de pointer sur un tableau de références **Weeble**, même si aucun objet **Weeble** n'est réellement stocké dans le tableau. Cependant, on peut toujours s'enquérir de la taille du tableau, puisque **b** pointe sur un objet valide. Ceci montre un inconvénient des tableaux : on ne peut savoir combien d'éléments sont actuellement stockés *dans* le tableau, puisque **length** renvoie seulement le nombre d'éléments qu'on *peut* stocker dans le tableau, autrement dit la taille de l'objet tableau, et non le nombre d'éléments qu'il contient réellement. Cependant, quand un objet tableau est créé, ses références sont automatiquement initialisées à **null**, on peut donc facilement savoir si une cellule du tableau contient un objet ou pas en testant si son contenu est **null**. De même, un tableau de scalaires est automatiquement initialisé à zéro pour les types numériques, **(char)0** pour les caractères et **false** pour les **booleans**.

Le tableau **c** montre la création d'un objet tableau suivi par l'assignation d'un objet **Weeble** à chacune des cellules du tableau. Le tableau **d** illustre la syntaxe d'« initialisation par agrégat » qui permet de créer un objet tableau (implicitement sur le segment avec **new**, comme le tableau **c**) *et* de l'initialiser avec des objets **Weeble**, le tout dans une seule instruction.

L'initialisation de tableau suivante peut être qualifiée d'« initialisation dynamique par agrégat ». L'initialisation par agrégat utilisée par **d** doit être utilisée lors de la définition de **d**, mais avec la seconde syntaxe il est possible de créer et d'initialiser un objet tableau n'importe où. Par exemple, supposons que **hide()** soit une méthode qui accepte un tableau d'objets **Weeble** comme argument. On peut l'appeler via :

```
hide(d);
```

mais on peut aussi créer dynamiquement le tableau qu'on veut passer comme argument :

```
hide(new Weeble[] { new Weeble(), new Weeble() });
```

Cette nouvelle syntaxe est bien plus pratique pour certaines parties de code. L'expression :

```
a = d;
```

montre comment prendre une référence attachée à un tableau d'objets et l'assigner à un autre objet tableau, de la même manière qu'avec n'importe quel type de référence. Maintenant **a** et **d** pointent sur le même tableau d'objets dans le segment. La seconde partie de **ArraySize.java** montre que les tableaux de scalaires fonctionnent de la même manière que les tableaux d'objets *sauf que* les tableaux de scalaires stockent directement les valeurs des scalaires.

Conteneurs de scalaires

Les classes conteneurs ne peuvent stocker que des références sur des objets. Un tableau, par contre, peut stocker directement des scalaires aussi bien que des références sur des objets. Il *est* possible d'utiliser des classes d'« encapsulation » telles qu'**Integer**, **Double**, etc. pour stocker des valeurs scalaires dans un conteneur, mais les classes d'encapsulation pour les types primitifs se révèlent souvent lourdes à utiliser. De plus, il est bien plus efficace de créer et d'accéder à un tableau de scalaires qu'à un conteneur de scalaires encapsulés.

Bien sûr, si on utilise un type primitif et qu'on a besoin de la flexibilité d'un conteneur qui ajuste sa taille automatiquement, le tableau ne convient plus et il faut se rabattre sur un conteneur de scalaires encapsulés. On pourrait se dire qu'il serait bon d'avoir un type **ArrayList** spécialisé pour chacun des types de base, mais ils n'existent pas dans Java. Un mécanisme de patrons permettra sans doute un jour à Java de mieux gérer ce problème [\[45\]](#).

Renvoyer un tableau

Supposons qu'on veuille écrire une méthode qui ne renvoie pas une seule chose, mais tout un ensemble de choses. Ce n'est pas facile à réaliser dans des langages tels que C ou C++ puisqu'ils ne permettent pas de renvoyer un tableau, mais seulement un pointeur sur un tableau. Cela ouvre la porte à de nombreux problèmes du fait qu'il devient ardu de contrôler la durée de vie du tableau, ce qui mène très rapidement à des fuites de mémoire.

Java utilise une approche similaire, mais permet de « renvoyer un tableau ». Bien sûr, il s'agit en fait d'une référence sur un tableau, mais Java assume de manière transparente la responsabilité de ce tableau - il sera disponible tant qu'on en aura besoin, et le ramasse-miettes le nettoiera lorsqu'on en aura fini avec lui.

Voici un exemple retournant un tableau de **String** :

```
//: c09:IceCream.java

// Renvoyer un tableau depuis des méthodes.

public class IceCream {

    static String[] flav = {

        "Chocolate", "Strawberry",

        "Vanilla Fudge Swirl", "Mint Chip",

        "Mocha Almond Fudge", "Rum Raisin",

        "Praline Cream", "Mud Pie"

    };

    static String[] flavorSet(int n) {

        // Force l'argument à être positif & à l'intérieur des indices :

        n = Math.abs(n) % (flav.length + 1);

        String[] results = new String[n];

        boolean[] picked =

            new boolean[flav.length];
```

```

    for (int i = 0; i < n; i++) {
        int t;
        do
            t = (int)(Math.random() * flav.length);
        while (picked[t]);
        results[i] = flav[t];
        picked[t] = true;
    }
    return results;
}

public static void main(String[] args) {
    for(int i = 0; i < 20; i++) {
        System.out.println(
            "flavorSet(" + i + ") = ");
        String[] fl = flavorSet(flav.length);
        for(int j = 0; j < fl.length; j++)
            System.out.println("\t" + fl[j]);
    }
}
} ///:~

```

La méthode **flavorSet()** crée un tableau de **Strings** de taille **n** (déterminé par l'argument de la méthode) appelé **results**. Elle choisit alors au hasard des parfums dans le tableau **flav** et les place dans **results**, qu'elle renvoie quand elle en a terminé. Renvoyer un tableau s'apparente à renvoyer n'importe quel autre objet - ce n'est qu'une référence. Le fait que le tableau ait été créé dans **flavorSet()** n'est pas important, il aurait pu être créé n'importe où. Le ramasse-miettes s'occupe de nettoyer le tableau quand on en a fini avec lui, mais le tableau existera tant qu'on en aura besoin.

Notez en passant que quand **flavorSet()** choisit des parfums au hasard, elle s'assure que le parfum n'a pas déjà été choisi auparavant. Ceci est réalisé par une boucle **do** qui continue de tirer un parfum au sort jusqu'à ce qu'elle en trouve un qui ne soit pas dans le tableau **picked** (bien sûr, on aurait pu utiliser une comparaison sur **String** avec les éléments du tableau **results**, mais les comparaisons sur **String** ne sont pas efficaces). Une fois le parfum sélectionné, elle l'ajoute dans le tableau et trouve le parfum suivant (**i** est alors incrémenté).

main() affiche 20 ensembles de parfums, et on peut voir que **flavorSet()** choisit les parfums dans un ordre aléatoire à chaque fois. Il est plus facile de s'en rendre compte si on redirige la sortie dans un fichier. Et lorsque vous examinerez ce fichier, rappelez-vous que vous *voulez* juste la glace, vous n'en avez pas *besoin*.

La classe Arrays

java.util contient la classe **Arrays**, qui propose un ensemble de méthodes **static** réalisant des opérations utiles sur les tableaux. Elle dispose de quatre fonctions de base : **equals()**, qui compare deux tableaux ; **fill()**, pour remplir un tableau avec une valeur ; **sort()**, pour trier un tableau ; et **binarySearch()**, pour trouver un élément dans un tableau trié. Toutes ces méthodes sont surchargées pour tous les types scalaires et les **Objects**. De plus, il existe une méthode **asList()** qui transforme un tableau en un conteneur **List** - que nous rencontrerons plus tard dans ce chapitre.

Bien que pratique, la classe **Arrays** montre vite ses limites. Par exemple, il serait agréable de pouvoir facilement afficher les éléments d'un tableau sans avoir à coder une boucle **for** à chaque fois. Et comme nous allons le voir, la méthode **fill()** n'accepte qu'une seule valeur pour remplir le tableau, ce qui la rend inutile si on voulait - par exemple - remplir le tableau avec des nombres aléatoires.

Nous allons donc compléter la classe **Arrays** avec d'autres utilitaires, qui seront placés dans le **package com.bruceeckel.util**. Ces utilitaires permettront d'afficher un tableau de n'importe quel type, et de remplir un tableau avec des valeurs ou des objets créés par un objet appelé *générateur* qu'il est possible de définir.

Du fait qu'il faille écrire du code pour chaque type scalaire de base aussi bien que pour la classe **Object**, une grande majorité de ce code est dupliqué [46]. Ainsi, par exemple une interface « générateur » est requise pour chaque type parce que le type renvoyé par **next()** doit être différent dans chaque cas :

```
//: com:bruceeckel:util:Generator.java
package com.bruceeckel.util;

public interface Generator {

    Object next();

} ///:~


//: com:bruceeckel:util:BooleanGenerator.java
package com.bruceeckel.util;

public interface BooleanGenerator {

    boolean next();

} ///:~


//: com:bruceeckel:util:ByteGenerator.java
package com.bruceeckel.util;

public interface ByteGenerator {

    byte next();

} ///:~


//: com:bruceeckel:util:CharGenerator.java
package com.bruceeckel.util;
```



```
public interface CharGenerator {  
    char next();  
} ///:~  
  
//: com:bruceeckel:util:ShortGenerator.java  
package com.bruceeckel.util;  
public interface ShortGenerator {  
    short next();  
} ///:~  
  
//: com:bruceeckel:util:IntGenerator.java  
package com.bruceeckel.util;  
public interface IntGenerator {  
    int next();  
} ///:~  
  
//: com:bruceeckel:util:LongGenerator.java  
package com.bruceeckel.util;  
public interface LongGenerator {  
    long next();  
} ///:~  
  
//: com:bruceeckel:util:FloatGenerator.java  
package com.bruceeckel.util;  
public interface FloatGenerator {  
    float next();  
} ///:~  
  
//: com:bruceeckel:util:DoubleGenerator.java  
package com.bruceeckel.util;  
public interface DoubleGenerator {  
    double next();  
} ///:~
```

Arrays2 contient un ensemble de fonctions **print()**, surchargées pour chacun des types. Il est possible d'imprimer un tableau, d'afficher un message avant que le tableau ne soit affiché, ou de n'afficher qu'une certaine plage d'éléments du tableau. Le code de la méthode **print()** s'explique de lui-même :

```
///  
// com:bruceeckel:util:Arrays2.java  
// Un complément à java.util.Arrays, pour fournir  
// de nouvelles fonctionnalités utiles lorsqu'on  
// travaille avec des tableaux. Permet d'afficher  
// n'importe quel tableau, et de le remplir via un  
// objet « générateur » personnalisable.  
  
package com.bruceeckel.util;  
  
import java.util.*;  
  
public class Arrays2 {  
    private static void  
    start(int from, int to, int length) {  
        if(from != 0 || to != length)  
            System.out.print("[ "+ from + ":" + to + " ] ");  
        System.out.print("(");  
    }  
  
    private static void end() {  
        System.out.println(")");  
    }  
  
    public static void print(Object[] a) {  
        print(a, 0, a.length);  
    }  
  
    public static void  
    print(String msg, Object[] a) {  
        System.out.print(msg + " ");  
        print(a, 0, a.length);  
    }  
  
    public static void  
    print(Object[] a, int from, int to){  
        start(from, to, a.length);  
        for(int i = from; i < to; i++) {
```

```
        System.out.print(a[i]);

        if(i < to -1)

            System.out.print(", ");

    }

    end();

}

public static void print(boolean[] a) {

    print(a, 0, a.length);

}

public static void

print(String msg, boolean[] a) {

    System.out.print(msg + " ");

    print(a, 0, a.length);

}

public static void

print(boolean[] a, int from, int to) {

    start(from, to, a.length);

    for(int i = from; i < to; i++) {

        System.out.print(a[i]);

        if(i < to -1)

            System.out.print(", ");

    }

    end();

}

public static void print(byte[] a) {

    print(a, 0, a.length);

}

public static void

print(String msg, byte[] a) {

    System.out.print(msg + " ");

    print(a, 0, a.length);

}

public static void

print(byte[] a, int from, int to) {
```

```
        start(from, to, a.length);
        for(int i = from; i < to; i++) {
            System.out.print(a[i]);
            if(i < to -1)
                System.out.print(", ");
        }
        end();
    }

    public static void print(char[] a) {
        print(a, 0, a.length);
    }

    public static void
    print(String msg, char[] a) {
        System.out.print(msg + " ");
        print(a, 0, a.length);
    }

    public static void
    print(char[] a, int from, int to) {
        start(from, to, a.length);
        for(int i = from; i < to; i++) {
            System.out.print(a[i]);
            if(i < to -1)
                System.out.print(", ");
        }
        end();
    }

    public static void print(short[] a) {
        print(a, 0, a.length);
    }

    public static void
    print(String msg, short[] a) {
        System.out.print(msg + " ");
        print(a, 0, a.length);
    }
}
```

```
public static void
print(short[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to - 1)
            System.out.print(", ");
    }
    end();
}

public static void print(int[] a) {
    print(a, 0, a.length);
}

public static void
print(String msg, int[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}

public static void
print(int[] a, int from, int to) {
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to - 1)
            System.out.print(", ");
    }
    end();
}

public static void print(long[] a) {
    print(a, 0, a.length);
}

public static void
print(String msg, long[] a) {
    System.out.print(msg + " ");
```

```
        print(a, 0, a.length);
    }

    public static void
    print(long[] a, int from, int to) {
        start(from, to, a.length);
        for(int i = from; i < to; i++) {
            System.out.print(a[i]);
            if(i < to - 1)
                System.out.print(", ");
        }
        end();
    }

    public static void print(float[] a) {
        print(a, 0, a.length);
    }

    public static void
    print(String msg, float[] a) {
        System.out.print(msg + " ");
        print(a, 0, a.length);
    }

    public static void
    print(float[] a, int from, int to) {
        start(from, to, a.length);
        for(int i = from; i < to; i++) {
            System.out.print(a[i]);
            if(i < to - 1)
                System.out.print(", ");
        }
        end();
    }

    public static void print(double[] a) {
        print(a, 0, a.length);
    }

    public static void
```

```
print(String msg, double[] a) {
    System.out.print(msg + " ");
    print(a, 0, a.length);
}

public static void
print(double[] a, int from, int to){
    start(from, to, a.length);
    for(int i = from; i < to; i++) {
        System.out.print(a[i]);
        if(i < to - 1)
            System.out.print(", ");
    }
    end();
}

// Remplit un tableau en utilisant un générateur :
public static void
fill(Object[] a, Generator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(Object[] a, int from, int to,
    Generator gen){
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

public static void
fill(boolean[] a, BooleanGenerator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(boolean[] a, int from, int to,
    BooleanGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}
```

```
}

public static void
fill(byte[] a, ByteGenerator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(byte[] a, int from, int to,
    ByteGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

public static void
fill(char[] a, CharGenerator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(char[] a, int from, int to,
    CharGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

public static void
fill(short[] a, ShortGenerator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(short[] a, int from, int to,
    ShortGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

public static void
fill(int[] a, IntGenerator gen) {
    fill(a, 0, a.length, gen);
```



```
}

public static void
fill(int[] a, int from, int to,
    IntGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

public static void
fill(long[] a, LongGenerator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(long[] a, int from, int to,
    LongGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

public static void
fill(float[] a, FloatGenerator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(float[] a, int from, int to,
    FloatGenerator gen) {
    for(int i = from; i < to; i++)
        a[i] = gen.next();
}

public static void
fill(double[] a, DoubleGenerator gen) {
    fill(a, 0, a.length, gen);
}

public static void
fill(double[] a, int from, int to,
    DoubleGenerator gen){
```

```
        for(int i = from; i < to; i++)
            a[i] = gen.next();
    }

    private static Random r = new Random();

    public static class RandBooleanGenerator
    implements BooleanGenerator {
        public boolean next() {
            return r.nextBoolean();
        }
    }

    public static class RandByteGenerator
    implements ByteGenerator {
        public byte next() {
            return (byte)r.nextInt();
        }
    }

    static String ssource =
        "ABCDEFGHJKLMNOPQRSTUVWXYZ" +
        "abcdefghijklmnopqrstuvwxyz";

    static char[] src = ssource.toCharArray();

    public static class RandCharGenerator
    implements CharGenerator {
        public char next() {
            int pos = Math.abs(r.nextInt());
            return src[pos % src.length];
        }
    }

    public static class RandStringGenerator
    implements Generator {
        private int len;

        private RandCharGenerator cg =
            new RandCharGenerator();

        public RandStringGenerator(int length) {
            len = length;
        }
    }
}
```

```
    }

    public Object next() {
        char[] buf = new char[len];
        for(int i = 0; i < len; i++)
            buf[i] = cg.next();
        return new String(buf);
    }
}

public static class RandShortGenerator
implements ShortGenerator {
    public short next() {
        return (short)r.nextInt();
    }
}

public static class RandIntGenerator
implements IntGenerator {
    private int mod = 10000;

    public RandIntGenerator() {}

    public RandIntGenerator(int modulo) {
        mod = modulo;
    }

    public int next() {
        return r.nextInt() % mod;
    }
}

public static class RandLongGenerator
implements LongGenerator {
    public long next() { return r.nextLong(); }
}

public static class RandFloatGenerator
implements FloatGenerator {
    public float next() { return r.nextFloat(); }
}

public static class RandDoubleGenerator
```

```

implements DoubleGenerator {

    public double next() {return r.nextDouble();}

}

} ///:~

```

Pour remplir un tableau en utilisant un générateur, la méthode **fill()** accepte une référence sur une **interface** générateur, qui dispose d'une méthode **next()** produisant d'une façon ou d'une autre (selon l'implémentation de l'interface) un objet du bon type. La méthode **fill()** se contente d'appeler **next()** jusqu'à ce que la plage désirée du tableau soit remplie. Il est donc maintenant possible de créer un générateur en implémentant l'**interface** appropriée, et d'utiliser ce générateur avec **fill()**.

Les générateurs de données aléatoires sont utiles lors des tests, un ensemble de classes internes a donc été créé pour implémenter toutes les interfaces pour les types scalaires de base, de même qu'un générateur de **String** pour représenter des **Objects**. On peut noter au passage que **RandStringGenerator** utilise **RandCharGenerator** pour remplir un tableau de caractères, qui est ensuite transformé en **String**. La taille du tableau est déterminée par l'argument du constructeur.

Afin de générer des nombres qui ne soient pas trop grands, **RandIntGenerator** utilise un modulus par défaut de 10'000, mais un constructeur surchargé permet de choisir une valeur plus petite.

Voici un programme qui teste la bibliothèque et illustre la manière de l'utiliser :

```

//: c09:TestArrays2.java

// Teste et illustre les utilitaires d'Arrays2

import com.bruceeckel.util.*;

public class TestArrays2 {

    public static void main(String[] args) {

        int size = 6;

        // Ou récupère la taille depuis la ligne de commande :

        if(args.length != 0)

            size = Integer.parseInt(args[0]);

        boolean[] a1 = new boolean[size];

        byte[] a2 = new byte[size];

        char[] a3 = new char[size];

        short[] a4 = new short[size];

        int[] a5 = new int[size];

        long[] a6 = new long[size];

        float[] a7 = new float[size];
    }
}

```

```
double[] a8 = new double[size];

String[] a9 = new String[size];

Arrays2.fill(a1,

    new Arrays2.RandBooleanGenerator());

Arrays2.print(a1);

Arrays2.print("a1 = ", a1);

Arrays2.print(a1, size/3, size/3 + size/3);

Arrays2.fill(a2,

    new Arrays2.RandByteGenerator());

Arrays2.print(a2);

Arrays2.print("a2 = ", a2);

Arrays2.print(a2, size/3, size/3 + size/3);

Arrays2.fill(a3,

    new Arrays2.RandCharGenerator());

Arrays2.print(a3);

Arrays2.print("a3 = ", a3);

Arrays2.print(a3, size/3, size/3 + size/3);

Arrays2.fill(a4,

    new Arrays2.RandShortGenerator());

Arrays2.print(a4);

Arrays2.print("a4 = ", a4);

Arrays2.print(a4, size/3, size/3 + size/3);

Arrays2.fill(a5,

    new Arrays2.RandIntGenerator());

Arrays2.print(a5);

Arrays2.print("a5 = ", a5);

Arrays2.print(a5, size/3, size/3 + size/3);

Arrays2.fill(a6,

    new Arrays2.RandLongGenerator());

Arrays2.print(a6);

Arrays2.print("a6 = ", a6);

Arrays2.print(a6, size/3, size/3 + size/3);

Arrays2.fill(a7,

    new Arrays2.RandFloatGenerator());
```

```

    Arrays2.print(a7);

    Arrays2.print("a7 = ", a7);

    Arrays2.print(a7, size/3, size/3 + size/3);

    Arrays2.fill(a8,

        new Arrays2.RandDoubleGenerator());

    Arrays2.print(a8);

    Arrays2.print("a8 = ", a8);

    Arrays2.print(a8, size/3, size/3 + size/3);

    Arrays2.fill(a9,

        new Arrays2.RandStringGenerator(7));

    Arrays2.print(a9);

    Arrays2.print("a9 = ", a9);

    Arrays2.print(a9, size/3, size/3 + size/3);

}

} ///:~

```

Le paramètre **size** a une valeur par défaut, mais il est possible de le fixer depuis la ligne de commande.

Remplir un tableau

La bibliothèque standard Java **Arrays** propose aussi une méthode **fill()**, mais celle-ci est relativement triviale : elle ne fait que dupliquer une certaine valeur dans chaque cellule, ou dans le cas d'objets, copier la même référence dans chaque cellule. En utilisant **Arrays2.print()**, les méthodes **Arrays.fill()** peuvent être facilement illustrées :

```

///: c09:FillingArrays.java

// Utilisation de Arrays.fill()

import com.bruceeckel.util.*;

import java.util.*;

public class FillingArrays {

    public static void main(String[] args) {

        int size = 6;

        // Ou récupère la taille depuis la ligne de commande :

        if(args.length != 0)

            size = Integer.parseInt(args[0]);

        boolean[] a1 = new boolean[size];
    }
}

```

```
byte[] a2 = new byte[size];
char[] a3 = new char[size];
short[] a4 = new short[size];
int[] a5 = new int[size];
long[] a6 = new long[size];
float[] a7 = new float[size];
double[] a8 = new double[size];
String[] a9 = new String[size];
Arrays.fill(a1, true);
Arrays2.print("a1 = ", a1);
Arrays.fill(a2, (byte)11);
Arrays2.print("a2 = ", a2);
Arrays.fill(a3, 'x');
Arrays2.print("a3 = ", a3);
Arrays.fill(a4, (short)17);
Arrays2.print("a4 = ", a4);
Arrays.fill(a5, 19);
Arrays2.print("a5 = ", a5);
Arrays.fill(a6, 23);
Arrays2.print("a6 = ", a6);
Arrays.fill(a7, 29);
Arrays2.print("a7 = ", a7);
Arrays.fill(a8, 47);
Arrays2.print("a8 = ", a8);
Arrays.fill(a9, "Hello");
Arrays2.print("a9 = ", a9);
// Manipulation de plages d'index :
Arrays.fill(a9, 3, 5, "World");
Arrays2.print("a9 = ", a9);
}
} ///:~
```

On peut soit remplir un tableau complètement, soit - comme le montrent les deux dernières instructions - une certaine plage d'indices. Mais comme il n'est possible de ne fournir qu'une seule valeur pour le remplissage dans **Arrays.fill()**, les méthodes **Arrays2.fill()** sont bien plus intéressantes.

Copier un tableau

La bibliothèque standard Java propose une méthode **static**, **System.arraycopy()**, qui réalise des copies de tableau bien plus rapidement qu'une boucle **for**. **System.arraycopy()** est surchargée afin de gérer tous les types. Voici un exemple qui manipule des tableaux d'**int** :

```
//: c09:CopyingArrays.java

// Utilisation de System.arraycopy()

import com.bruceeckel.util.*;

import java.util.*;

public class CopyingArrays {

    public static void main(String[] args) {

        int[] i = new int[25];

        int[] j = new int[25];

        Arrays.fill(i, 47);

        Arrays.fill(j, 99);

        Arrays2.print("i = ", i);

        Arrays2.print("j = ", j);

        System.arraycopy(i, 0, j, 0, i.length);

        Arrays2.print("j = ", j);

        int[] k = new int[10];

        Arrays.fill(k, 103);

        System.arraycopy(i, 0, k, 0, k.length);

        Arrays2.print("k = ", k);

        Arrays.fill(k, 103);

        System.arraycopy(k, 0, i, 0, k.length);

        Arrays2.print("i = ", i);

        // Objects :

        Integer[] u = new Integer[10];

        Integer[] v = new Integer[5];

        Arrays.fill(u, new Integer(47));

        Arrays.fill(v, new Integer(99));

        Arrays2.print("u = ", u);

        Arrays2.print("v = ", v);
```



```

        System.arraycopy(v, 0,
            u, u.length/2, v.length);
        Arrays2.print("u = ", u);
    }
} ///:~

```

arraycopy() accepte comme arguments le tableau source, le déplacement dans le tableau source à partir duquel démarrer la copie, le tableau destination, le déplacement dans le tableau destination à partir duquel démarrer la copie, et le nombre d'éléments à copier. Bien entendu, toute violation des frontières du tableau générera une exception.

L'exemple montre bien qu'on peut copier des tableaux de scalaires comme des tableaux d'objets. Cependant, dans le cas de la copie de tableaux d'objets, seules les références sont copiées - il n'y a pas duplication des objets eux-mêmes. C'est ce qu'on appelle une *copie superficielle* (voir l'Annexe A).

Comparer des tableaux

Arrays fournit la méthode surchargée **equals()** pour comparer des tableaux entiers. Encore une fois, ces méthodes sont surchargées pour chacun des types de base, ainsi que pour les **Objects**. Pour être égaux, les tableaux doivent avoir la même taille et chaque élément doit être équivalent (au sens de la méthode **equals()**) à l'élément correspondant dans l'autre tableau (pour les types scalaires, la méthode **equals()** de la classe d'encapsulation du type concerné est utilisé ; par exemple, **Integer.equals()** est utilisé pour les **int**). Voici un exemple :

```

///: c09:ComparingArrays.java
// Utilisation de Arrays.equals()

import java.util.*;

public class ComparingArrays {
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = new int[10];

        Arrays.fill(a1, 47);
        Arrays.fill(a2, 47);

        System.out.println(Arrays.equals(a1, a2));

        a2[3] = 11;

        System.out.println(Arrays.equals(a1, a2));

        String[] s1 = new String[5];
        Arrays.fill(s1, "Hi");

        String[] s2 = {"Hi", "Hi", "Hi", "Hi", "Hi"};
    }
}

```

```

        System.out.println(Arrays.equals(s1, s2));
    }
} ///:~

```

Au début du programme, **a1** et **a2** sont identiques, donc le résultat est « true » ; puis l'un des éléments est changé donc la deuxième ligne affichée est « false ». Dans le dernier cas, tous les éléments des **s1** pointent sur le même objet, alors que **s2** contient cinq objets différents. Cependant, l'égalité de tableaux est basée sur le contenu (via **Object.equals()**) et donc le résultat est « true ».

Comparaison d'éléments de tableau

L'une des fonctionnalités manquantes dans les bibliothèques Java 1.0 et 1.1 sont les opérations algorithmiques - y compris les simples tris. Ceci était relativement frustrant pour quiconque s'attendait à une bibliothèque standard conséquente. Heureusement, Java 2 a corrigé cette situation, au moins pour le problème du tri.

Le problème posé par l'écriture d'une méthode de tri générique est que le tri doit réaliser des comparaisons basées sur le type réel de l'objet. Bien sûr, l'une des approches consiste à écrire une méthode de tri différente pour chaque type, mais cela va à l'encontre du principe de réutilisabilité du code pour les nouveaux types.

L'un des buts principaux de la conception est de « séparer les choses qui changent de celles qui ne bougent pas » ; ici, le code qui reste le même est l'algorithme général de tri, alors que la manière de comparer les objets entre eux est ce qui change d'un cas d'utilisation à l'autre. Donc au lieu de coder en dur le code de comparaison dans différentes procédures de tri, on utilise ici la technique des *callbacks*. Avec un callback, la partie du code qui varie d'un cas à l'autre est encapsulé dans sa propre classe, et la partie du code qui ne change pas appellera ce code pour réaliser les comparaisons. De cette manière, il est possible de créer différents objets pour exprimer différentes sortes de comparaisons et de les passer au même code de tri.

Dans Java 2, il existe deux manières de fournir des fonctionnalités de comparaison. La *méthode naturelle de comparaison* constitue la première, elle est annoncée dans une classe en implémentant l'interface **java.lang.Comparable**. C'est une interface très simple ne disposant que d'une seule méthode, **compareTo()**. Cette méthode accepte un autre **Object** comme argument, et renvoie une valeur négative si l'argument est plus grand que l'objet courant, zéro si ils sont égaux, ou une valeur positive si l'argument est plus petit que l'objet courant.

Voici une classe qui implémente **Comparable** et illustre la comparaison en utilisant la méthode **Arrays.sort()** de la bibliothèque standard Java :

```

///: c09:CompType.java

// Implémenter Comparable dans une classe.

import com.bruceeckel.util.*;

import java.util.*;

public class CompType implements Comparable {

    int i;

    int j;

```

```

public CompType(int n1, int n2) {
    i = n1;
    j = n2;
}

public String toString() {
    return "[i = " + i + ", j = " + j + "]";
}

public int compareTo(Object rv) {
    int rvi = ((CompType)rv).i;
    return (i < rvi ? -1 : (i == rvi ? 0 : 1));
}

private static Random r = new Random();

private static int randInt() {
    return Math.abs(r.nextInt()) % 100;
}

public static Generator generator() {
    return new Generator() {
        public Object next() {
            return new CompType(randInt(),randInt());
        }
    };
}

public static void main(String[] args) {
    CompType[] a = new CompType[10];
    Arrays2.fill(a, generator());
    Arrays2.print("before sorting, a = ", a);
    Arrays.sort(a);
    Arrays2.print("after sorting, a = ", a);
}

} ///:~

```

Lorsque la fonction de comparaison est définie, il vous incombe de décider du sens à donner à la comparaison entre deux objets. Ici, seules les valeurs **i** sont utilisées dans la comparaison, les valeurs **j** sont ignorées.

La méthode **static randInt()** produit des valeurs positives entre zéro et 100, et la méthode **generator()** produit

un objet implémentant l'interface **Generator**, en créant une classe interne anonyme (cf. Chapitre 8). Celui-ci génère des objets **CompType** en les initialisant avec des valeurs aléatoires. Dans **main()**, le générateur est utilisé pour remplir un tableau de **CompType**, qui est alors trié. Si **Comparable** n'avait pas été implémentée, une erreur de compilation aurait été générée lors d'un appel à **sort()**.

Dans le cas où une classe n'implémente pas **Comparable**, ou qu'elle l'implémente d'une manière qui ne vous satisfait pas (c'est à dire que vous souhaitez une autre fonction de comparaison pour ce type), il faut utiliser une autre approche pour comparer des objets. Cette approche nécessite de créer une classe séparée qui implémente l'interface **Comparator**, comportant les deux méthodes **compare()** et **equals()**. Cependant, sauf cas particuliers (pour des raisons de performance notamment), il n'est pas nécessaire d'implémenter **equals()** car chaque classe dérive implicitement de **Object**, qui fournit déjà cette méthode. On peut donc se contenter de la méthode **Object.equals()** pour satisfaire au contrat imposé par l'interface.

La classe **Collections** (que nous étudierons plus en détails par la suite) dispose d'un **Comparator** qui inverse l'ordre de tri. Ceci peut facilement être appliqué à **CompType** :

```

//: c09:Reverse.java

// Le Comparator Collections.reverseOrder().

import com.bruceeckel.util.*;

import java.util.*;

public class Reverse {

    public static void main(String[] args) {

        CompType[] a = new CompType[10];

        Arrays2.fill(a, CompType.generator());

        Arrays2.print("before sorting, a = ", a);

        Arrays.sort(a, Collections.reverseOrder());

        Arrays2.print("after sorting, a = ", a);

    }

} ///:~

```

L'appel à **Collections.reverseOrder()** produit une référence sur le **Comparator**.

Voici un deuxième exemple dans lequel un **Comparator** compare des objets **CompType** en se basant cette fois sur la valeur de leur **j** plutôt que sur celle de **i** :

```

//: c09:ComparatorTest.java

// Implémenter un Comparator pour une classe.

import com.bruceeckel.util.*;

import java.util.*;

```

```

class CompTypeComparator implements Comparator {
    public int compare(Object o1, Object o2) {
        int j1 = ((CompType)o1).j;
        int j2 = ((CompType)o2).j;
        return (j1 < j2 ? -1 : (j1 == j2 ? 0 : 1));
    }
}

public class ComparatorTest {
    public static void main(String[] args) {
        CompType[] a = new CompType[10];
        Arrays2.fill(a, CompType.generator());
        Arrays2.print("before sorting, a = ", a);
        Arrays.sort(a, new CompTypeComparator());
        Arrays2.print("after sorting, a = ", a);
    }
} ///:~

```

La méthode **compare()** doit renvoyer un entier négatif, zéro ou un entier positif selon que le premier argument est respectivement plus petit, égal ou plus grand que le second.

Trier un tableau

Avec les méthodes de tri intégrées, il est maintenant possible de trier n'importe quel tableau de scalaires ou d'objets implémentant **Comparable** ou disposant d'une classe **Comparator** associée. Ceci comble un énorme trou dans les bibliothèques de Java - croyez-le ou non, Java 1.0 ou 1.1 ne fournissait aucun moyen de trier des **Strings** ! Voici un exemple qui génère des objets **String** aléatoirement et les trie :

```

///: c09:StringSorting.java
// Trier un tableau de Strings.

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

public class StringSorting {
    public static void main(String[] args) {
        String[] sa = new String[30];
        Arrays2.fill(sa,

```

```

        new Arrays2.RandStringGenerator(5));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa);
    Arrays2.print("After sorting: ", sa);
}
} ///:~

```

Il est bon de noter que le tri effectué sur les **Strings** est *lexicographique*, c'est à dire que les mots commençant par des majuscules apparaissent avant ceux débutant par une minuscule (typiquement les annuaires sont triés de cette façon). Il est toutefois possible de redéfinir ce comportement et d'ignorer la casse en définissant une classe **Comparator**. Cette classe sera placée dans le package « util » à des fins de réutilisation :

```

///: com:bruceeckel:util:AlphabeticComparator.java
// Garder les lettres majuscules et minuscules ensemble.
package com.bruceeckel.util;
import java.util.*;

public class AlphabeticComparator
implements Comparator{
    public int compare(Object o1, Object o2) {
        String s1 = (String)o1;
        String s2 = (String)o2;
        return s1.toLowerCase().compareTo(
            s2.toLowerCase());
    }
} ///:~

```

Chaque **String** est convertie en minuscules avant la comparaison. La méthode **compareTo()** de **String** fournit ensuite le comparateur désiré. Voici un exemple d'utilisation d'**AlphabeticComparator** :

```

///: c09:AlphabeticSorting.java
// Garder les lettres majuscules et minuscules ensemble.
import com.bruceeckel.util.*;
import java.util.*;

public class AlphabeticSorting {

```

```

public static void main(String[] args) {
    String[] sa = new String[30];
    Arrays2.fill(sa,
        new Arrays2.RandStringGenerator(5));
    Arrays2.print("Before sorting: ", sa);
    Arrays.sort(sa, new AlphabeticComparator());
    Arrays2.print("After sorting: ", sa);
}
} ///:~

```

L'algorithme de tri utilisé dans la bibliothèque standard de Java est conçu pour être optimal suivant le type d'objets triés : un Quicksort pour les scalaires, et un tri-fusion stable pour les objets. Vous ne devriez donc pas avoir à vous soucier des performances à moins qu'un outil de profilage ne vous démontre explicitement que le goulot d'étranglement de votre programme soit le processus de tri.

Effectuer une recherche sur un tableau trié

Une fois un tableau trié, il est possible d'effectuer une recherche rapide sur un item en utilisant **Arrays.binarySearch()**. Il est toutefois très important de ne pas utiliser **binarySearch()** sur un tableau non trié ; le résultat en serait imprévisible. L'exemple suivant utilise un **RandIntGenerator** pour remplir un tableau et produire des valeurs à chercher dans ce tableau :

```

///: c09:ArraySearching.java
// Utilisation de Arrays.binarySearch().

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

public class ArraySearching {
    public static void main(String[] args) {
        int[] a = new int[100];
        Arrays2.RandIntGenerator gen =
            new Arrays2.RandIntGenerator(1000);
        Arrays2.fill(a, gen);
        Arrays.sort(a);
        Arrays2.print("Sorted array: ", a);
        while(true) {
            int r = gen.next();
            int location = Arrays.binarySearch(a, r);

```

```

        if(location >= 0) {
            System.out.println("Location of " + r +
                " is " + location + ", a[" +
                location + "] = " + a[location]);
            break; // Sortie de la boucle while
        }
    }
}
} ///:~

```

Dans la boucle **while**, des valeurs aléatoires sont générées jusqu'à ce qu'une d'entre elles soit trouvée dans le tableau. **Arrays.binarySearch()** renvoie une valeur supérieure ou égale à zéro si l'item recherché est trouvé. Dans le cas contraire, elle renvoie une valeur négative représentant l'endroit où insérer l'élément si on désirait maintenir le tableau trié à la main. La valeur retournée est :

```

-(point d'insertion) - 1

```

Le point d'insertion est l'index du premier élément plus grand que la clef, ou **a.size()** si tous les éléments du tableau sont plus petits que la clef spécifiée.

Si le tableau contient des éléments dupliqués, aucune garantie n'est apportée quant à celui qui sera trouvé. L'algorithme n'est donc pas conçu pour les tableaux comportant des doublons, bien qu'il les tolère. Dans le cas où on a besoin d'une liste triée d'éléments sans doublons, mieux vaut se tourner vers un **TreeSet** (qui sera introduit plus loin dans ce chapitre) qui gère tous ces détails automatiquement, plutôt que de maintenir un tableau à la main (à moins que des questions de performance ne se greffent là-dessus).

Il faut fournir à **binarySearch()** le même objet **Comparator** que celui utilisé pour trier le tableau d'objets (les tableaux de scalaires n'autorisent pas les tris avec des **Comparator**), afin qu'elle utilise la version redéfinie de la fonction de comparaison. Ainsi, le programme **AlphabeticSorting.java** peut être modifié pour effectuer une recherche :

```

///: c09:AlphabeticSearch.java
// Rechercher avec un Comparator.

import com.bruceeckel.util.*;

import java.util.*;

public class AlphabeticSearch {
    public static void main(String[] args) {
        String[] sa = new String[30];
        Arrays2.fill(sa,
            new Arrays2.RandStringGenerator(5));
    }
}

```



```

        AlphabeticComparator comp =
            new AlphabeticComparator();
        Arrays.sort(sa, comp);

        int index =
            Arrays.binarySearch(sa, sa[10], comp);

        System.out.println("Index = " + index);
    }
} ///:~

```

binarySearch() accepte le **Comparator** en troisième argument. Dans l'exemple précédent, le succès de la recherche est garanti puisque l'item recherché est tiré du tableau lui-même.

Résumé sur les tableaux

Pour résumer ce qu'on a vu jusqu'à présent, un tableau se révèle la manière la plus simple et la plus efficace pour stocker un groupe d'objets, et le seul choix possible dans le cas où on veut stocker un ensemble de scalaires. Dans le reste de ce chapitre nous allons étudier le cas plus général dans lequel on ne sait pas au moment de l'écriture du programme combien d'objets seront requis, ainsi que des moyens plus sophistiqués de stocker les objets. Java propose en effet des *classes conteneurs* qui adressent différents problèmes. Les types de base en sont les **Lists**, les **Sets** et les **Maps**. Un nombre surprenant de problèmes peuvent être facilement résolus grâce à ces outils.

Entre autres caractéristiques - les **Sets**, par exemple, ne stockent qu'un objet de chaque valeur, les **Maps** sont des *tableaux associatifs* qui permettent d'associer n'importe quel objet avec n'importe quel autre objet - les classes conteneurs de Java se redimensionnent automatiquement. A l'inverse des tableaux, ils peuvent donc stocker un nombre quelconque d'objets et on n'a pas besoin de se soucier de leur taille lors de l'écriture du programme.

Introduction sur les conteneurs

Les classes conteneurs sont à mon sens l'un des outils les plus puissants disponibles parce qu'ils augmentent de façon significative la productivité du développement. Les conteneurs de Java 2 résultent d'une reconception approfondie [\[47\]](#) des implémentations relativement pauvres disponibles dans Java 1.0 et 1.1. Cette reconception a permis d'unifier et de rationaliser certains fonctionnements. Elle a aussi comblé certains manques de la bibliothèque des conteneurs tels que les listes chaînées, les files (queues) et les files doubles (queues à double entrée).

La conception d'une bibliothèque de conteneurs est difficile (de même que tous les problèmes de conception des bibliothèques). En C++, les classes conteneurs couvrent les bases grâce à de nombreuses classes différentes. C'est mieux que ce qui était disponible avant (ie, rien), mais le résultat ne se transpose pas facilement dans Java. J'ai aussi rencontré l'approche opposée, où la bibliothèque de conteneurs consistait en une seule classe qui fonctionnait à la fois comme une séquence linéaire et un tableau associatif. La bibliothèque de conteneurs de Java 2 essaie de trouver un juste milieu : les fonctionnalités auxquelles on peut s'attendre de la part d'une bibliothèque de conteneurs mûre, mais plus facile à appréhender que les classes conteneurs du C++ ou d'autres bibliothèques de conteneurs similaires. Le résultat peut paraître étrange dans certains cas. Mais contrairement à certaines décisions prises dans la conception des premières bibliothèques Java, ces bizarreries ne sont pas des

accidents de conception, mais des compromis minutieusement examinés sur la complexité. Il vous faudra peut-être un petit moment avant d'être à l'aise avec certains aspects de la bibliothèque, mais je pense que vous adopterez quand même très rapidement ces nouveaux outils.

Le but de la bibliothèque de conteneurs de Java 2 est de « stocker des objets » et le divise en deux concepts bien distincts :

1. **Collection** : un groupe d'éléments individuels, souvent associé à une règle définissant leur comportement. Une **List** doit garder les éléments dans un ordre précis, et un **Set** ne peut contenir de doublons (les *sacs* [NdT : *bag* en anglais], qui ne sont pas implémentés dans la bibliothèque de conteneurs de Java - les **Lists** fournissant des fonctionnalités équivalentes - ne possèdent pas une telle règle).
2. **Map** : un ensemble de paires clef - valeur. A première vue, on pourrait penser qu'il ne s'agit que d'une **Collection** de paires, mais lorsqu'on essaie de l'implémenter de cette manière, le design devient très rapidement bancal et lourd à mettre en oeuvre ; il est donc plus simple d'en faire un concept séparé. D'un autre côté, il est bien pratique d'examiner certaines portions d'une **Map** en créant une **Collection** représentant cette portion. Une **Map** peut donc renvoyer un **Set** de ses clefs, une **Collection** de ses valeurs, ou un **Set** de ses paires. Les **Maps**, comme les tableaux, peuvent facilement être étendus dans de multiples dimensions sans ajouter de nouveaux concepts : il suffit de créer une **Map** dont les valeurs sont des **Maps** (les valeurs de ces **Maps** pouvant *elles-mêmes* être des **Maps**, etc.).

Nous allons d'abord examiner les fonctionnalités générales des conteneurs, puis aller dans les spécificités des conteneurs et enfin nous apprendrons pourquoi certains conteneurs sont déclinés en plusieurs versions, et comment choisir entre eux.

Imprimer les conteneurs

A l'inverse des tableaux, les conteneurs s'affichent correctement sans aide. Voici un exemple qui introduit en même temps les conteneurs de base :

```
//: c09:PrintingContainers.java
// Les conteneurs savent comment s'afficher.

import java.util.*;

public class PrintingContainers {

    static Collection fill(Collection c) {

        c.add("dog");

        c.add("dog");

        c.add("cat");

        return c;

    }

    static Map fill(Map m) {

        m.put("dog", "Bosco");

        m.put("dog", "Spot");
```

```

        m.put("cat", "Rags");

        return m;
    }

    public static void main(String[] args) {

        System.out.println(fill(new ArrayList()));

        System.out.println(fill(new HashSet()));

        System.out.println(fill(new HashMap()));

    }

} ///:~

```

Comme mentionné précédemment, il existe deux catégories de base dans la bibliothèque de conteneurs Java. La distinction est basée sur le nombre d'items stockés dans chaque cellule du conteneur. La catégorie **Collection** ne stocke qu'un item dans chaque emplacement (le nom est un peu trompeur puisque les bibliothèques des conteneurs sont souvent appelées des « collections »). Elle inclut la **List**, qui stocke un groupe d'items dans un ordre spécifique, et le **Set**, qui autorise l'addition d'une seule instance pour chaque item. Une **ArrayList** est un type de **List**, et **HashSet** est un type de **Set**. La méthode **add()** permet d'ajouter des éléments dans une **Collection**.

Une **Map** contient des paires clef - valeur, un peu à la manière d'une mini base de données. Le programme précédent utilise un type de **Map**, le **HashMap**. Si on dispose d'une **Map** qui associe les états des USA avec leur capitale et qu'on souhaite connaître la capitale de l'Ohio, il suffit de la rechercher - comme si on indexait un tableau (les **Maps** sont aussi appelés des *tableaux associatifs*). La méthode **put()**, qui accepte deux arguments - la clef et la valeur -, permet de stocker des éléments dans une **Map**. L'exemple précédent se contente d'ajouter des éléments mais ne les récupère pas une fois stockés. Ceci sera illustré plus tard.

Les méthodes surchargées **fill()** remplissent respectivement des **Collections** et des **Maps**. En examinant la sortie produite par le programme, on peut voir que le comportement par défaut pour l'affichage (fourni par les méthodes **toString()** des différents conteneurs) produit un résultat relativement clair, il n'est donc pas nécessaire d'ajouter du code pour imprimer les conteneurs comme nous avons du le faire avec les tableaux :

```

[dog, dog, cat]

[cat, dog]

{cat=Rags, dog=Spot}

```

Une **Collection** est imprimée entre crochets, chaque élément étant séparé par une virgule. Une **Map** est entourée par des accolades, chaque clef étant associée à sa valeur avec un signe égal (les clefs à gauche, les valeurs à droite).

Le comportement de base des différents conteneurs est évident dans cet exemple. La **List** stocke les objets dans l'ordre exact où ils ont été ajoutés, sans aucun réarrangement ni édition. Le **Set**, lui, n'accepte qu'une seule instance d'un objet et utilise une méthode interne de tri (en général, un **Set** sert à savoir si un élément est un membre d'un **Set** ou non, et non l'ordre dans lequel il apparaît dans ce **Set** - pour cela il faut utiliser une **List**). La **Map** elle aussi n'accepte qu'une seule instance d'un objet pour la clef, possède elle aussi sa propre organisation interne et ne tient pas compte de l'ordre dans lequel les éléments ont été insérés.

Remplir les conteneurs

Bien que le problème d'impression des conteneurs soit géré pour nous, le remplissage des conteneurs souffre des mêmes limitations que **java.util.Arrays**. De même que pour les **Arrays**, il existe une classe compagnon appelée **Collections** contenant des méthodes **static** dont l'une s'appelle **fill()**. Cette méthode **fill()** ne fait que dupliquer une unique référence sur un objet dans le conteneur, et ne fonctionne que sur les objets **List**, pas sur les **Sets** ni les **Maps** :

```

//: c09:FillingLists.java
// La méthode Collections.fill().

import java.util.*;

public class FillingLists {
    public static void main(String[] args) {
        List list = new ArrayList();
        for(int i = 0; i < 10; i++)
            list.add("");
        Collections.fill(list, "Hello");
        System.out.println(list);
    }
} ///:~

```

Cette méthode est encore moins intéressante parce qu'elle ne fait que remplacer les éléments déjà présents dans la **List**, sans ajouter aucun élément.

Pour être capable de créer des exemples intéressants, voici une bibliothèque complémentaire **Collections2** (appartenant par commodité à **com.bruceeckel.util**) disposant d'une méthode **fill()** utilisant un générateur pour ajouter des éléments, et permettant de spécifier le nombre d'éléments qu'on souhaite ajouter. L'**interface Generator** définie précédemment fonctionne pour les **Collections**, mais les **Maps** requièrent leur propre **interface** générateur puisqu'un appel à **next()** doit produire une paire d'objets (une clef et une valeur). Voici tout d'abord la classe **Pair** :

```

//: com:bruceeckel:util:Pair.java

package com.bruceeckel.util;

public class Pair {
    public Object key, value;

    Pair(Object k, Object v) {
        key = k;
        value = v;
    }
}

```

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

Ensuite, l'**interface** générateur qui produit un objet **Pair** :

```
///  
com:bruceeckel:util:MapGenerator.java  
  
package com.bruceeckel.util;  
  
public interface MapGenerator {  
    Pair next();  
}  
///:~
```

Avec ces deux objets, un ensemble d'utilitaires pour travailler avec les classes conteneurs peuvent être développés :

```
///  
com:bruceeckel:util:Collections2.java  
  
// Remplir n'importe quel type de conteneur en  
// utilisant un objet générateur.  
  
package com.bruceeckel.util;  
  
import java.util.*;  
  
public class Collections2 {  
    // Remplit une Collection en utilisant un générateur :  
  
    public static void  
    fill(Collection c, Generator gen, int count) {  
        for(int i = 0; i < count; i++)  
            c.add(gen.next());  
    }  
  
    public static void  
    fill(Map m, MapGenerator gen, int count) {  
        for(int i = 0; i < count; i++) {  
            Pair p = gen.next();  
            m.put(p.key, p.value);  
        }  
    }  
  
    public static class RandStringPairGenerator  
        implements MapGenerator {
```

```
private Arrays2.RandStringGenerator gen;

public RandStringPairGenerator(int len) {
    gen = new Arrays2.RandStringGenerator(len);
}

public Pair next() {
    return new Pair(gen.next(), gen.next());
}
}

// Objet par défaut afin de ne pas avoir
// à en créer un de notre cru :
public static RandStringPairGenerator rsp =
    new RandStringPairGenerator(10);

public static class StringPairGenerator
implements MapGenerator {
    private int index = -1;
    private String[][] d;

    public StringPairGenerator(String[][] data) {
        d = data;
    }

    public Pair next() {
        // Force l'index dans la plage de valeurs :
        index = (index + 1) % d.length;

        return new Pair(d[index][0], d[index][1]);
    }

    public StringPairGenerator reset() {
        index = -1;

        return this;
    }
}

// Utilisation d'un ensemble de données prédéfinies :
public static StringPairGenerator geography =
    new StringPairGenerator(
        CountryCapitals.pairs);

// Produit une séquence à partir d'un tableau 2D :
```

```

public static class StringGenerator
implements Generator {
    private String[][] d;
    private int position;
    private int index = -1;

    public
    StringGenerator(String[][] data, int pos) {
        d = data;
        position = pos;
    }

    public Object next() {
        // Force l'index dans la plage de valeurs :
        index = (index + 1) % d.length;
        return d[index][position];
    }

    public StringGenerator reset() {
        index = -1;
        return this;
    }
}

// Utilisation d'un ensemble de données prédéfinies :
public static StringGenerator countries =
    new StringGenerator(CountryCapitals.pairs,0);
public static StringGenerator capitals =
    new StringGenerator(CountryCapitals.pairs,1);
} ///:~

```

Les deux versions de **fill()** acceptent un argument qui détermine le nombre d'éléments à ajouter dans le conteneur. De plus, il existe deux générateurs pour le tableau associatif : **RandStringPairGenerator**, qui génère un nombre de paires de **Strings** aléatoires dont la taille est déterminée par l'argument du constructeur ; et **StringPairGenerator**, qui produit des paires de **Strings** à partir d'un tableau bidimensionnel de **Strings**. **StringGenerator** accepte lui aussi un tableau bidimensionnel de **Strings** mais génère des éléments individuels plutôt que des **Pairs**. Les objets **static rsp**, **geography**, **countries** et **capitals** fournissent des générateurs intégrés, les trois derniers utilisant les pays du monde et leur capitale. Notez que si on essaie de créer plus de paires que le nombre disponible dans le tableau, les générateurs retourneront au début du tableau, et dans le cas d'une **Map**, les doublons seront simplement ignorés.

Voici l'ensemble de données prédéfinies, qui consiste en noms de pays avec leur capitale. Il est affiché avec une

petite fonte afin de réduire l'espace occupé :

```

//: com:bruceeckel:util:CountryCapitals.java

package com.bruceeckel.util;

public class CountryCapitals {

    public static final String[][] pairs = {

        // Afrique

        {"ALGERIA", "Algiers"}, {"ANGOLA", "Luanda"},

        {"BENIN", "Porto-Novo"}, {"BOTSWANA", "Gaberone"},

        {"BURKINA FASO", "Ouagadougou"}, {"BURUNDI", "Bujumbura"},

        {"CAMEROON", "Yaounde"}, {"CAPE VERDE", "Praia"},

        {"CENTRAL AFRICAN REPUBLIC", "Bangui"},

        {"CHAD", "N'djamena"}, {"COMOROS", "Moroni"},

        {"CONGO", "Brazzaville"}, {"DJIBOUTI", "Djibouti"},

        {"EGYPT", "Cairo"}, {"EQUATORIAL GUINEA", "Malabo"},

        {"ERITREA", "Asmara"}, {"ETHIOPIA", "Addis Ababa"},

        {"GABON", "Libreville"}, {"THE GAMBIA", "Banjul"},

        {"GHANA", "Accra"}, {"GUINEA", "Conakry"},

        {"GUINEA", "-"}, {"BISSAU", "Bissau"},

        {"CETE D'IVOIR (IVORY COAST)", "Yamoussoukro"},

        {"KENYA", "Nairobi"}, {"LESOTHO", "Maseru"},

        {"LIBERIA", "Monrovia"}, {"LIBYA", "Tripoli"},

        {"MADAGASCAR", "Antananarivo"}, {"MALAWI", "Lilongwe"},

        {"MALI", "Bamako"}, {"MAURITANIA", "Nouakchott"},

        {"MAURITIUS", "Port Louis"}, {"MOROCCO", "Rabat"},

        {"MOZAMBIQUE", "Maputo"}, {"NAMIBIA", "Windhoek"},

        {"NIGER", "Niamey"}, {"NIGERIA", "Abuja"},

        {"RWANDA", "Kigali"}, {"SAO TOME E PRINCIPE", "Sao Tome"},

        {"SENEGAL", "Dakar"}, {"SEYCHELLES", "Victoria"},

        {"SIERRA LEONE", "Freetown"}, {"SOMALIA", "Mogadishu"},

        {"SOUTH AFRICA", "Pretoria/Cape Town"}, {"SUDAN", "Khartoum"},

        {"SWAZILAND", "Mbabane"}, {"TANZANIA", "Dodoma"},

        {"TOGO", "Lome"}, {"TUNISIA", "Tunis"},

        {"UGANDA", "Kampala"},
    
```



```

{"DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)", "Kinshasa"},
{"ZAMBIA", "Lusaka"}, {"ZIMBABWE", "Harare"},
// Asie
{"AFGHANISTAN", "Kabul"}, {"BAHRAIN", "Manama"},
{"BANGLADESH", "Dhaka"}, {"BHUTAN", "Thimphu"},
{"BRUNEI", "Bandar Seri Begawan"}, {"CAMBODIA", "Phnom Penh"},
{"CHINA", "Beijing"}, {"CYPRUS", "Nicosia"},
{"INDIA", "New Delhi"}, {"INDONESIA", "Jakarta"},
{"IRAN", "Tehran"}, {"IRAQ", "Baghdad"},
{"ISRAEL", "Jerusalem"}, {"JAPAN", "Tokyo"},
{"JORDAN", "Amman"}, {"KUWAIT", "Kuwait City"},
{"LAOS", "Vientiane"}, {"LEBANON", "Beirut"},
{"MALAYSIA", "Kuala Lumpur"}, {"THE MALDIVES", "Male"},
{"MONGOLIA", "Ulan Bator"}, {"MYANMAR (BURMA)", "Rangoon"},
{"NEPAL", "Katmandu"}, {"NORTH KOREA", "P'yongyang"},
{"OMAN", "Muscat"}, {"PAKISTAN", "Islamabad"},
{"PHILIPPINES", "Manila"}, {"QATAR", "Doha"},
{"SAUDI ARABIA", "Riyadh"}, {"SINGAPORE", "Singapore"},
{"SOUTH KOREA", "Seoul"}, {"SRI LANKA", "Colombo"},
{"SYRIA", "Damascus"}, {"TAIWAN (REPUBLIC OF CHINA)", "Taipei"},
{"THAILAND", "Bangkok"}, {"TURKEY", "Ankara"},
{"UNITED ARAB EMIRATES", "Abu Dhabi"}, {"VIETNAM", "Hanoi"},
{"YEMEN", "Sana'a"},
// Australie et Océanie
{"AUSTRALIA", "Canberra"}, {"FIJI", "Suva"},
{"KIRIBATI", "Bairiki"},
{"MARSHALL ISLANDS", "Dalap-Uliga-Darrit"},
{"MICRONESIA", "Palikir"}, {"NAURU", "Yaren"},
{"NEW ZEALAND", "Wellington"}, {"PALAU", "Koror"},
{"PAPUA NEW GUINEA", "Port Moresby"},
{"SOLOMON ISLANDS", "Honaira"}, {"TONGA", "Nuku'alofa"},
{"TUVALU", "Fongafale"}, {"VANUATU", "Port-Vila"},
{"WESTERN SAMOA", "Apia"},
// Europe de l'Est et ancienne URSS

```

```

{ "ARMENIA", "Yerevan" }, { "AZERBAIJAN", "Baku" },
{ "BELARUS (BYELORUSSIA)", "Minsk" }, { "GEORGIA", "Tbilisi" },
{ "KAZAKSTAN", "Almaty" }, { "KYRGYZSTAN", "Alma-Ata" },
{ "MOLDOVA", "Chisinau" }, { "RUSSIA", "Moscow" },
{ "TAJIKISTAN", "Dushanbe" }, { "TURKMENISTAN", "Ashkabad" },
{ "UKRAINE", "Kyiv" }, { "UZBEKISTAN", "Tashkent" },
// Europe
{ "ALBANIA", "Tirana" }, { "ANDORRA", "Andorra la Vella" },
{ "AUSTRIA", "Vienna" }, { "BELGIUM", "Brussels" },
{ "BOSNIA", "-" }, { "HERZEGOVINA", "Sarajevo" },
{ "CROATIA", "Zagreb" }, { "CZECH REPUBLIC", "Prague" },
{ "DENMARK", "Copenhagen" }, { "ESTONIA", "Tallinn" },
{ "FINLAND", "Helsinki" }, { "FRANCE", "Paris" },
{ "GERMANY", "Berlin" }, { "GREECE", "Athens" },
{ "HUNGARY", "Budapest" }, { "ICELAND", "Reykjavik" },
{ "IRELAND", "Dublin" }, { "ITALY", "Rome" },
{ "LATVIA", "Riga" }, { "LIECHTENSTEIN", "Vaduz" },
{ "LITHUANIA", "Vilnius" }, { "LUXEMBOURG", "Luxembourg" },
{ "MACEDONIA", "Skopje" }, { "MALTA", "Valletta" },
{ "MONACO", "Monaco" }, { "MONTENEGRO", "Podgorica" },
{ "THE NETHERLANDS", "Amsterdam" }, { "NORWAY", "Oslo" },
{ "POLAND", "Warsaw" }, { "PORTUGAL", "Lisbon" },
{ "ROMANIA", "Bucharest" }, { "SAN MARINO", "San Marino" },
{ "SERBIA", "Belgrade" }, { "SLOVAKIA", "Bratislava" },
{ "SLOVENIA", "Ljubljana" }, { "SPAIN", "Madrid" },
{ "SWEDEN", "Stockholm" }, { "SWITZERLAND", "Berne" },
{ "UNITED KINGDOM", "London" }, { "VATICAN CITY", "---" },
// Amérique du Nord et Amérique Centrale
{ "ANTIGUA AND BARBUDA", "Saint John's" }, { "BAHAMAS", "Nassau" },
{ "BARBADOS", "Bridgetown" }, { "BELIZE", "Belmopan" },
{ "CANADA", "Ottawa" }, { "COSTA RICA", "San Jose" },
{ "CUBA", "Havana" }, { "DOMINICA", "Roseau" },
{ "DOMINICAN REPUBLIC", "Santo Domingo" },
{ "EL SALVADOR", "San Salvador" }, { "GRENADA", "Saint George's" },

```

```

    {"GUATEMALA", "Guatemala City"}, {"HAITI", "Port-au-Prince"},
    {"HONDURAS", "Tegucigalpa"}, {"JAMAICA", "Kingston"},
    {"MEXICO", "Mexico City"}, {"NICARAGUA", "Managua"},
    {"PANAMA", "Panama City"}, {"ST. KITTS", "-"},
    {"NEVIS", "Basseterre"}, {"ST. LUCIA", "Castries"},
    {"ST. VINCENT AND THE GRENADINES", "Kingstown"},
    {"UNITED STATES OF AMERICA", "Washington, D.C."},
    // Amérique du Sud
    {"ARGENTINA", "Buenos Aires"},
    {"BOLIVIA", "Sucre (legal)/La Paz(administrative)"},
    {"BRAZIL", "Brasilia"}, {"CHILE", "Santiago"},
    {"COLOMBIA", "Bogota"}, {"ECUADOR", "Quito"},
    {"GUYANA", "Georgetown"}, {"PARAGUAY", "Asuncion"},
    {"PERU", "Lima"}, {"SURINAME", "Paramaribo"},
    {"TRINIDAD AND TOBAGO", "Port of Spain"},
    {"URUGUAY", "Montevideo"}, {"VENEZUELA", "Caracas"},
};
} ///:~

```

Il s'agit juste d'un tableau bidimensionnel de **String**[\[48\]](#). Voici un simple test illustrant les méthodes **fill()** et les générateurs :

```

//: c09:FillTest.java

import com.bruceeckel.util.*;

import java.util.*;

public class FillTest {
    static Generator sg =
        new Arrays2.RandStringGenerator(7);

    public static void main(String[] args) {
        List list = new ArrayList();
        Collections2.fill(list, sg, 25);
        System.out.println(list + "\n");
        List list2 = new ArrayList();
        Collections2.fill(list2,

```

```

        Collections2.capitals, 25);

System.out.println(list2 + "\n");

Set set = new HashSet();

Collections2.fill(set, sg, 25);

System.out.println(set + "\n");

Map m = new HashMap();

Collections2.fill(m, Collections2.rsp, 25);

System.out.println(m + "\n");

Map m2 = new HashMap();

Collections2.fill(m2,

    Collections2.geography, 25);

System.out.println(m2);
    }
} ///:~

```

Avec ces outils vous pourrez facilement tester les différents conteneurs en les remplissant avec des données intéressantes.

L'inconvénient des conteneurs : le type est inconnu

L'« inconvénient » des conteneurs Java est qu'on perd l'information du type lorsqu'un objet est stocké dedans, ce qui est tout à fait normal puisque le programmeur de la classe conteneur n'a aucune idée du type spécifique qu'on veut stocker dans le conteneur, et que fournir un conteneur qui ne sache stocker qu'un seul type d'objets irait à l'encontre du but de généricité de l'outil conteneur. C'est pourquoi les conteneurs stockent des références sur des **Objects**, la classe de base de toutes les classes, afin de pouvoir stocker n'importe quel type d'objet (à l'exception bien sûr des types scalaires, qui ne dérivent aucune classe). Cette solution est formidable dans sa conception, sauf sur deux points :

1. Puisque l'information de type est ignorée lorsqu'on stocke une référence dans un conteneur, on ne peut placer aucune restriction sur le type de l'objet stocké dans le conteneur, même si on l'a créé pour ne contenir, par exemple, que des chats. Quelqu'un pourrait très bien ajouter un chien dans le conteneur.
2. Puisque l'information de type est perdue, la seule chose que le conteneur sache est qu'il contient une référence sur un objet. Il faut réaliser un transtypage sur le type adéquat avant de l'utiliser.

Du côté des choses positives, Java ne permettra pas une mauvaise utilisation des objets stockés dans un conteneur. Si on stocke un chien dans le conteneur de chats et qu'on essaie ensuite de traiter tous les objets du conteneur comme un chat, Java générera une *run-time* exception lors de la tentative de transtypage en chat de la référence sur le chien.

Voici un exemple utilisant le conteneur à tout faire **ArrayList**. Les débutants peuvent considérer une **ArrayList** comme « un tableau qui se redimensionne de lui-même ». L'utilisation d'une **ArrayList** est aisée : il suffit de la créer, d'y ajouter des éléments avec la méthode **add()**, et d'y accéder par la suite grâce à la méthode **get()** en utilisant un index - comme pour un tableau, mais sans les crochets [49]. **ArrayList** propose aussi une méthode **size()** qui permet de savoir combien d'éléments ont été stockés afin de ne pas dépasser les frontières et

causer une exception.

Tout d'abord, nous créons les classes **Cat** et **Dog** :

```
//: c09:Cat.java
public class Cat {
    private int catNumber;

    Cat(int i) { catNumber = i; }

    void print() {
        System.out.println("Cat #" + catNumber);
    }
} ///:~
```

```
//: c09:Dog.java
public class Dog {
    private int dogNumber;

    Dog(int i) { dogNumber = i; }

    void print() {
        System.out.println("Dog #" + dogNumber);
    }
} ///:~
```

Des **Cats** et des **Dogs** sont placés dans le conteneur, puis extraits :

```
//: c09:CatsAndDogs.java
// Exemple simple avec un conteneur.
import java.util.*;

public class CatsAndDogs {
    public static void main(String[] args) {
        ArrayList cats = new ArrayList();

        for(int i = 0; i < 7; i++)
            cats.add(new Cat(i));

        // Ce n'est pas un problème d'ajouter un chien parmi les chats :
        cats.add(new Dog(7));

        for(int i = 0; i < cats.size(); i++)
```

```

        ((Cat)cats.get(i)).print();

        // Le chien est détecté seulement lors de l'exécution.
    }

} ///:~

```

Les classes **Cat** et **Dog** sont distinctes - elles n'ont rien en commun sinon le fait que ce sont des **Objects** (dans le cas où une classe ne spécifie pas de classe de base, elle hérite automatiquement de la classe **Object**). Comme l'**ArrayList** contient des **Objects**, on peut stocker aussi bien des objets **Cat** que des objets **Dog** via la méthode **add()** sans aucune erreur, aussi bien lors de la compilation que lors de l'exécution du programme. Par contre, lorsqu'une référence sur ce qu'on pense être un **Cat** est extraite via la méthode **get()** de **ArrayList**, il faut la transtyper en **Cat**. Pour éviter une erreur de syntaxe, il faut entourer l'expression par des parenthèses pour forcer l'évaluation du transtypage avant d'appeler la méthode **print()** de **Cat**. Mais à l'exécution, lorsqu'on tente de transtyper un objet **Dog** en **Cat**, la machine virtuelle Java générera une exception.

Ceci est plus qu'ennuyeux. Cela peut mener de plus à des bugs relativement durs à trouver. Si une partie (ou plusieurs parties) du programme insère des objets dans le conteneur, et qu'on découvre dans une partie complètement différente du programme via une exception qu'un objet du mauvais type a été placé dans le conteneur, il faut alors déterminer où l'insertion coupable s'est produite. Cependant, il est pratique de démarrer avec les classes conteneur standard pour programmer, en dépit de leurs limitations et de leur lourdeur.

Quelquefois ça marche quand même

Dans certains cas les choses semblent fonctionner correctement sans avoir à transtyper vers le type originel. Un cas particulier est constitué par la classe **String** que le compilateur traite de manière particulière pour la faire fonctionner de manière idoine. Quand le compilateur attend un objet **String** et qu'il obtient autre chose, il appellera automatiquement la méthode **toString()** définie dans **Object** et qui peut être redéfinie par chaque classe Java. Cette méthode produit l'objet **String** désiré, qui est ensuite utilisé là où il était attendu.

Il suffit donc de redéfinir la méthode **toString()** pour afficher un objet d'une classe donnée, comme on peut le voir dans l'exemple suivant :

```

//: c09:Mouse.java
// Redéfinition de toString().

public class Mouse {

    private int mouseNumber;

    Mouse(int i) { mouseNumber = i; }

    // Redéfinition de Object.toString():

    public String toString() {

        return "This is Mouse #" + mouseNumber;

    }

    public int getNumber() {

        return mouseNumber;

    }

}

```

```

    }
} ///:~

//: c09:WorksAnyway.java
// Dans certains cas spéciaux, les choses
// semblent fonctionner correctement.
import java.util.*;

class MouseTrap {
    static void caughtYa(Object m) {
        Mouse mouse = (Mouse)m; // Transtypage depuis un Object
        System.out.println("Mouse: " +
            mouse.getNumber());
    }
}

public class WorksAnyway {
    public static void main(String[] args) {
        ArrayList mice = new ArrayList();
        for(int i = 0; i < 3; i++)
            mice.add(new Mouse(i));
        for(int i = 0; i < mice.size(); i++) {
            // Aucun transtypage nécessaire, appel
            // automatique à Object.toString() :
            System.out.println(
                "Free mouse: " + mice.get(i));
            MouseTrap.caughtYa(mice.get(i));
        }
    }
} ///:~

```

La méthode **toString()** est redéfinie dans **Mouse**. Dans la deuxième boucle **for** de **main()** on peut voir l'instruction :

```
System.out.println("Free mouse: " + mice.get(i));
```

Après le signe « + » le compilateur s'attend à trouver un objet **String**. **get()** renvoie un **Object**, le compilateur appelle donc implicitement la méthode **toString()** pour obtenir l'objet **String** désiré. Cependant, ce comportement magique n'est possible qu'avec les **String**, il n'est pas disponible pour les autres types. Une seconde approche pour cacher le transtypage est de le placer dans la classe **MouseTrap**. La méthode **caughtYa()** n'accepte pas une **Mouse** mais un **Object**, qu'elle transtype alors en **Mouse**. Ceci ne fait que repousser le problème puisqu'en acceptant un **Object** on peut passer un objet de n'importe quel type à la méthode. Cependant, si le transtypage n'est pas valide - si un objet du mauvais type est passé en argument - une exception est générée lors de l'exécution. Ce n'est pas aussi bien qu'un contrôle lors de la compilation, mais l'approche reste robuste. Notez qu'aucun transtypage n'est nécessaire lors de l'utilisation de cette méthode :

```
MouseTrap.caughtYa(mice.get(i));
```

Créer une ArrayList consciente du type

Pas question toutefois de s'arrêter en si bon chemin. Une solution encore plus robuste consiste à créer une nouvelle classe utilisant une **ArrayList**, n'acceptant et ne produisant que des objets du type voulu :

```
///  
// Une ArrayList consciente du type.  
import java.util.*;  
  
public class MouseList {  
    private ArrayList list = new ArrayList();  
    public void add(Mouse m) {  
        list.add(m);  
    }  
    public Mouse get(int index) {  
        return (Mouse)list.get(index);  
    }  
    public int size() { return list.size(); }  
} ///:~
```

Voici un test pour le nouveau conteneur :

```
///  
public class MouseListTest {  
    public static void main(String[] args) {  
        MouseList mice = new MouseList();  
        for(int i = 0; i < 3; i++)
```



```

        mice.add(new Mouse(i));

    for(int i = 0; i < mice.size(); i++)

        MouseTrap.caughtYa(mice.get(i));

    }

} ///:~

```

Cet exemple est similaire au précédent, sauf que la nouvelle classe **MouseListener** dispose d'un membre **private** de type **ArrayList**, et de méthodes identiques à celles fournies par **ArrayList**. Cependant, ces méthodes n'acceptent et ne produisent pas des **Objects** génériques, mais seulement des objets **Mouse**.

Notez que si **MouseListener** avait été *dérivée* de **ArrayList**, la méthode **add(Mouse)** aurait simplement surchargé la méthode existante **add(Object)** et aucune restriction sur le type d'objets acceptés n'aurait donc été ajoutée. La **MouseListener** est donc un *substitut* à l'**ArrayList**, réalisant certaines opérations avant de déléguer la responsabilité (cf. *Thinking in Patterns with Java*, téléchargeable sur www.BruceEckel.com).

Du fait qu'une **MouseListener** n'accepte qu'une **Mouse**, l'instruction suivante :

```
mice.add(new Pigeon());
```

provoque un message d'erreur durant la *phase de compilation*. Cette approche, bien que plus fastidieuse du point de vue du code, signalera immédiatement si on utilise un type de façon incorrecte.

Notez aussi qu'aucun transtypage n'est nécessaire lors d'un appel à **get()** - elle renvoie toujours une **Mouse**.

Types paramétrés

Ce type de problème n'est pas isolé - nombreux sont les cas dans lesquels on a besoin de créer de nouveaux types basés sur d'autres types, et dans lesquels il serait bon de récupérer des informations de type lors de la phase de compilation. C'est le concept des *types paramétrés*. En C++, ceci est directement supporté par le langage via les *templates*. Les futures versions de Java supporteront probablement une implémentation des types paramétrés ; actuellement il faut se contenter de créer des classes similaires à **MouseListener**.

Itérateurs

Chaque classe conteneur fournit des méthodes pour stocker des objets et pour les extraire - après tout, le but d'un conteneur est de stocker des choses. Dans **ArrayList**, on insère des objets via la méthode **add()**, et **get()** est l'un des moyens de récupérer ces objets. **ArrayList** est relativement souple - il est possible de sélectionner n'importe quel élément à n'importe quel moment, ou de sélectionner plusieurs éléments en même temps en utilisant différents index.

Si on se place à un niveau d'abstraction supérieur, on s'aperçoit d'un inconvénient : on a besoin de connaître le type exact du conteneur afin de l'utiliser. Ceci peut sembler bénin à première vue, mais qu'en est-il si on commence à programmer en utilisant une **ArrayList**, et qu'on se rende compte par la suite qu'il serait plus efficace d'utiliser une **LinkedList** à la place ? Ou alors si on veut écrire une portion de code générique qui ne connaît pas le type de conteneur avec lequel elle travaille, afin de pouvoir être utilisé avec différents types de conteneurs sans avoir à réécrire ce code ?

Le concept d'*itérateur* peut être utilisé pour réaliser cette abstraction. Un itérateur est un objet dont le travail est de se déplacer dans une séquence d'objets et de sélectionner chaque objet de cette séquence sans que le programmeur client n'ait à se soucier de la structure sous-jacente de cette séquence. De plus, un itérateur est généralement ce qu'il est convenu d'appeler un objet « léger » : un objet bon marché à construire. Pour cette raison, vous trouverez souvent des contraintes étranges sur les itérateurs ; par exemple, certains itérateurs ne peuvent se déplacer que dans un sens.

L'**Iterator** Java est l'exemple type d'un itérateur avec ce genre de contraintes. On ne peut faire grand-chose avec mis à part :

1. Demander à un conteneur de renvoyer un **Iterator** en utilisant une méthode appelée **iterator()**. Cet **Iterator** sera prêt à renvoyer le premier élément dans la séquence au premier appel à sa méthode **next()**.
2. Récupérer l'objet suivant dans la séquence grâce à sa méthode **next()**.
3. Vérifier s'il reste *encore* d'autres objets dans la séquence via la méthode **hasNext()**.
4. Enlever le dernier élément renvoyé par l'itérateur avec la méthode **remove()**.

Et c'est tout. C'est une implémentation simple d'un itérateur, mais néanmoins puissante (et il existe un **ListIterator** plus sophistiqué pour les **Lists**). Pour le voir en action, revisitons le programme **CatsAndDogs.java** rencontré précédemment dans ce chapitre. Dans sa version originale, la méthode **get()** était utilisée pour sélectionner chaque élément, mais la version modifiée suivante se sert d'un **Iterator** :

```

//: c09:CatsAndDogs2.java

// Conteneur simple utilisant un Iterator.

import java.util.*;

public class CatsAndDogs2 {

    public static void main(String[] args) {

        ArrayList cats = new ArrayList();

        for(int i = 0; i < 7; i++)

            cats.add(new Cat(i));

        Iterator e = cats.iterator();

        while(e.hasNext())

            ((Cat)e.next()).print();

    }

} ///:~

```

Les dernières lignes utilisent maintenant un **Iterator** pour se déplacer dans la séquence à la place d'une boucle **for**. Avec l'**Iterator**, on n'a pas besoin de se soucier du nombre d'éléments dans le conteneur. Cela est géré via les méthodes **hasNext()** et **next()**.

Comme autre exemple, considérons maintenant la création d'une méthode générique d'impression :

```

//: c09:HamsterMaze.java

```

```

// Utilisation d'un Iterator.

import java.util.*;

class Hamster {

    private int hamsterNumber;

    Hamster(int i) { hamsterNumber = i; }

    public String toString() {

        return "This is Hamster #" + hamsterNumber;

    }

}

class Printer {

    static void printAll(Iterator e) {

        while(e.hasNext())

            System.out.println(e.next());

    }

}

public class HamsterMaze {

    public static void main(String[] args) {

        ArrayList v = new ArrayList();

        for(int i = 0; i < 3; i++)

            v.add(new Hamster(i));

        Printer.printAll(v.iterator());

    }

} ///:~

```

Examinez attentivement la méthode **printAll()** et notez qu'elle ne dispose d'aucune information sur le type de séquence. Tout ce dont elle dispose est un **Iterator**, et c'est la seule chose dont elle a besoin pour utiliser la séquence : elle peut récupérer l'objet suivant, et savoir si elle se trouve à la fin de la séquence. Cette idée de prendre un conteneur d'objets et de le parcourir pour réaliser une opération sur chaque élément est un concept puissant, et on le retrouvera tout au long de ce livre.

Cet exemple est même encore plus générique, puisqu'il utilise implicitement la méthode **Object.toString()**. La méthode **println()** est surchargée pour tous les types scalaires ainsi que dans **Object** ; dans chaque cas une **String** est automatiquement produite en appelant la méthode **toString()** appropriée.

Bien que cela ne soit pas nécessaire, on pourrait être plus explicite en transtypant le résultat, ce qui aurait pour effet d'appeler `toString()` :

```
System.out.println((String)e.next());
```

En général, cependant, on voudra certainement aller au-delà de l'appel des méthodes de la classe **Object**, et on se heurte de nouveau au problème du transtypage. Il faut donc assumer qu'on a récupéré un **Iterator** sur une séquence contenant des objets du type particulier qui nous intéresse, et transtyper les objets dans ce type (et recevoir une exception à l'exécution si on se trompe).

Récursion indésirable

Comme les conteneurs standard Java héritent de la classe **Object** (comme toutes les autres classes), ils contiennent une méthode `toString()`. Celle-ci a été redéfinie afin de produire une représentation **String** d'eux-mêmes, incluant les objets qu'ils contiennent. A l'intérieur d'**ArrayList**, la méthode `toString()` parcourt les éléments de l'**ArrayList** et appelle `toString()` pour chacun d'eux. Supposons qu'on veuille afficher l'adresse de l'instance. Il semble raisonnable de se référer à **this** (en particulier pour les programmeurs C++ qui sont habitués à cette approche) :

```
/// c09:InfiniteRecursion.java
// Récursion accidentelle.
import java.util.*;

public class InfiniteRecursion {
    public String toString() {
        return " InfiniteRecursion address: "
            + this + "\n";
    }
    public static void main(String[] args) {
        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new InfiniteRecursion());
        System.out.println(v);
    }
} ///::~
```

Si on crée un objet **InfiniteRecursion** et qu'on veut l'afficher, on se retrouve avec une séquence infinie d'exceptions. C'est également vrai si on place des objets **InfiniteRecursion** dans une **ArrayList** et qu'on imprime cette **ArrayList** comme c'est le cas ici. Ceci est du à la conversion automatique de type sur les **Strings**. Quand on écrit :

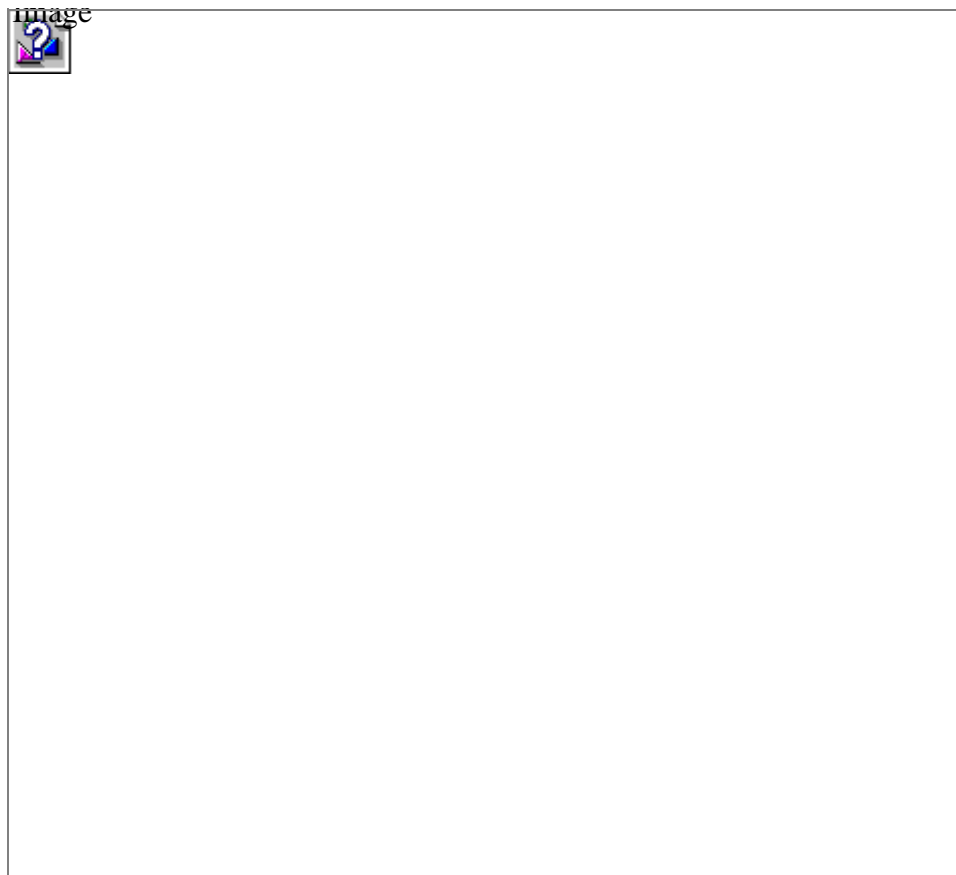
```
"InfiniteRecursion address: " + this
```

Le compilateur voit une **String** suivie par un « + » et quelque chose qui n'est pas une **String**, il essaie donc de convertir **this** en **String**. Il réalise cette conversion en appelant **toString()**, ce qui produit un appel récursif.

Si on veut réellement imprimer l'adresse de l'objet dans ce cas, la solution est d'appeler la méthode **Object.toString()**, qui réalise exactement ceci. Il faut donc utiliser **super.toString()** à la place de **this** (ceci ne fonctionnera que si on hérite directement de la classe **Object**, ou si aucune classe parent n'a redéfini la méthode **toString()**).

Classification des conteneurs

Les **Collections** et les **Maps** peuvent être implémentés de différentes manières, à vous de choisir la bonne selon vos besoins. Le diagramme suivant peut aider à s'y retrouver parmi les conteneurs Java 2 :



Ce diagramme peut sembler un peu surchargé à première vue, mais il n'y a en fait que trois types conteneurs de base : les **Maps**, les **Lists** et les **Sets**, chacun d'entre eux ne proposant que deux ou trois implémentations (avec typiquement une version préférée). Quand on se ramène à cette observation, les conteneurs ne sont plus aussi intimidants.

Les boîtes en pointillé représentent les **interfaces**, les boîtes en tirets représentent des classes **abstract**, et les boîtes pleines sont des classes normales (concrètes). Les lignes pointillées indiquent qu'une classe particulière implémente une **interface** (ou dans le cas d'une classe **abstract**, implémente partiellement cette **interface**). Une flèche pleine indique qu'une classe peut produire des objets de la classe sur laquelle la flèche pointe. Par

exemple, une **Collection** peut produire un **Iterator**, tandis qu'une **List** peut produire un **ListIterator** (ainsi qu'un **Iterator** ordinaire, puisque **List** est dérivée de **Collection**).

Les interfaces concernées par le stockage des objets sont **Collection**, **List**, **Set** et **Map**. Idéalement, la majorité du code qu'on écrit sera destinée à ces interfaces, et le seul endroit où on spécifiera le type précis utilisé est lors de la création. On pourra donc créer une **List** de cette manière :

```
List x = new LinkedList();
```

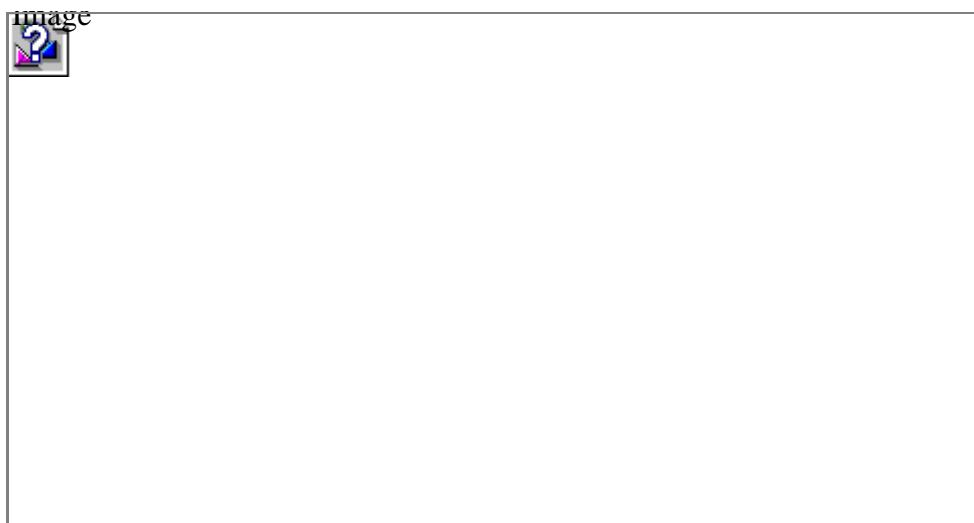
Bien sûr, on peut aussi décider de faire de **x** une **LinkedList** (au lieu d'une **List** générique) et véhiculer le type précis d'informations avec **x**. La beauté (et l'intérêt) de l'utilisation d'une interface est que si on décide de changer l'implémentation, la seule chose qu'on ait besoin de changer est l'endroit où la liste est créée, comme ceci :

```
List x = new ArrayList();
```

Et on ne touche pas au reste du code (une telle généricité peut aussi être réalisée via des itérateurs).

Dans la hiérarchie de classes, on peut voir un certain nombre de classes dont le nom débute par « **Abstract** », ce qui peut paraître un peu déroutant au premier abord. Ce sont simplement des outils qui implémentent partiellement une interface particulière. Si on voulait réaliser notre propre **Set**, par exemple, il serait plus simple de dériver **AbstractSet** et de réaliser le travail minimum pour créer la nouvelle classe, plutôt que d'implémenter l'interface **Set** et toutes les méthodes qui vont avec. Cependant, la bibliothèque de conteneurs possède assez de fonctionnalités pour satisfaire quasiment tous nos besoins. De notre point de vue, nous pouvons donc ignorer les classes débutant par « **Abstract** ».

Ainsi, lorsqu'on regarde le diagramme, on n'est réellement concerné que par les **interfaces** du haut du diagramme et les classes concrètes (celles qui sont entourées par des boîtes solides). Typiquement, on se contentera de créer un objet d'une classe concrète, de la transtyper dans son **interface** correspondante, et ensuite utiliser cette **interface** tout au long du code. De plus, on n'a pas besoin de se préoccuper des éléments pré-existants lorsqu'on produit du nouveau code. Le diagramme peut donc être grandement simplifié pour ressembler à ceci :



Il n'inclut plus maintenant que les classes et les interfaces que vous serez amenés à rencontrer régulièrement,

ainsi que les éléments sur lesquels nous allons nous pencher dans ce chapitre.

Voici un exemple simple, qui remplit une **Collection** (représenté ici par une **ArrayList**) avec des objets **String**, et affiche ensuite chaque élément de la **Collection** :

```
//: c09:SimpleCollection.java
// Un exemple simple d'utilisation des Collections Java 2.
import java.util.*;

public class SimpleCollection {
    public static void main(String[] args) {
        // Transtypage ascendant parce qu'on veut juste
        // travailler avec les fonctionnalités d'une Collection
        Collection c = new ArrayList();
        for(int i = 0; i < 10; i++)
            c.add(Integer.toString(i));
        Iterator it = c.iterator();
        while(it.hasNext())
            System.out.println(it.next());
    }
} ///:~
```

La première ligne de **main()** crée un objet **ArrayList** et le transtype ensuite en une **Collection**. Puisque cet exemple n'utilise que les méthodes de **Collection**, tout objet d'une classe dérivée de **Collection** fonctionnerait, mais l'**ArrayList** est la **Collection** à tout faire typique.

La méthode **add()**, comme son nom le suggère, ajoute un nouvel élément dans la **Collection**. En fait, la documentation précise bien que **add()** « assure que le conteneur contiendra l'élément spécifié ». Cette précision concerne les **Sets**, qui n'ajoutent un élément que s'il n'est pas déjà présent. Avec une **ArrayList**, ou n'importe quel type de **List**, **add()** veut toujours dire « stocker dans », parce qu'une **List** se moque de contenir des doublons.

Toutes les **Collections** peuvent produire un **Iterator** grâce à leur méthode **iterator()**. Ici, un **Iterator** est créé et utilisé pour traverser la **Collection**, en affichant chaque élément.

Fonctionnalités des Collections

La table suivante contient toutes les opérations définies pour une **Collection** (sans inclure les méthodes directement héritées de la classe **Object**), et donc pour un **Set** ou une **List** (les **Lists** possèdent aussi d'autres fonctionnalités). Les **Maps** n'héritant pas de **Collection**, elles seront traitées séparément.

boolean add(Object)	Assure que le conteneur stocke l'argument. Renvoie false si elle n'ajoute pas l'argument (c'est une méthode « optionnelle », décrite plus tard dans ce chapitre).
boolean addAll(Collection)	Ajoute tous les éléments de l'argument. Renvoie true si un élément a été ajouté (« optionnelle »).
void clear()	Supprime tous les éléments du conteneur (« optionnelle »).
boolean contains(Object)	true si le conteneur contient l'argument.
boolean containsAll(Collection)	true si le conteneur contient tous les éléments de l'argument.
boolean isEmpty()	true si le conteneur ne contient pas d'éléments.
Iterator iterator()	Renvoie un Iterator qu'on peut utiliser pour parcourir les éléments du conteneur.
boolean remove(Object)	Si l'argument est dans le conteneur, une instance de cet élément est enlevée. Renvoie true si c'est le cas (« optionnelle »).
boolean removeAll(Collection)	Supprime tous les éléments contenus dans l'argument. Renvoie true si au moins une suppression a été effectuée (« optionnelle »).
boolean retainAll(Collection)	Ne garde que les éléments contenus dans l'argument (une « intersection » selon la théorie des ensembles). Renvoie true s'il y a eu un changement (« optionnelle »).
int size()	Renvoie le nombre d'éléments dans le conteneur.
Object[] toArray()	Renvoie un tableau contenant tous les éléments du conteneur.
Object[] toArray(Object[] a)	Renvoie un tableau contenant tous les éléments du conteneur, dont le type est celui du tableau a au lieu d' Objects génériques (il faudra toutefois transtyper le tableau dans son type correct).

Notez qu'il n'existe pas de fonction **get()** permettant un accès aléatoire. Ceci parce que les **Collections** contiennent aussi les **Sets**, qui maintiennent leur propre ordre interne, faisant de toute tentative d'accès aléatoire un non-sens. Il faut donc utiliser un **Iterator** pour parcourir tous les éléments d'une **Collection** ; c'est la seule façon de récupérer les objets stockés.

L'exemple suivant illustre toutes ces méthodes. Encore une fois, cet exemple marcherait avec tout objet héritant de **Collection**, mais nous utilisons ici une **ArrayList** comme « plus petit dénominateur commun » :

```

//: c09:Collection1.java
// Opérations disponibles sur les Collections.

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

public class Collection1 {
    public static void main(String[] args) {
        Collection c = new ArrayList();

        Collections2.fill(c,
            Collections2.countries, 10);
    }
}

```



```
c.add("ten");
c.add("eleven");
System.out.println(c);
// Crée un tableau à partir de la List :
Object[] array = c.toArray();
// Crée un tableau de Strings à partir de la List :
String[] str =
    (String[])c.toArray(new String[1]);
// Trouve les éléments mini et maxi ; ceci peut
// signifier différentes choses suivant la manière
// dont l'interface Comparable est implémentée :
System.out.println("Collections.max(c) = " +
    Collections.max(c));
System.out.println("Collections.min(c) = " +
    Collections.min(c));
// Ajoute une Collection à une autre Collection
Collection c2 = new ArrayList();
Collections2.fill(c2,
    Collections2.countries, 10);
c.addAll(c2);
System.out.println(c);
c.remove(CountryCapitals.pairs[0][0]);
System.out.println(c);
c.remove(CountryCapitals.pairs[1][0]);
System.out.println(c);
// Supprime tous les éléments
// de la Collection argument :
c.removeAll(c2);
System.out.println(c);
c.addAll(c2);
System.out.println(c);
// Est-ce qu'un élément est dans la Collection ?
String val = CountryCapitals.pairs[3][0];
System.out.println(
```

```

        "c.contains(" + val + ") = "
        + c.contains(val));

// Est-ce qu'une Collection est contenue dans la Collection ?
System.out.println(
    "c.containsAll(c2) = " + c.containsAll(c2));
Collection c3 = ((List)c).subList(3, 5);
// Garde les éléments présents à la fois dans
// c2 et c3 (intersection d'ensembles) :
c2.retainAll(c3);
System.out.println(c);
// Supprime tous les éléments
// de c2 contenus dans c3 :
c2.removeAll(c3);
System.out.println("c.isEmpty() = " +
    c.isEmpty());
c = new ArrayList();
Collections2.fill(c,
    Collections2.countries, 10);
System.out.println(c);
c.clear(); // Supprime tous les éléments
System.out.println("after c.clear():");
System.out.println(c);
}
} ///:~

```

Les **ArrayLists** sont créées et initialisées avec différents ensembles de données, puis transtypées en objets **Collection** ; il est donc clair que seules les fonctions de l'interface **Collection** sont utilisées. **main()** réalise de simples opérations pour illustrer toutes les méthodes de **Collection**.

Les sections suivantes décrivent les diverses implémentations des **Lists**, **Sets** et **Maps** et indiquent dans chaque cas (à l'aide d'une astérisque) laquelle devrait être votre choix par défaut. Vous noterez que les classes pré-existantes **Vector**, **Stack** et **Hashtable** ne sont *pas* incluses car certains conteneurs Java 2 fournissent les mêmes fonctionnalités.

Fonctionnalités des Lists

La **List** de base est relativement simple à utiliser, comme vous avez pu le constater jusqu'à présent avec les **ArrayLists**. Mis à part les méthodes courantes **add()** pour insérer des objets, **get()** pour les retrouver un par un,

et **iterator()** pour obtenir un **Iterator** sur la séquence, les listes possèdent par ailleurs tout un ensemble de méthodes qui peuvent se révéler très pratiques.

Les **Lists** sont déclinées en deux versions : l'**ArrayList** de base, qui excelle dans les accès aléatoires aux éléments, et la **LinkedList**, bien plus puissante (qui n'a pas été conçue pour un accès aléatoire optimisé, mais dispose d'un ensemble de méthodes bien plus conséquent).

List (interface)	L'ordre est la caractéristique la plus importante d'une List ; elle garantit de maintenir les éléments dans un ordre particulier. Les Lists disposent de méthodes supplémentaires permettant l'insertion et la suppression d'éléments au sein d'une List (ceci n'est toutefois recommandé que pour une LinkedList). Une List produit des ListIterators , qui permettent de parcourir la List dans les deux directions, d'insérer et de supprimer des éléments au sein de la List .
ArrayList*	Une List implémentée avec un tableau. Permet un accès aléatoire instantané aux éléments, mais se révèle inefficace lorsqu'on insère ou supprime un élément au milieu de la liste. Le ListIterator ne devrait être utilisé que pour parcourir l' ArrayList dans les deux sens, et non pour l'insertion et la suppression d'éléments, opérations coûteuses comparées aux LinkedLists .
LinkedList	Fournit un accès séquentiel optimal, avec des coûts d'insertion et de suppression d'éléments au sein de la List négligeables. Relativement lente pour l'accès aléatoire (préférer une ArrayList pour cela). Fournit aussi les méthodes addFirst() , addLast() , getFirst() , getLast() , removeFirst() et removeLast() (qui ne sont définies dans aucune interface ou classe de base) afin de pouvoir l'utiliser comme une pile, une file (une queue) ou une file double (queue à double entrée).

Les méthodes dans l'exemple suivant couvrent chacune un groupe de fonctionnalités : les opérations disponibles pour toutes les listes (**basicTest()**), le déplacement dans une liste avec un **Iterator** (**iterMotion()**) ainsi que la modification dans une liste avec un **Iterator** (**iterManipulation()**), la visualisation des manipulations sur la **List** (**testVisual()**) et les opérations disponibles uniquement pour les **LinkedLists**.

```

//: c09:List1.java

// Opérations disponibles sur les Lists.

import java.util.*;

import com.bruceeckel.util.*;

public class List1 {

    public static List fill(List a) {

        Collections2.countries.reset();

        Collections2.fill(a,

            Collections2.countries, 10);

        return a;

    }

    static boolean b;

```

```
static Object o;

static int i;

static Iterator it;

static ListIterator lit;

public static void basicTest(List a) {

    a.add(1, "x"); // Ajout à l'emplacement 1

    a.add("x"); // Ajout à la fin

    // Ajout d'une Collection :

    a.addAll(fill(new ArrayList()));

    // Ajout d'une Collection à partir du 3ème élément :

    a.addAll(3, fill(new ArrayList()));

    b = a.contains("1"); // L'élément est-il présent ?

    // La Collection entière est-elle présente ?

    b = a.containsAll(fill(new ArrayList()));

    // Les Lists permettent un accès aléatoire aux éléments,

    // bon marché pour les ArrayLists, coûteux pour les LinkedLists :

    o = a.get(1); // Récupère l'objet du premier emplacement

    i = a.indexOf("1"); // Donne l'index de l'objet

    b = a.isEmpty(); // La List contient-elle des éléments ?

    it = a.iterator(); // Iterator de base

    lit = a.listIterator(); // ListIterator

    lit = a.listIterator(3); // Démarre au 3ème élément

    i = a.lastIndexOf("1"); // Dernière concordance

    a.remove(1); // Supprime le premier élément

    a.remove("3"); // Supprime cet objet

    a.set(1, "y"); // Positionne le premier élément à "y"

    // Garde tous les éléments présents dans l'argument

    // (intersection de deux ensembles) :

    a.retainAll(fill(new ArrayList()));

    // Supprime tous les éléments présents dans l'argument :

    a.removeAll(fill(new ArrayList()));

    i = a.size(); // Taille de la List ?

    a.clear(); // Supprime tous les éléments

}
```

```
public static void iterMotion(List a) {  
    ListIterator it = a.listIterator();  
  
    b = it.hasNext();  
    b = it.hasPrevious();  
    o = it.next();  
    i = it.nextIndex();  
    o = it.previous();  
    i = it.previousIndex();  
}  
  
public static void iterManipulation(List a) {  
    ListIterator it = a.listIterator();  
  
    it.add("47");  
    // Doit aller sur un élément après add() :  
    it.next();  
    // Supprime l'élément qui vient d'être produit :  
    it.remove();  
    // Doit aller sur un élément après remove() :  
    it.next();  
    // Change l'élément qui vient d'être produit :  
    it.set("47");  
}  
  
public static void testVisual(List a) {  
    System.out.println(a);  
  
    List b = new ArrayList();  
    fill(b);  
    System.out.print("b = ");  
    System.out.println(b);  
    a.addAll(b);  
    a.addAll(fill(new ArrayList()));  
    System.out.println(a);  
  
    // Insère, supprime et remplace des éléments  
    // en utilisant un ListIterator :  
    ListIterator x = a.listIterator(a.size()/2);  
    x.add("one");  
}
```

```
System.out.println(a);

System.out.println(x.next());

x.remove();

System.out.println(x.next());

x.set("47");

System.out.println(a);

// Traverse la liste à l'envers :

x = a.listIterator(a.size());

while(x.hasPrevious())

    System.out.print(x.previous() + " ");

System.out.println();

System.out.println("testVisual finished");
}

// Certaines opérations ne sont disponibles
// que pour des LinkedLists :

public static void testLinkedList() {

    LinkedList ll = new LinkedList();

    fill(ll);

    System.out.println(ll);

    // Utilisation comme une pile, insertion (push) :

    ll.addFirst("one");

    ll.addFirst("two");

    System.out.println(ll);

    // Utilisation comme une pile, récupération de la valeur du premier élément (peek

    System.out.println(ll.getFirst());

    // Utilisation comme une pile, suppression (pop) :

    System.out.println(ll.removeFirst());

    System.out.println(ll.removeFirst());

    // Utilisation comme une file, en retirant les

    // éléments à la fin de la liste :

    System.out.println(ll.removeLast());

    // Avec les opérations ci-dessus, c'est une file double !

    System.out.println(ll);

}
```

```

public static void main(String[] args) {
    // Crée et remplit une nouvelle List à chaque fois :
    basicTest(fill(new LinkedList()));
    basicTest(fill(new ArrayList()));
    iterMotion(fill(new LinkedList()));
    iterMotion(fill(new ArrayList()));
    iterManipulation(fill(new LinkedList()));
    iterManipulation(fill(new ArrayList()));
    testVisual(fill(new LinkedList()));
    testLinkedList();
}
} ///:~

```

basicTest() et **iterMotion()** sont des procédures qui n'ont d'autre but que celui de montrer la syntaxe d'appel des méthodes dont la valeur de retour n'est pas utilisée bien que récupérée. Dans certains cas, la valeur de retour des méthodes n'est même pas récupérée car jamais utilisée même dans un programme réel. Se référer à la documentation online de *java.sun.com* pour un examen approfondi de l'utilisation de chacune de ces méthodes.

Réaliser une pile à partir d'une LinkedList

Une pile est un conteneur « dernier arrivé, premier sorti » (LIFO - « Last In, First Out »). C'est à dire que l'objet qu'on « pousse » (« push ») sur la pile en dernier sera le premier accessible lors d'une extraction (« pop »). Comme tous les autres conteneurs de Java, on stocke et récupère des **Objects**, qu'il faudra donc retrans typer après leur extraction, à moins qu'on ne se contente des fonctionnalités de la classe **Object**.

La classe **LinkedList** possède des méthodes qui implémentent directement les fonctionnalités d'une pile, on peut donc utiliser directement une **LinkedList** plutôt que de créer une classe implémentant une pile. Cependant une classe est souvent plus explicite :

```

//: c09:StackL.java
// Réaliser une pile à partir d'une LinkedList.

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

public class StackL {
    private LinkedList list = new LinkedList();

    public void push(Object v) {
        list.addFirst(v);
    }
}

```

```

public Object top() { return list.getFirst(); }

public Object pop() {
    return list.removeFirst();
}

public static void main(String[] args) {
    StackL stack = new StackL();

    for(int i = 0; i < 10; i++)
        stack.push(Collections2.countries.next());

    System.out.println(stack.top());
    System.out.println(stack.top());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
}
} ///:~

```

L'héritage n'est pas approprié ici puisqu'il produirait une classe contenant toutes les méthodes d'une **LinkedList** (on verra que cette erreur a déjà été faite par les concepteurs de la bibliothèque Java 1.0 avec la classe **Stack**).

Réaliser une file à partir d'une LinkedList

Une file (ou queue) est un conteneur « premier arrivé, premier sorti » (FIFO - « First In, First Out »). C'est à dire qu'on « pousse » des objets à une extrémité et qu'on les en retire à l'autre extrémité. L'ordre dans lequel on pousse les objets sera donc le même que l'ordre dans lequel on les récupérera. La classe **LinkedList** possède des méthodes qui implémentent directement les fonctionnalités d'une file, on peut donc les utiliser directement dans une classe **Queue** :

```

//: c09:Queue.java

// Réaliser une file à partir d'une LinkedList.

import java.util.*;

public class Queue {
    private LinkedList list = new LinkedList();

    public void put(Object v) { list.addFirst(v); }

    public Object get() {
        return list.removeLast();
    }

    public boolean isEmpty() {

```



```

        return list.isEmpty();
    }

    public static void main(String[] args) {
        Queue queue = new Queue();

        for(int i = 0; i < 10; i++)
            queue.put(Integer.toString(i));

        while(!queue.isEmpty())
            System.out.println(queue.get());
    }
} ///:~

```

Il est aussi facile de créer une file double (queue à double entrée) à partir d'une **LinkedList**. Une file double est une file à laquelle on peut ajouter et supprimer des éléments à chacune de ses extrémités.

Fonctionnalités des Sets

Les **Sets** ont exactement la même interface que les **Collections**, et à l'inverse des deux différentes **Lists**, ils ne proposent aucune fonctionnalité supplémentaire. Les **Sets** sont donc juste une **Collection** ayant un comportement particulier (implémenter un comportement différent constitue l'exemple type où il faut utiliser l'héritage et le polymorphisme). Un **Set** refuse de contenir plus d'une instance de chaque valeur d'un objet (savoir ce qu'est la « valeur » d'un objet est plus compliqué, comme nous allons le voir).

Set (interface)	Chaque élément ajouté au Set doit être unique ; sinon le Set n'ajoutera pas le doublon. Les Objects ajoutés à un Set doivent définir la méthode equals() pour pouvoir établir l'unicité de l'objet. Un Set possède la même interface qu'une Collection . L'interface Set ne garantit pas qu'il maintiendra les éléments dans un ordre particulier.
HashSet *	Pour les Sets où le temps d'accès aux éléments est primordial. Les Objects doivent définir la méthode hashCode() .
TreeSet	Un Set trié stocké dans un arbre. De cette manière, on peut extraire une séquence triée à partir du Set .

L'exemple suivant ne montre *pas* tout ce qu'il est possible de faire avec un **Set**, puisque l'interface est la même que pour les **Collections**, et comme telle a déjà été testée dans l'exemple précédent. Par contre, il illustre les comportements qui rendent un **Set** particulier :

```

///: c09:Set1.java

// Opérations disponibles pour les Sets.

import java.util.*;

import com.bruceeckel.util.*;

public class Set1 {

```

```

static Collections2.StringGenerator gen =
    Collections2.countries;

public static void testVisual(Set a) {
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    Collections2.fill(a, gen.reset(), 10);
    System.out.println(a); // Pas de doublons !
    // Ajoute un autre ensemble à celui-ci :
    a.addAll(a);
    a.add("one");
    a.add("one");
    a.add("one");
    System.out.println(a);
    // Extraction d'item :
    System.out.println("a.contains(\"one\") : " +
        a.contains("one"));
}

public static void main(String[] args) {
    System.out.println("HashSet");
    testVisual(new HashSet());
    System.out.println("TreeSet");
    testVisual(new TreeSet());
}
} ///:~

```

Cet exemple tente d'ajouter des valeurs dupliquées au **Set**, mais lorsqu'on l'imprime on voit que le **Set** n'accepte qu'une instance de chaque valeur.

Lorsqu'on lance ce programme, on voit que l'ordre interne maintenu par le **HashSet** est différent de celui de **TreeSet**, puisque chacune de ces implémentations stocke les éléments d'une manière différente (**TreeSet** garde les éléments triés, tandis que **HashSet** utilise une fonction de hachage, conçue spécialement pour des accès optimisés). Quand on crée un nouveau type, il faut bien se rappeler qu'un **Set** a besoin de maintenir un ordre de stockage, ce qui veut dire qu'il faut implémenter l'interface **Comparable** et définir la méthode **compareTo()**. Voici un exemple :

```

//: c09:Set2.java

// Ajout d'un type particulier dans un Set.

```

```
import java.util.*;

class MyType implements Comparable {
    private int i;

    public MyType(int n) { i = n; }

    public boolean equals(Object o) {
        return
            (o instanceof MyType)
            && (i == ((MyType)o).i);
    }

    public int hashCode() { return i; }

    public String toString() { return i + " "; }

    public int compareTo(Object o) {
        int i2 = ((MyType)o).i;
        return (i2 < i ? -1 : (i2 == i ? 0 : 1));
    }
}

public class Set2 {
    public static Set fill(Set a, int size) {
        for(int i = 0; i < size; i++)
            a.add(new MyType(i));
        return a;
    }

    public static void test(Set a) {
        fill(a, 10);
        fill(a, 10); // Tente de créer des doublons
        fill(a, 10);
        a.addAll(fill(new TreeSet(), 10));
        System.out.println(a);
    }

    public static void main(String[] args) {
        test(new HashSet());
        test(new TreeSet());
    }
}
```

```

    }
} ///:~

```

La forme que doivent avoir les définitions des méthodes **equals()** et **hashCode()** sera décrite plus tard dans ce chapitre. Il faut définir une méthode **equals()** pour les deux implémentations de **Set**, mais **hashCode()** n'est nécessaire que si la classe est placée dans un **HashSet** (ce qui est probable, puisqu'il s'agit de l'implémentation recommandée pour un **Set**). Cependant, c'est une bonne pratique de programmation de redéfinir **hashCode()** lorsqu'on redéfinit **equals()**. Ce processus sera examiné en détails plus loin dans ce chapitre.

Notez que je n'ai *pas* utilisé la forme « simple et évidente » **return i-i2** dans la méthode **compareTo()**. Bien que ce soit une erreur de programmation classique, elle ne fonctionne que si **i** et **i2** sont des **ints** « non signés » (si Java *disposait* d'un mot-clef « **unsigned** », ce qu'il n'a pas). Elle ne marche pas pour les **ints** signés de Java, qui ne sont pas assez grands pour représenter la différence de deux **ints** signés. Si **i** est un grand entier positif et **j** un grand entier négatif, **i-j** débordera et renverra une valeur négative, ce qui n'est pas le résultat attendu.

Sets triés : les SortedSets

Un **SortedSet** (dont **TreeSet** est l'unique représentant) garantit que ses éléments seront stockés triés, ce qui permet de proposer de nouvelles fonctionnalités grâce aux méthodes supplémentaires de l'interface **SortedSet** suivantes :

Comparator comparator() : Renvoie le **Comparator** utilisé pour ce **Set**, ou **null** dans le cas d'un tri naturel.

Object first() : Renvoie le plus petit élément.

Object last() : Renvoie le plus grand élément.

SortedSet subSet(fromElement, toElement) : Renvoie une vue du **Set** contenant les éléments allant de **fromElement** inclus à **toElement** exclu.

SortedSet headSet(toElement) : Renvoie une vue du **Set** contenant les éléments inférieurs à **toElement**.

SortedSet tailSet(fromElement) : Renvoie une vue du **Set** contenant les éléments supérieurs ou égaux à **fromElement**.

Fonctionnalités des Maps

Une **ArrayList** permet de sélectionner des éléments dans une séquence d'objets en utilisant un nombre, elle associe donc des nombres à des objets. Mais qu'en est-il si on souhaite sélectionner des éléments d'une séquence en utilisant un autre critère ? Dans l'exemple d'une pile, son critère de sélection est « le dernier objet poussé sur la pile ». Un *tableau associatif*, ou *map*, ou *dictionnaire* est une alternative particulièrement puissante de cette idée de « sélection dans une séquence ». Conceptuellement, cela ressemble à une **ArrayList**, mais au lieu de sélectionner un objet par un nombre, on le sélectionne en utilisant un *autre objet* ! Ce fonctionnement est d'une valeur inestimable dans un programme.

Le concept est illustré dans Java via l'interface **Map**. La méthode **put(Object key, Object value)** ajoute une valeur (la chose qu'on veut stocker), et l'associe à une clef (la chose grâce à laquelle on va retrouver la valeur). La méthode **get(Object key)** renvoie la valeur associée à la clef correspondante. Il est aussi possible de tester une **Map** pour voir si elle contient une certaine clef ou une certaine valeur avec les méthodes **containsKey()** et **containsValue()**.

La bibliothèque Java standard propose deux types de **Maps** : **HashMap** et **TreeMap**. Les deux implémentations ont la même interface (puisqu'elles implémentent toutes les deux **Map**), mais diffèrent sur un point particulier : les performances. Dans le cas d'un appel à **get()**, il est peu efficace de chercher dans une **ArrayList** (par exemple) pour trouver une clef. C'est là que le **HashMap** intervient. Au lieu d'effectuer une recherche lente sur la clef, il utilise une valeur spéciale appelée *code de hachage* (*hash code*). Le code de hachage est une façon d'extraire une partie de l'information de l'objet en question et de la convertir en un **int** « relativement unique ». Tous les objets Java peuvent produire un code de hachage, et **hashCode()** est une méthode de la classe racine **Object**. Un **HashMap** récupère le **hashCode()** de l'objet et l'utilise pour retrouver rapidement la clef. Le résultat en est une augmentation drastique des performances [\[50\]](#).

Map (interface)	Maintient des associations clef - valeur (des paires), afin de pouvoir accéder à une valeur en utilisant une clef.
HashMap *	Implémentation basée sur une table de hachage (utilisez ceci à la place d'une Hashtable). Fournit des performances constantes pour l'insertion et l'extraction de paires. Les performances peuvent être ajustées via des constructeurs qui permettent de positionner la <i>capacité</i> et le <i>facteur de charge</i> de la table de hachage.
TreeMap	Implémentation basée sur un arbre rouge-noir. L'extraction des clefs ou des paires fournit une séquence triée (selon l'ordre spécifié par Comparable ou Comparator , comme nous le verrons plus loin). Le point important dans un TreeMap est qu'on récupère les résultats dans l'ordre. TreeMap est la seule Map disposant de la méthode subMap() , qui permet de renvoyer une portion de l'arbre.

Nous nous pencherons sur les mécanismes de hachage un peu plus loin. L'exemple suivant utilise la méthode **Collections2.fill()** et les ensembles de données définis précédemment :

```

//: c09:Map1.java

// Opérations disponibles pour les Maps.

import java.util.*;

import com.bruceeckel.util.*;

public class Map1 {

    static Collections2.StringPairGenerator geo =
        Collections2.geography;

    static Collections2.RandStringPairGenerator
        rsp = Collections2.rsp;

    // Produire un Set de clefs :

    public static void printKeys(Map m) {

        System.out.print("Size = " + m.size() + ", ");

        System.out.print("Keys: ");

        System.out.println(m.keySet());

    }

```

```
// Produire une Collection de valeurs :
public static void printValues(Map m) {
    System.out.print("Values: ");
    System.out.println(m.values());
}

public static void test(Map m) {
    Collections2.fill(m, geo, 25);
    // Une Map a un comportement de « Set » pour les clefs :
    Collections2.fill(m, geo.reset(), 25);
    printKeys(m);
    printValues(m);
    System.out.println(m);

    String key = CountryCapitals.pairs[4][0];
    String value = CountryCapitals.pairs[4][1];
    System.out.println("m.containsKey(\"" + key +
        "\"): " + m.containsKey(key));
    System.out.println("m.get(\"" + key + "\"): "
        + m.get(key));
    System.out.println("m.containsValue(\""
        + value + "\"): " +
        m.containsValue(value));
    Map m2 = new TreeMap();
    Collections2.fill(m2, rsp, 25);
    m.putAll(m2);
    printKeys(m);
    key = m.keySet().iterator().next().toString();
    System.out.println("First key in map: "+key);
    m.remove(key);
    printKeys(m);
    m.clear();
    System.out.println("m.isEmpty(): "
        + m.isEmpty());
    Collections2.fill(m, geo.reset(), 25);
    // Les opérations sur le Set changent la Map :
```

```

        m.keySet().removeAll(m.keySet());

        System.out.println("m.isEmpty(): "

            + m.isEmpty());
    }

    public static void main(String[] args) {

        System.out.println("Testing HashMap");

        test(new HashMap());

        System.out.println("Testing TreeMap");

        test(new TreeMap());

    }

} ///:~

```

Les méthodes **printKeys()** et **printValues()** ne sont pas seulement des utilitaires pratiques, elles illustrent aussi comment produire une vue (sous la forme d'une **Collection**) d'une **Map**. La méthode **keySet()** renvoie un **Set** rempli par les clefs de la **Map**. La méthode **values()** renvoie quant à elle une **Collection** contenant toutes les valeurs de la **Map** (notez bien que les clefs doivent être uniques, alors que les valeurs peuvent contenir des doublons). Ces **Collections** sont liées à la **Map**, tout changement effectué dans une **Collection** sera donc répercuté dans la **Map** associée.

Le reste du programme fournit des exemples simples pour chacune des opérations disponibles pour une **Map**, et teste les deux types de **Maps**.

Comme exemple d'utilisation d'un **HashMap**, considérons un programme vérifiant la nature aléatoire de la méthode **Math.random()** de Java. Idéalement, elle devrait produire une distribution parfaite de nombres aléatoires, mais pour tester cela il faut générer un ensemble de nombres aléatoires et compter ceux qui tombent dans les différentes plages. Un **HashMap** est parfait pour ce genre d'opérations, puisqu'il associe des objets à d'autres objets (dans ce cas, les objets valeurs contiennent le nombre produit par **Math.random()** ainsi que le nombre de fois où ce nombre apparaît) :

```

///: c09:Statistics.java

// Simple démonstration de l'utilisation d'un HashMap.

import java.util.*;

class Counter {

    int i = 1;

    public String toString() {

        return Integer.toString(i);

    }

}

```

```

class Statistics {
    public static void main(String[] args) {
        HashMap hm = new HashMap();

        for(int i = 0; i < 10000; i++) {
            // Produit un nombre entre 0 et 20 :

            Integer r =

                new Integer((int)(Math.random() * 20));

            if(hm.containsKey(r))

                ((Counter)hm.get(r)).i++;

            else

                hm.put(r, new Counter());

        }

        System.out.println(hm);

    }
} ///:~

```

Dans **main()**, chaque fois qu'un nombre aléatoire est généré, il est encapsulé dans un objet **Integer** dont la référence sera utilisée par le **HashMap** (on ne peut stocker de scalaire dans un conteneur, uniquement une référence sur un objet). La méthode **containsKey()** vérifie si la clef existe dans le conteneur (c'est à dire, est-ce que le nombre a déjà été généré auparavant ?). Si c'est le cas, la méthode **get()** renvoie la valeur associée à cette clef, un objet **Counter** dans notre cas. La valeur **i** à l'intérieur du compteur est incrémentée pour indiquer qu'une occurrence de plus de ce nombre aléatoire particulier a été généré.

Si la clef n'a pas déjà été stockée, la méthode **put()** insèrera une nouvelle paire clef - valeur dans le **HashMap**. Puisque **Counter** initialise automatiquement sa variable **i** à 1 lorsqu'elle est créée, cela indique une première occurrence de ce nombre aléatoire particulier.

Pour afficher le **HashMap**, il est simplement imprimé. La méthode **toString()** de **HashMap** parcourt toutes les paires clef - valeur et appelle **toString()** pour chacune d'entre elles. La méthode **Integer.toString()** est prédéfinie, et on peut voir la méthode **toString()** de la classe **Counter**. La sortie du programme (après l'insertion de quelques retours chariot) ressemble à :

```

{19=526, 18=533, 17=460, 16=513, 15=521, 14=495,
 13=512, 12=483, 11=488, 10=487, 9=514, 8=523,
 7=497, 6=487, 5=480, 4=489, 3=509, 2=503, 1=475,
 0=505}

```

On peut s'interroger sur la nécessité d'une classe **Counter**, qui ne semble même pas avoir les fonctionnalités de la classe d'encapsulation **Integer**. Pourquoi ne pas utiliser un **int** ou un **Integer** ? On ne peut utiliser un **int** puisque les conteneurs ne peuvent stocker que des références d'**Object**. On pourrait alors être tenté de se tourner

vers les classes Java d'encapsulation des types primitifs. Cependant, ces classes ne permettent que de stocker une valeur initiale et de lire cette valeur. C'est à dire qu'il n'existe aucun moyen de changer cette valeur une fois qu'un objet d'encapsulation a été créé. Cela rend la classe **Integer** inutile pour résoudre notre problème, et nous force à créer une nouvelle classe qui satisfait l'ensemble de nos besoins.

Maps triées : les SortedMaps

Une **SortedMap** (dont **TreeMap** est l'unique représentant) garantit que ses éléments seront stockés triés selon leur clef, ce qui permet de proposer de nouvelles fonctionnalités grâce aux méthodes supplémentaires de l'interface **SortedMap** suivantes :

Comparator comparator() : Renvoie le **Comparator** utilisé pour cette **Map**, ou **null** dans le cas d'un tri naturel.

Object firstKey() : Renvoie la plus petite clef.

Object lastKey() : Renvoie la plus grande clef.

SortedMap subMap(fromKey, toKey) : Renvoie une vue de la **Map** contenant les paires dont les clefs vont de **fromKey** inclus à **toKey** exclu.

SortedMap headMap(toKey) : Renvoie une vue de la **Map** contenant les paires dont la clef est inférieure à **toKey**.

SortedMap tailMap(fromKey) : Renvoie une vue de la **Map** contenant les paires dont la clef est supérieure ou égale à **fromKey**.

Hachage et codes de hachage

Dans l'exemple précédent, une classe de la bibliothèque standard (**Integer**) était utilisée comme clef pour le **HashMap**. Cela ne pose pas de problèmes car elle dispose de tout ce qu'il faut pour fonctionner correctement comme une clef. Mais il existe un point d'achoppement classique avec les **HashMaps** lorsqu'on crée une classe destinée à être utilisée comme clef. Considérons par exemple un système de prévision météorologique qui associe des objets **Groundhog** à des objets **Prediction**. Cela semble relativement simple : il suffit de créer deux classes, et d'utiliser **Groundhog** comme clef et **Prediction** comme valeur :

```
///  
// Semble plausible, mais ne fonctionne pas.  
  
import java.util.*;  
  
class Groundhog {  
    int ghNumber;  
  
    Groundhog(int n) { ghNumber = n; }  
}  
  
class Prediction {
```

```

    boolean shadow = Math.random() > 0.5;

    public String toString() {
        if(shadow)
            return "Six more weeks of Winter!";
        else
            return "Early Spring!";
    }
}

public class SpringDetector {
    public static void main(String[] args) {
        HashMap hm = new HashMap();

        for(int i = 0; i < 10; i++)
            hm.put(new Groundhog(i), new Prediction());

        System.out.println("hm = " + hm + "\n");

        System.out.println(
            "Looking up prediction for Groundhog #3:");

        Groundhog gh = new Groundhog(3);

        if(hm.containsKey(gh))
            System.out.println((Prediction)hm.get(gh));
        else
            System.out.println("Key not found: " + gh);
    }
} ///:~

```

Chaque **Groundhog** se voit attribuer un numéro d'identité, afin de pouvoir récupérer une **Prediction** dans le **HashMap** en disant « Donne-moi la **Prediction** associée au **Groundhog** numéro 3 ». La classe **Prediction** contient un **boolean** initialisé en utilisant **Math.random()**, et une méthode **toString()** pour en interpréter la valeur. Dans **main()**, un **HashMap** est rempli avec des **Groundhogs** et leurs **Predictions** associées. Le **HashMap** est affiché afin de voir qu'il a été correctement rempli. Un **Groundhog** avec une identité de 3 est alors utilisé comme clef pour extraire la prédiction du **Groundhog** numéro 3 (qui doit être dans le **HashMap**).

Tout ceci semble très simple, mais ne marche pas. Le problème vient du fait que **Groundhog** hérite de la classe de base **Object** (ce qui est le comportement par défaut si aucune superclasse n'est précisée ; toutes les classes dérivent donc en fin de compte de la classe **Object**). C'est donc la méthode **hashCode()** de **Object** qui est utilisée pour générer le code de hachage pour chaque objet, et par défaut, celle-ci renvoie juste l'adresse de cet objet. La première instance de **Groundhog(3)** ne renvoie donc *pas* le même code de hachage que celui de la seconde instance de **Groundhog(3)** que nous avons tenté d'utiliser comme clef d'extraction.

On pourrait penser qu'il suffit de redéfinir **hashCode()**. Mais ceci ne fonctionnera toujours pas tant qu'on n'aura pas aussi redéfini la méthode **equals()** qui fait aussi partie de la classe **Object**. Cette méthode est utilisée par le **HashMap** lorsqu'il essaie de déterminer si la clef est égale à l'une des autres clefs de la table. Et la méthode par défaut **Object.equals()** compare simplement les adresses des objets, ce qui fait qu'un objet **Groundhog(3)** est différent d'un autre **Groundhog(3)**.

Pour utiliser un nouveau type comme clef dans un **HashMap**, il faut donc redéfinir les deux méthodes **hashCode()** et **equals()**, comme le montre la solution suivante :

```
//: c09:SpringDetector2.java
// Une classe utilisée comme clef dans un HashMap
// doit redéfinir hashCode() et equals().

import java.util.*;

class Groundhog2 {
    int ghNumber;

    Groundhog2(int n) { ghNumber = n; }

    public int hashCode() { return ghNumber; }

    public boolean equals(Object o) {
        return (o instanceof Groundhog2)
            && (ghNumber == ((Groundhog2)o).ghNumber);
    }
}

public class SpringDetector2 {
    public static void main(String[] args) {
        HashMap hm = new HashMap();

        for(int i = 0; i < 10; i++)
            hm.put(new Groundhog2(i), new Prediction());

        System.out.println("hm = " + hm + "\n");

        System.out.println(
            "Looking up prediction for groundhog #3:");

        Groundhog2 gh = new Groundhog2(3);

        if(hm.containsKey(gh))
            System.out.println((Prediction)hm.get(gh));
    }
}
```

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

Notez que cet exemple utilise la classe **Prediction** de l'exemple précédent, donc **SpringDetector.java** doit déjà avoir été compilé ou vous aurez une erreur lorsque vous tenterez de compiler **SpringDetector2.java**.

Groundhog2.hashCode() renvoie le numéro de marmotte comme identifiant. Dans cet exemple, le programmeur doit s'assurer que deux marmottes ne portent pas le même identifiant. La méthode **hashCode()** n'est pas obligée de renvoyer un identifiant unique (vous comprendrez mieux ceci plus tard dans ce chapitre), mais la méthode **equals()** doit être capable de déterminer si deux objets sont strictement équivalents.

Bien que la méthode **equals()** semble ne vérifier que si l'argument est bien une instance de **Groundhog2** (en utilisant le mot-clef **instanceof**, expliqué plus en détails dans le Chapitre 12), **instanceof** effectue en fait implicitement un deuxième contrôle puisqu'il renvoie **false** si l'argument de gauche est **null**. En assumant que le contrôle de type s'est bien passé, la comparaison est basée sur les **ghNumbers** des instances. Et cette fois, lorsqu'on lance le programme, on peut voir qu'il produit le résultat attendu.

On rencontre les mêmes problèmes quand on crée un nouveau type destiné à être stocké dans un **HashSet** ou utilisé comme clef dans un **HashMap**.

Comprendre hashCode()

L'exemple précédent n'est que le début de la solution complète et correcte de ce problème. Il montre que la structure de données hachée (**HashSet** ou **HashMap**) ne sera pas capable de gérer correctement les objets clef si on ne redéfinit pas les méthodes **hashCode()** et **equals()** pour ces objets. Cependant, pour fournir une solution propre au problème, il faut comprendre ce qui se passe derrière la structure de données hachée.

Pour cela, il faut tout d'abord comprendre le pourquoi du hachage : on veut extraire un objet associé à un autre objet. Mais il est aussi possible d'accomplir ceci avec un **TreeSet** ou un **TreeMap**. Il est même possible d'implémenter sa propre **Map**. Pour cela, il nous faut fournir une méthode **Map.entrySet()** qui renvoie un ensemble d'objets **Map.Entry**. **MPair** sera définie comme le nouveau type de **Map.Entry**. Afin qu'il puisse être placé dans un **TreeSet** il doit implémenter **equals()** et être **Comparable** :

```
///  
// Une Map implémentée avec des ArrayLists.  
import java.util.*;  
  
public class MPair  
implements Map.Entry, Comparable {  
    Object key, value;  
    MPair(Object k, Object v) {  
        key = k;  
        value = v;  
    }  
    public Object getKey() { return key; }
```

```

    public Object getValue() { return value; }

    public Object setValue(Object v){
        Object result = value;
        value = v;
        return result;
    }

    public boolean equals(Object o) {
        return key.equals(((MPair)o).key);
    }

    public int compareTo(Object rv) {
        return ((Comparable)key).compareTo(
            ((MPair)rv).key);
    }
} ///:~

```

Notez que les comparaisons ne s'effectuent que sur les clefs, les valeurs dupliquées sont donc parfaitement légales.

L'exemple suivant implémente une **Map** en utilisant une paire d'**ArrayLists** :

```

///: c09:SlowMap.java
// Une Map implémentée avec des ArrayLists.

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

public class SlowMap extends AbstractMap {
    private ArrayList
        keys = new ArrayList(),
        values = new ArrayList();

    public Object put(Object key, Object value) {
        Object result = get(key);
        if(!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else
            values.set(keys.indexOf(key), value);
    }
}

```

```

        return result;
    }

    public Object get(Object key) {
        if(!keys.contains(key))
            return null;
        return values.get(keys.indexOf(key));
    }

    public Set entrySet() {
        Set entries = new HashSet();
        Iterator
            ki = keys.iterator(),
            vi = values.iterator();
        while(ki.hasNext())
            entries.add(new MPair(ki.next(), vi.next()));
        return entries;
    }

    public static void main(String[] args) {
        SlowMap m = new SlowMap();
        Collections2.fill(m,
            Collections2.geography, 25);
        System.out.println(m);
    }
} ///:~

```

La méthode **put()** stocke simplement les clefs et les valeurs dans les **ArrayLists** correspondantes. Dans **main()**, une **SlowMap** est remplie et imprimée pour montrer qu'elle fonctionne.

Ceci montre qu'il n'est pas difficile de produire un nouveau type de **Map**. Mais comme son nom le suggère, une **SlowMap** n'est pas très rapide, et on ne l'utilisera probablement pas si on dispose d'une autre alternative. Le problème se trouve dans la recherche de la clef : comme elles sont stockées sans aucun ordre, une recherche linéaire est effectuée, ce qui constitue la manière la plus lente de rechercher un item particulier.

Tout l'intérêt du hachage réside dans la vitesse : le hachage permet d'effectuer la recherche rapidement. Puisque le goulot d'étranglement est la recherche de la clef, une des solutions du problème serait de garder les clefs triées et d'utiliser ensuite **Collections.binarySearch()** pour réaliser la recherche (un exercice à la fin du chapitre vous mènera le long de ce processus).

Le hachage va encore plus loin en spécifiant que tout ce qu'on a besoin de faire est de stocker la clef *quelque part* afin de pouvoir la retrouver rapidement. Comme on l'a déjà vu dans ce chapitre, la structure la plus efficace

pour stocker un ensemble d'éléments est un tableau, c'est donc ce que nous utiliserons pour stocker les informations des clefs (notez bien que j'ai dit : « information des clefs » et non les clefs elles-mêmes). Nous avons aussi vu dans ce chapitre qu'un tableau, une fois alloué, ne peut être redimensionné, nous nous heurtons donc à un autre problème : nous voulons être capable de stocker un nombre quelconque de valeurs dans la **Map**, mais comment cela est-ce possible si le nombre de clefs est fixé par la taille du tableau ?

Tout simplement, le tableau n'est pas destiné à stocker les clefs. Un nombre dérivé de l'objet clef servira d'index dans le tableau. Ce nombre est le *code de hachage*, renvoyé par la méthode **hashCode()** (dans le jargon informatique, on parle de *fonction de hachage*) définie dans la classe **Object** et éventuellement redéfinie dans un nouveau type. Pour résoudre le problème du tableau de taille fixe, plus d'une clef peut produire le même index ; autrement dit, les *collisions* sont autorisées. Et de ce fait, la taille du tableau importe peu puisque chaque objet clef atterrira quelque part dans ce tableau.

Le processus de recherche d'une valeur débute donc par le calcul du code de hachage, qu'on utilise pour indexer le tableau. Si on peut garantir qu'il n'y a pas eu de collisions (ce qui est possible si on a un nombre fixé de valeurs), alors on dispose d'une *fonction de hachage parfaite*, mais il s'agit d'un cas spécial. Dans les autres cas, les collisions sont gérées par un *chaînage externe* : le tableau ne pointe pas directement sur une valeur, mais sur une liste de valeurs. Ces valeurs sont alors parcourues de façon linéaire en utilisant la méthode **equals()**. Bien sûr, cet aspect de la recherche est plus lent, mais si la fonction de hachage est correctement écrite, il n'y aura que quelques valeurs au plus dans chaque emplacement. Et donc au lieu de parcourir toute la liste pour trouver une valeur, on saute directement dans une cellule où seules quelques entrées devront être comparées pour trouver la valeur. Cette approche est bien plus efficace, ce qui explique pourquoi un **HashMap** est si rapide.

Connaissant les bases du hachage, il est possible d'implémenter une **Map** simple hachée :

```
//: c09:SimpleHashMap.java
// Démonstration d'une Map hachée.

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

public class SimpleHashMap extends AbstractMap {
    // Choisir un nombre premier pour la taille de la table
    // de hachage, afin d'obtenir une distribution uniforme :
    private final static int SZ = 997;
    private LinkedList[] bucket= new LinkedList[SZ];
    public Object put(Object key, Object value) {
        Object result = null;
        int index = key.hashCode() % SZ;
        if(index < 0) index = -index;
        if(bucket[index] == null)
            bucket[index] = new LinkedList();
        LinkedList pairs = bucket[index];
```

```
MPair pair = new MPair(key, value);

ListIterator it = pairs.listIterator();

boolean found = false;
while(it.hasNext()) {
    Object iPair = it.next();

    if(iPair.equals(pair)) {
        result = ((MPair)iPair).getValue();

        it.set(pair); // Remplace l'ancien par le nouveau

        found = true;

        break;
    }
}

if(!found)
    bucket[index].add(pair);

return result;
}

public Object get(Object key) {
    int index = key.hashCode() % SZ;

    if(index < 0) index = -index;

    if(bucket[index] == null) return null;

    LinkedList pairs = bucket[index];

    MPair match = new MPair(key, null);

    ListIterator it = pairs.listIterator();

    while(it.hasNext()) {
        Object iPair = it.next();

        if(iPair.equals(match))
            return ((MPair)iPair).getValue();
    }

    return null;
}

public Set entrySet() {
    Set entries = new HashSet();

    for(int i = 0; i < bucket.length; i++) {
        if(bucket[i] == null) continue;
    }
}
```



```

        Iterator it = bucket[i].iterator();

        while(it.hasNext())

            entries.add(it.next());

    }

    return entries;

}

public static void main(String[] args) {

    SimpleHashMap m = new SimpleHashMap();

    Collections2.fill(m,

        Collections2.geography, 25);

    System.out.println(m);

}

} ///:~

```

Comme on appelle souvent *seaux* les « emplacements » d'une table de hachage, le tableau représentant la table est appelé **bucket**. Pour s'assurer d'une distribution la plus régulière possible, le nombre de seaux est typiquement un nombre premier. Notez qu'il s'agit d'un tableau de **LinkedLists**, qui permet de gérer automatiquement les collisions - chaque nouvel item est simplement ajouté à la fin de la liste.

La valeur de retour de **put()** est **null** ou l'ancienne valeur associée à la clef si la celle-ci était présente dans la liste. La valeur de retour est **result**, qui est initialisée à **null**, mais se voit assigner une clef si celle-ci est découverte dans la liste.

Les deux méthodes **put()** et **get()** commencent par appeler la méthode **hashCode()** de l'objet clef, dont le résultat est forcé à un nombre positif. Il est alors forcé dans la plage du tableau via l'opérateur modulo et la taille du tableau. Si l'emplacement est **null**, cela veut dire qu'aucun élément ne hache à cette localisation, et donc une nouvelle **LinkedList** est créée pour contenir l'objet qui vient de le faire. Sinon, le processus normal est de parcourir la liste pour voir s'il existe un doublon, et si c'est le cas, l'ancienne valeur est stockée dans **result** et la nouvelle valeur remplace l'ancienne. Le flag **found** permet de savoir si une ancienne paire clef - valeur a été trouvée, et dans le cas contraire, une nouvelle paire est ajoutée à la fin de la liste.

Le code de **get()** est similaire à celui de **put()**, en plus simple. L'index dans le tableau **bucket** est calculé, et si une **LinkedList** existe elle est parcourue pour trouver une concordance.

entrySet() doit trouver et parcourir toutes les listes, ajoutant tous les éléments dans le **Set** résultat. Une fois cette méthode fournie, la **Map** peut être testée en la remplissant avec des valeurs et en les imprimant.

Facteurs de performance d'un HashMap

Voici tout d'abord une terminologie nécessaire pour comprendre les mécanismes mis en jeu :

Capacité : Le nombre de seaux dans la table.

Capacité initiale : Le nombre de seaux dans la table quand celle-ci est créée. Les **HashMap** et les **HashSet** proposent des constructeurs qui permettent de spécifier la capacité initiale.

Taille : Le nombre courant d'entrées dans la table.

Facteur de charge : taille/capacité. Un facteur de charge de 0 correspond à une table vide, 0.5 correspond à une table à moitié pleine, etc. Une table faiblement chargée aura peu de collisions et sera donc optimale pour les insertions et les recherches (mais ralentira le processus de parcours avec un itérateur). **HashMap** et **HashSet** proposent des constructeurs qui permettent de spécifier un facteur de charge, ce qui veut dire que lorsque ce facteur de charge est atteint le conteneur augmentera automatiquement sa capacité (le nombre de seaux) en la doublant d'un coup, et redistribuera les objets existants dans le nouvel ensemble de seaux (c'est ce qu'on appelle le *rehachage*).

Le facteur de charge par défaut utilisé par **HashMap** est 0.75 (il ne se rehache pas avant que la table ne soit aux $\frac{3}{4}$ pleine). Cette valeur est un bon compromis entre les performances et le coût en espace. Un facteur de charge plus élevé réduit l'espace requis par une table mais augmente le coût d'une recherche, ce qui est important parce que les recherches sont les opérations les plus courantes (incluant les appels **get()** et **put()**).

Si un **HashMap** est destiné à recevoir beaucoup d'entrées, le créer avec une grosse capacité initiale permettra d'éviter le surcoût du rehachage automatique.

Redéfinir hashCode()

Maintenant que nous avons vu les processus impliqués dans le fonctionnement d'un **HashMap**, les problèmes rencontrés dans l'écriture d'une méthode **hashCode()** prennent tout leur sens.

Tout d'abord, on ne contrôle pas la valeur réellement utilisée pour indexer le seau dans le tableau. Celle-ci est dépendante de la capacité de l'objet **HashMap**, et cette capacité change suivant la taille et la charge du conteneur. La valeur renvoyée par la méthode **hashCode()** est simplement utilisée pour calculer l'index du seau (dans **SimpleHashMap** le calcul se résume à un modulo de la taille du tableau de seaux).

Le facteur le plus important lors de la création d'une méthode **hashCode()** est qu'elle doit toujours renvoyer la même valeur pour un objet particulier, quel que soit le moment où **hashCode()** est appelée. Si on a un objet dont la méthode **hashCode()** renvoie une valeur lors d'un **put()** dans un **HashMap**, et une autre durant un appel à **get()**, on sera incapable de retrouver cet objet. Si la méthode **hashCode()** s'appuie sur des données modifiables dans l'objet, l'utilisateur doit alors être prévenu que changer ces données produira une clef différente en générant un code de hachage différent.

De plus, on ne veut pas *non plus* générer un code de hachage qui soit basé uniquement sur des informations uniques spécifiques à l'instance de l'objet - en particulier, la valeur de **this** est une mauvaise idée pour un code de hachage, puisqu'on ne peut générer une nouvelle clef identique à celle utilisée pour stocker la paire originale clef-valeur. C'est le problème que nous avons rencontré dans **SpringDetector.java** parce que l'implémentation par défaut de **hashCode()** utilise l'adresse de l'objet. Il faut donc utiliser des informations de l'objet qui identifient l'objet d'une façon sensée.

Un exemple en est trouvé dans la classe **String**. Les **Strings** ont cette caractéristique spéciale : si un programme utilise plusieurs objets **String** contenant la même séquence de caractères, alors ces objets **String** pointent tous vers la même zone de mémoire (ce mécanisme est décrit dans l'annexe A). Il semble donc sensé que le code de hachage produit par deux instances distinctes de **new String("hello")** soit identique. On peut le vérifier avec ce petit programme :

```
//: c09:StringHashCode.java
```

```

public class StringHashCode {

    public static void main(String[] args) {

        System.out.println("Hello".hashCode());

        System.out.println("Hello".hashCode());

    }

} ///:~

```

Pour que ceci fonctionne, le code de hachage de **String** doit être basé sur le contenu de la **String**.

Pour qu'un code de hachage soit efficace, il faut donc qu'il soit rapide et chargé de sens : c'est donc une valeur basée sur le contenu de l'objet. Rappelons que cette valeur n'a pas à être unique - mieux vaut se pencher sur la vitesse que sur l'unicité - mais l'identité d'un objet doit être complètement résolue entre **hashCode()** et **equals()**.

Parce qu'un code de hachage est traité avant de produire un index de seau, la plage de valeurs n'est pas importante ; il suffit de générer un **int**.

Enfin, il existe un autre facteur : une méthode **hashCode()** bien conçue doit renvoyer des valeurs bien distribuées. Si les valeurs tendent à se regrouper, alors les **HashMaps** et les **HashSets** seront plus chargés dans certaines parties et donc moins rapides que ce qu'ils pourraient être avec une fonction de hachage mieux répartie.

Voici un exemple qui respecte ces règles de base :

```

///: c09:CountedString.java

// Créer une bonne méthode hashCode().

import java.util.*;

public class CountedString {

    private String s;

    private int id = 0;

    private static ArrayList created =

        new ArrayList();

    public CountedString(String str) {

        s = str;

        created.add(s);

        Iterator it = created.iterator();

        // id est le nombre total d'instances de cette

        // chaîne utilisées par CountedString :

        while(it.hasNext())

            if(it.next().equals(s))

```

```

        id++;
    }

    public String toString() {
        return "String: " + s + " id: " + id +
            " hashCode(): " + hashCode() + "\n";
    }

    public int hashCode() {
        return s.hashCode() * id;
    }

    public boolean equals(Object o) {
        return (o instanceof CountedString)
            && s.equals(((CountedString)o).s)
            && id == ((CountedString)o).id;
    }

    public static void main(String[] args) {
        HashMap m = new HashMap();
        CountedString[] cs = new CountedString[10];
        for(int i = 0; i < cs.length; i++) {
            cs[i] = new CountedString("hi");
            m.put(cs[i], new Integer(i));
        }
        System.out.println(m);
        for(int i = 0; i < cs.length; i++) {
            System.out.print("Looking up " + cs[i]);
            System.out.println(m.get(cs[i]));
        }
    }
} ///:~

```

CountedString inclut une **String** et un **id** représentant le nombre d'objets **CountedString** contenant une **String** identique. Le compte est réalisé dans le constructeur en parcourant la **static ArrayList** où toutes les **Strings** sont stockées.

Les méthodes **hashCode()** et **equals()** renvoient des résultats basés sur les deux champs ; si elles étaient basées juste sur la **String** ou sur l'**id**, il y aurait eu des doublons pour des valeurs distinctes.

Notez comme la fonction de hachage est simple : le code de hachage de la **String** multiplié par l'**id**. Généralement, la qualité et la rapidité d'une fonction de hachage est inversement proportionnelle à sa taille.

Dans **main()**, un ensemble d'objets **CountedString** est créé, en utilisant la même **String** pour montrer que les doublons créent des valeurs uniques grâce au compteur **id**. Le **HashMap** est affiché afin de voir son organisation interne (aucun ordre n'est discernable) ; chaque clef est alors recherchée individuellement pour démontrer que le mécanisme de recherche fonctionne correctement.

Stocker des références

La bibliothèque **java.lang.ref** contient un ensemble de classes qui permettent une plus grande flexibilité dans le nettoyage des objets, et qui se révèlent particulièrement pratiques lorsqu'on a de gros objets qui peuvent saturer la mémoire. Il y a trois classes dérivées de la classe abstraite **Reference** : **SoftReference**, **WeakReference** et **PhantomReference**. Chacune d'entre elles fournit un niveau différent d'abstraction au ramasse miettes, si l'objet en question n'est accessible *qu'à travers* un de ces objets **Reference**.

Si un objet est *accessible* cela veut dire que l'objet peut être trouvé quelque part dans le programme. Ceci peut vouloir dire qu'on a une référence ordinaire sur la pile qui pointe directement sur l'objet, mais on peut aussi avoir une référence sur un objet qui possède une référence sur l'objet en question ; il peut y avoir de nombreux liens intermédiaires. Si un objet est accessible, le ramasse miettes ne peut pas le nettoyer parce qu'il est toujours utilisé par le programme. Si un objet n'est pas accessible, le programme ne dispose d'aucun moyen pour y accéder et on peut donc nettoyer cet objet tranquillement.

On utilise des objets **Reference** quand on veut continuer à stocker une référence sur cet objet - on veut être capable d'atteindre cet objet - mais on veut aussi permettre au ramasse miettes de nettoyer cet objet. Il s'agit donc d'un moyen permettant de continuer à utiliser l'objet, mais si la saturation de la mémoire est imminente, on permet que cet objet soit nettoyé.

Un objet **Reference** sert donc d'intermédiaire entre le programme et la référence ordinaire, *et* aucune référence ordinaire sur cet objet ne doit exister (mis à part celles encapsulées dans les objets **Reference**). Si le ramasse miette découvre qu'un objet est accessible à travers une référence ordinaire, il ne nettoiera pas cet objet.

Dans l'ordre **SoftReference**, **WeakReference** et **PhantomReference**, chacune d'entre elles est « plus faible » que la précédente, et correspond à un niveau différent d'accessibilité. Les références douces (**SoftReferences**) permettent d'implémenter des caches concernés par les problèmes de mémoire. Les références faibles (**WeakReferences**) sont destinées à implémenter des « mappages canoniques » - où des instances d'objets peuvent être utilisées simultanément dans différents endroits du programme, pour économiser le stockage - qui n'empêchent pas leurs clefs (ou valeurs) d'être nettoyées. Les références fantômes (**PhantomReferences**) permettent d'organiser les actions de nettoyage pre-mortem d'une manière plus flexible que ce qui est possible avec le mécanisme de finalisation de Java.

Pour les **SoftReferences** et les **WeakReferences**, on peut choisir de les stocker dans une **ReferenceQueue** (le dispositif utilisé pour les actions de nettoyage pre-mortem) ou non, mais une **PhantomReference** ne peut être créée que dans une **ReferenceQueue**. En voici la démonstration :

```
///  
// Illustrer les objets Reference.  
  
import java.lang.ref.*;
```

```
class VeryBig {
    static final int SZ = 10000;
    double[] d = new double[SZ];
    String ident;

    public VeryBig(String id) { ident = id; }
    public String toString() { return ident; }
    public void finalize() {
        System.out.println("Finalizing " + ident);
    }
}

public class References {
    static ReferenceQueue rq= new ReferenceQueue();
    public static void checkQueue() {
        Object inq = rq.poll();
        if(inq != null)
            System.out.println("In queue: " +
                (VeryBig)((Reference)inq).get());
    }
    public static void main(String[] args) {
        int size = 10;
        // La taille peut être choisie via la ligne de commande :
        if(args.length > 0)
            size = Integer.parseInt(args[0]);
        SoftReference[] sa =
            new SoftReference[size];
        for(int i = 0; i < sa.length; i++) {
            sa[i] = new SoftReference(
                new VeryBig("Soft " + i), rq);
            System.out.println("Just created: " +
                (VeryBig)sa[i].get());
            checkQueue();
        }
        WeakReference[] wa =
```

```

        new WeakReference[size];
    for(int i = 0; i < wa.length; i++) {
        wa[i] = new WeakReference(
            new VeryBig("Weak " + i), rq);
        System.out.println("Just created: " +
            (VeryBig)wa[i].get());
        checkQueue();
    }

    SoftReference s = new SoftReference(
        new VeryBig("Soft"));

    WeakReference w = new WeakReference(
        new VeryBig("Weak"));

    System.gc();

    PhantomReference[] pa =
        new PhantomReference[size];
    for(int i = 0; i < pa.length; i++) {
        pa[i] = new PhantomReference(
            new VeryBig("Phantom " + i), rq);
        System.out.println("Just created: " +
            (VeryBig)pa[i].get());
        checkQueue();
    }
}

} ///:~

```

Quand on lance ce programme (vous voudrez probablement piper la sortie à travers un utilitaire « more » afin de pouvoir l'observer page par page), on verra que les objets sont récupérés par le ramasse miettes, même si on a toujours accès à eux à travers les objets **Reference** (pour obtenir la référence réelle sur l'objet, il faut utiliser la méthode **get()**). On notera aussi que **ReferenceQueue** renvoie toujours une **Reference** contenant un objet **null**. Pour utiliser les références, on peut dériver la classe **Reference** particulière qui nous intéresse et ajouter des méthodes au nouveau type de **Reference**.

Le WeakHashMap

La bibliothèque de conteneurs propose une **Map** spéciale pour stocker les références faibles : le **WeakHashMap**. Cette classe est conçue pour faciliter la création de mappages canoniques. Dans de tels mappages, on économise sur le stockage en ne créant qu'une instance d'une valeur particulière. Quand le programme a besoin de cette valeur, il recherche l'objet existant dans le mappage et l'utilise (plutôt que d'en

créer un complètement nouveau). Le mappage peut créer les valeurs comme partie de son initialisation, mais il est plus courant que les valeurs soient créées à la demande.

Puisqu'il s'agit d'une technique permettant d'économiser sur le stockage, il est très pratique que le **WeakHashMap** autorise le ramasse miettes à nettoyer automatiquement les clefs et les valeurs. Aucune opération particulière n'est nécessitée sur les clefs et les valeurs qu'on veut placer dans le **WeakHashMap** ; ils sont automatiquement encapsulés dans des **WeakReferences** par le **WeakHashMap**. Le déclenchement qui autorise le nettoyage survient lorsque la clef n'est plus utilisée, ainsi que démontré dans cet exemple :

```
//: c09:CanonicalMapping.java
// Illustre les WeakHashMaps.

import java.util.*;
import java.lang.ref.*;

class Key {
    String ident;

    public Key(String id) { ident = id; }

    public String toString() { return ident; }

    public int hashCode() {
        return ident.hashCode();
    }

    public boolean equals(Object r) {
        return (r instanceof Key)
            && ident.equals(((Key)r).ident);
    }

    public void finalize() {
        System.out.println("Finalizing Key " + ident);
    }
}

class Value {
    String ident;

    public Value(String id) { ident = id; }

    public String toString() { return ident; }

    public void finalize() {
        System.out.println("Finalizing Value " + ident);
    }
}
```



```

    }

    public class CanonicalMapping {

        public static void main(String[] args) {

            int size = 1000;

            // La taille peut être choisie via la ligne de commande :

            if(args.length > 0)

                size = Integer.parseInt(args[0]);

            Key[] keys = new Key[size];

            WeakHashMap whm = new WeakHashMap();

            for(int i = 0; i < size; i++) {

                Key k = new Key(Integer.toString(i));

                Value v = new Value(Integer.toString(i));

                if(i % 3 == 0)

                    keys[i] = k; // Save as "real" references

                whm.put(k, v);

            }

            System.gc();

        }

    } ///:~

```

La classe **Key** doit fournir les méthodes **hashCode()** et **equals()** puisqu'elle est utilisée comme clef dans une structure de données hachée, comme décrit précédemment dans ce chapitre.

Quand on lance le programme, on s'aperçoit que le ramasse miettes évite une clef sur trois, parce qu'une référence ordinaire sur cette clef a aussi été placée dans le tableau **keys** et donc ces objets ne peuvent être nettoyés.

Les itérateurs revisités

Nous pouvons maintenant démontrer la vraie puissance d'un **Iterator** : la capacité de séparer l'opération de parcourir une séquence de la structure sous-jacente de cette séquence. Dans l'exemple suivant, la classe **PrintData** utilise un **Iterator** pour se déplacer à travers une séquence et appelle la méthode **toString()** pour chaque objet. Deux types de conteneurs différents sont créés - une **ArrayList** et un **HashMap** - et remplis, respectivement, avec des objets **Mouse** et **Hamster** (ces classes ont été définies précédemment dans ce chapitre). Parce qu'un **Iterator** cache la structure sous-jacente du conteneur associé, **PrintData** ne se soucie pas du type de conteneur dont l'**Iterator** provient :

```

///: c09:Iterators2.java

```

```

// Les Iterators revisités.

import java.util.*;

class PrintData {

    static void print(Iterator e) {

        while(e.hasNext())

            System.out.println(e.next());

    }

}

class Iterators2 {

    public static void main(String[] args) {

        ArrayList v = new ArrayList();

        for(int i = 0; i < 5; i++)

            v.add(new Mouse(i));

        HashMap m = new HashMap();

        for(int i = 0; i < 5; i++)

            m.put(new Integer(i), new Hamster(i));

        System.out.println("ArrayList");

        PrintData.print(v.iterator());

        System.out.println("HashMap");

        PrintData.print(m.entrySet().iterator());

    }

} ///:~

```

Pour le **HashMap**, la méthode **entrySet()** renvoie un **Set** d'objets **Map.entry**, qui contient à la fois la clef et la valeur pour chaque entrée, afin d'afficher les deux.

Notez que **PrintData.print()** s'appuie sur le fait que les objets dans les conteneurs appartiennent à la classe **Object** et donc l'appel à **toString()** par **System.out.println()** est automatique. Il est toutefois plus courant de devoir assumer qu'un **Iterator** parcourt un conteneur d'un type spécifique. Par exemple, on peut assumer que tous les objets d'un conteneur sont une **Shape** possédant une méthode **draw()**. On doit alors effectuer un transtypage descendant depuis l'**Object** renvoyé par **Iterator.next()** pour produire une **Shape**.

Choisir une implémentation

Vous devriez maintenant être conscient qu'il n'existe que trois types de conteneurs : les **Maps**, les **Lists** et les **Sets**, avec seulement deux ou trois implémentations pour chacune de ces interfaces. Mais si on décide d'utiliser les fonctionnalités offertes par une **interface** particulière, comment choisir l'implémentation qui conviendra le

mieux ?

Il faut bien voir que chaque implémentation dispose de ses propres fonctionnalités, forces et faiblesses. Par exemple, on peut voir dans le diagramme que les classes **Hashtable**, **Vector** et **Stack** sont des reliquats des versions précédentes de Java, ce vieux code testé et retesté n'est donc pas près d'être pris en défaut. D'un autre côté, il vaut mieux utiliser du code Java 2.

La distinction entre les autres conteneurs se ramène la plupart du temps à leur « support sous-jacent » ; c'est à dire la structure de données qui implémente physiquement l'**interface** désirée. Par exemple, les **ArrayLists** et les **LinkedLists** implémentent toutes les deux l'**interface List**, donc un programme produira les mêmes résultats quelle que soit celle qui est choisie. Cependant, une **ArrayList** est sous-tendue par un tableau, tandis qu'une **LinkedList** est implémentée sous la forme d'une liste doublement chaînée, dont chaque objet individuel contient des données ainsi que des références sur les éléments précédents et suivants dans la liste. De ce fait, une **LinkedList** est le choix approprié si on souhaite effectuer de nombreuses insertions et suppressions au milieu de la liste (les **LinkedLists** proposent aussi des fonctionnalités supplémentaires précisées dans **AbstractSequentialList**). Dans les autres cas, une **ArrayList** est typiquement plus rapide.

De même, un **Set** peut être implémenté soit sous la forme d'un **TreeSet** ou d'un **HashSet**. Un **TreeSet** est supporté par un **TreeMap** et est conçu pour produire un **Set** constamment trié. Cependant, si le **Set** est destiné à stocker de grandes quantités d'objets, les performances en insertion du **TreeSet** vont se dégrader. Quand vous écrierez un programme nécessitant un **Set**, choisissez un **HashSet** par défaut, et changez pour un **TreeSet** s'il est plus important de disposer d'un **Set** constamment trié.

Choisir entre les Lists

Un test de performances constitue la façon la plus flagrante de voir les différences entre les implémentations des **Lists**. Le code suivant crée une classe interne de base à utiliser comme structure de test, puis crée un tableau de classes internes anonymes, une pour chaque test différent. Chacune de ces classes internes est appelée par la méthode **test()**. Cette approche permet d'ajouter et de supprimer facilement de nouveaux tests.

```
///  
// c09:ListPerformance.java  
  
// Illustre les différences de performance entre les Lists.  
  
import java.util.*;  
  
import com.bruceeckel.util.*;  
  
public class ListPerformance {  
    private abstract static class Tester {  
        String name;  
  
        int size; // Nombre de tests par répétition  
  
        Tester(String name, int size) {  
            this.name = name;  
            this.size = size;  
        }  
  
        abstract void test(List a, int reps);  
    }  
}
```

```
}

private static Tester[] tests = {
    new Tester("get", 300) {
        void test(List a, int reps) {
            for(int i = 0; i < reps; i++) {
                for(int j = 0; j < a.size(); j++)
                    a.get(j);
            }
        }
    },
    new Tester("iteration", 300) {
        void test(List a, int reps) {
            for(int i = 0; i < reps; i++) {
                Iterator it = a.iterator();
                while(it.hasNext())
                    it.next();
            }
        }
    },
    new Tester("insert", 5000) {
        void test(List a, int reps) {
            int half = a.size()/2;
            String s = "test";
            ListIterator it = a.listIterator(half);
            for(int i = 0; i < size * 10; i++)
                it.add(s);
        }
    },
    new Tester("remove", 5000) {
        void test(List a, int reps) {
            ListIterator it = a.listIterator(3);
            while(it.hasNext()) {
                it.next();
                it.remove();
            }
        }
    }
};
```

```
    }
}
},
};

public static void test(List a, int reps) {
    // Une astuce pour imprimer le nom de la classe :
    System.out.println("Testing " +
        a.getClass().getName());
    for(int i = 0; i < tests.length; i++) {
        Collections2.fill(a,
            Collections2.countries.reset(),
            tests[i].size());
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(a, reps);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
}

public static void testArray(int reps) {
    System.out.println("Testing array as List");
    // On ne peut effectuer que les deux premiers tests sur un tableau :
    for(int i = 0; i < 2; i++) {
        String[] sa = new String[tests[i].size];
        Arrays2.fill(sa,
            Collections2.countries.reset());
        List a = Arrays.asList(sa);
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(a, reps);
        long t2 = System.currentTimeMillis();
        System.out.println(": " + (t2 - t1));
    }
}
```

```

public static void main(String[] args) {

    int reps = 50000;

    // Le nombre de répétitions peut être spécifié
    // via la ligne de commande :

    if(args.length > 0)

        reps = Integer.parseInt(args[0]);

    System.out.println(reps + " repetitions");

    testArray(reps);

    test(new ArrayList(), reps);

    test(new LinkedList(), reps);

    test(new Vector(), reps);

}

} ///:~

```

La classe interne **Tester** est **abstract**, pour fournir une classe de base aux tests spécifiques. Elle contient une **String** à imprimer quand le test débute, un paramètre **size** destiné à être utilisé par le test comme quantité d'éléments ou nombre de répétitions des tests, un constructeur pour initialiser les champs et un méthode **abstract test()** qui réalise le travail. Tous les types de tests sont regroupés dans le tableau **tests**, initialisé par différentes classes internes anonymes dérivées de **Tester**. Pour ajouter ou supprimer des tests, il suffit d'ajouter ou de supprimer la définition d'une classe interne dans le tableau, et le reste est géré automatiquement.

Pour comparer l'accès aux tableaux avec l'accès aux conteneurs (et particulièrement avec les **ArrayLists**), un test spécial est créé pour les tableaux en encapsulant un dans une **List** via **Arrays.asList()**. Notez que seuls les deux premiers tests peuvent être réalisés dans ce cas, parce qu'on ne peut insérer ou supprimer des éléments dans un tableau.

La **List** passée à **test()** est d'abord remplie avec des éléments, puis chaque test du tableau **tests** est chronométré. Les résultats dépendent bien entendu de la machine ; ils sont seulement conçus pour donner un ordre de comparaison entre les performances des différents conteneurs. Voici un résumé pour une exécution :

Type	Get	Iteration	Insert	Remove
tableau	1430	3850	na	na
ArrayList	3070	12200	500	46850
LinkedList	16320	9110	110	60
Vector	4890	16250	550	46850

Comme prévu, les tableaux sont plus rapides que n'importe quel conteneur pour les accès aléatoires et les itérations. On peut voir que les accès aléatoires (**get()**) sont bon marché pour les **ArrayLists** et coûteux pour les **LinkedLists** (bizarrement, l'itération est *plus rapide* pour une **LinkedList** que pour une **ArrayList**, ce qui est quelque peu contre-intuitif). D'un autre côté, les insertions et les suppressions au milieu d'une liste sont spectaculairement meilleur marché pour une **LinkedList** que pour une **ArrayList** - et *particulièrement* les suppressions. Les **Vectors** ne sont pas aussi rapides que les **ArrayLists**, et doivent être évités ; ils ne sont

présents dans la bibliothèque que pour fournir une compatibilité ascendante avec le code existant (la seule raison pour laquelle ils fonctionnent dans ce programme est qu'ils ont été adaptés pour être une **List** dans Java 2). La meilleure approche est de choisir une **ArrayList** par défaut, et de changer pour une **LinkedList** si on découvre des problèmes de performance dûs à de nombreuses insertions et suppressions au milieu de la liste. Bien sûr, si on utilise un ensemble d'éléments de taille fixée, il faut se tourner vers un tableau.

Choisir entre les Sets

Suivant la taille du **Set**, on peut se tourner vers un **TreeSet** ou un **HashSet** (si on a besoin de produire une séquence ordonnée à partir d'un **Set**, il faudra utiliser un **TreeSet**). Le programme de test suivant donne une indication de ce compromis :

```
///  
c09:SetPerformance.java  
  
import java.util.*;  
  
import com.bruceeckel.util.*;  
  
public class SetPerformance {  
    private abstract static class Tester {  
        String name;  
  
        Tester(String name) { this.name = name; }  
  
        abstract void test(Set s, int size, int reps);  
    }  
  
    private static Tester[] tests = {  
        new Tester("add") {  
            void test(Set s, int size, int reps) {  
                for(int i = 0; i < reps; i++) {  
                    s.clear();  
  
                    Collections2.fill(s,  
                        Collections2.countries.reset(),size);  
                }  
            }  
        },  
        new Tester("contains") {  
            void test(Set s, int size, int reps) {  
                for(int i = 0; i < reps; i++)  
                    for(int j = 0; j < size; j++)  
                        s.contains(Integer.toString(j));  
            }  
        }  
    }  
}
```

```
    },
    new Tester("iteration") {
        void test(Set s, int size, int reps) {
            for(int i = 0; i < reps * 10; i++) {
                Iterator it = s.iterator();
                while(it.hasNext())
                    it.next();
            }
        }
    },
};

public static void
test(Set s, int size, int reps) {
    System.out.println("Testing " +
        s.getClass().getName() + " size " + size);
    Collections2.fill(s,
        Collections2.countries.reset(), size);
    for(int i = 0; i < tests.length; i++) {
        System.out.print(tests[i].name);
        long t1 = System.currentTimeMillis();
        tests[i].test(s, size, reps);
        long t2 = System.currentTimeMillis();
        System.out.println(": " +
            ((double)(t2 - t1)/(double)size));
    }
}

public static void main(String[] args) {
    int reps = 50000;
    // Le nombre de répétitions peut être spécifié
    // via la ligne de commande :
    if(args.length > 0)
        reps = Integer.parseInt(args[0]);
    // Petit :
    test(new TreeSet(), 10, reps);
}
```



```

    test(new HashSet(), 10, reps);

    // Moyen :

    test(new TreeSet(), 100, reps);

    test(new HashSet(), 100, reps);

    // Gros :

    test(new TreeSet(), 1000, reps);

    test(new HashSet(), 1000, reps);

}

} ///:~

```

Le tableau suivant montre les résultats d'une exécution (bien sûr, vous obtiendrez des valeurs différentes suivant votre ordinateur et la JVM que vous utilisez ; lancez les tests vous-même pour vous faire une idée) :

Type	Test size	Add	Contains	Iteration
TreeSet	10	138.0	115.0	187.0
	100	189.5	151.1	206.5
	1000	150.6	177.4	40.04
HashSet	10	55.0	82.0	192.0
	100	45.6	90.0	202.2
	1000	36.14	106.5	39.39

Les performances d'un **HashSet** sont généralement supérieures à celles d'un **TreeSet** pour toutes les opérations (et en particulier le stockage et la recherche, les deux opérations les plus fréquentes). La seule raison d'être du **TreeSet** est qu'il maintient ses éléments triés, on ne l'utilisera donc que lorsqu'on aura besoin d'un **Set** trié.

Choisir entre les Maps

Lorsqu'on doit choisir entre les différentes implémentations d'une **Map**, sa taille est le critère qui affecte le plus les performances, et le programme de test suivant donne une indication des compromis :

```

///: c09:MapPerformance.java

// Illustre les différences de performance entre les Maps.

import java.util.*;

import com.bruceeckel.util.*;

public class MapPerformance {

    private abstract static class Tester {

        String name;

        Tester(String name) { this.name = name; }
    }
}

```

```
    abstract void test(Map m, int size, int reps);
}

private static Tester[] tests = {
    new Tester("put") {
        void test(Map m, int size, int reps) {
            for(int i = 0; i < reps; i++) {
                m.clear();
                Collections2.fill(m,
                    Collections2.geography.reset(), size);
            }
        }
    },
    new Tester("get") {
        void test(Map m, int size, int reps) {
            for(int i = 0; i < reps; i++)
                for(int j = 0; j < size; j++)
                    m.get(Integer.toString(j));
        }
    },
    new Tester("iteration") {
        void test(Map m, int size, int reps) {
            for(int i = 0; i < reps * 10; i++) {
                Iterator it = m.entrySet().iterator();
                while(it.hasNext())
                    it.next();
            }
        }
    },
};

public static void
test(Map m, int size, int reps) {
    System.out.println("Testing " +
        m.getClass().getName() + " size " + size);
    Collections2.fill(m,
```

```
        Collections2.geography.reset(), size);
    for(int i = 0; i < tests.length; i++) {
        System.out.print(tests[i].name);

        long t1 = System.currentTimeMillis();

        tests[i].test(m, size, reps);

        long t2 = System.currentTimeMillis();

        System.out.println(": " +

            ((double)(t2 - t1)/((double)size)));
    }
}

public static void main(String[] args) {
    int reps = 50000;

    // Le nombre de répétitions peut être spécifié
    // via la ligne de commande :
    if(args.length > 0)
        reps = Integer.parseInt(args[0]);

    // Petit :
    test(new TreeMap(), 10, reps);
    test(new HashMap(), 10, reps);
    test(new Hashtable(), 10, reps);

    // Moyen :
    test(new TreeMap(), 100, reps);
    test(new HashMap(), 100, reps);
    test(new Hashtable(), 100, reps);

    // Gros :
    test(new TreeMap(), 1000, reps);
    test(new HashMap(), 1000, reps);
    test(new Hashtable(), 1000, reps);
}

} ///:~
```

Parce que la taille du dictionnaire constitue le facteur principal, les tests de chronométrage divisent le temps par la taille du dictionnaire pour normaliser chaque mesure. Voici un ensemble de résultats (les vôtres différeront probablement) :

Type	Test size	Put	Get	Iteration
	10	143.0	110.0	186.0
TreeMap	100	201.1	188.4	280.1
	1000	222.8	205.2	40.7
	10	66.0	83.0	197.0
HashMap	100	80.7	135.7	278.5
	1000	48.2	105.7	41.4
	10	61.0	93.0	302.0
Hashtable	100	90.6	143.3	329.0
	1000	54.1	110.95	47.3

Comme on pouvait s'y attendre, les performances d'une **HashTable** sont à peu près équivalentes à celles d'un **HashMap** (bien que ceux-ci soient généralement un petit peu plus rapides). Les **TreeMaps** étant généralement plus lents que les **HashMaps**, pourquoi voudrait-on les utiliser ? En fait, on les utilise non comme des **Maps** mais comme une façon de créer une liste ordonnée. Le comportement d'un arbre est tel qu'il est toujours ordonné et n'a pas besoin d'être spécifiquement trié. Une fois un **TreeMap** rempli, il est possible d'appeler **keySet()** pour récupérer un **Set** des clefs, puis **toArray()** pour produire un tableau de ces clefs. On peut alors utiliser la méthode **static Arrays.binarySearch()** (que nous étudierons plus loin) pour trouver rapidement des objets dans ce tableau trié. Bien sûr, on ne ferait ceci que si, pour une raison ou une autre, le comportement d'un **HashMap** ne convenait pas, puisqu'un **HashMap** est conçu justement pour retrouver rapidement des objets. De plus, on peut facilement créer un **HashMap** à partir d'un **TreeMap** avec une simple création d'objet. Pour résumer, votre premier réflexe si vous voulez utiliser une **Map** devrait être de se tourner vers un **HashMap**, et n'utiliser un **TreeMap** que si vous avez besoin d'une **Map** constamment triée.

Trier et rechercher dans les Lists

Les fonctions effectuant des tris et des recherches dans les **Lists** ont les mêmes noms et signature que celles réalisant ces opérations sur des tableaux d'objets, mais sont des méthodes **static** appartenant à la classe **Collections** au lieu de **Arrays**. En voici un exemple, adapté de **ArraySearching.java** :

```

//: c09:ListSortSearch.java
// Trier et rechercher dans les Lists avec 'Collections.'

import com.bruceeckel.util.*;

import java.util.*;

public class ListSortSearch {

    public static void main(String[] args) {

        List list = new ArrayList();

        Collections2.fill(list,

            Collections2.capitals, 25);

        System.out.println(list + "\n");
    }
}

```

```

    Collections.shuffle(list);

    System.out.println("After shuffling: "+list);

    Collections.sort(list);

    System.out.println(list + "\n");

    Object key = list.get(12);

    int index =

        Collections.binarySearch(list, key);

    System.out.println("Location of " + key +

        " is " + index + ", list.get(" +

        index + ") = " + list.get(index));

    AlphabeticComparator comp =

        new AlphabeticComparator();

    Collections.sort(list, comp);

    System.out.println(list + "\n");

    key = list.get(12);

    index =

        Collections.binarySearch(list, key, comp);

    System.out.println("Location of " + key +

        " is " + index + ", list.get(" +

        index + ") = " + list.get(index));

}

} ///:~

```

L'utilisation de ces méthodes est identique à celles dans **Arrays**, mais une **List** est utilisée à la place d'un tableau. Comme pour les tableaux, il faut passer le **Comparator** utilisé pour trier la liste lorsqu'on fait un appel à **binarySearch()**.

Ce programme illustre aussi la méthode **shuffle()** de la classe **Collections**, qui modifie aléatoirement l'ordre d'une **List**.

Utilitaires

Il existe un certain nombre d'autres méthodes bien pratiques dans la classe **Collections** :

enumeration(Collection)	Renvoie une Enumeration de l'argument.
max(Collection) min(Collection)	Renvoie l'élément maximum ou minimum de l'argument en utilisant la méthode de comparaison naturelle des objets de la Collection .
max(Collection, Comparator) min(Collection, Comparator)	Renvoie l'élément maximum ou minimum de la Collection en utilisant le Comparator .
reverse()	Inverse tous les éléments sur place.
copy(List dest, List src)	Copie les éléments de src vers dest .
fill(List list, Object o)	Remplace tous les éléments de la liste avec o .
nCopies(int n, Object o)	Renvoie une List non-modifiable de taille n dont les références pointent toutes sur o .

Notez que comme **min()** et **max()** fonctionnent avec des objets **Collection**, et non avec des **Lists**, il n'est pas nécessaire que celle-ci soit triée (ainsi que nous l'avons déjà vu, il *faut* trier une **List** ou un tableau avant de leur appliquer **binarySearch()**).

Rendre une Collection ou une Map non-modifiable

Il est souvent bien pratique de créer une version en lecture seule d'une **Collection** ou d'une **Map**. La classe **Collections** permet ceci en passant le conteneur original à une méthode qui en renvoie une version non modifiable. Il existe quatre variantes de cette méthode, pour les **Collections** (si on ne veut pas traiter une **Collection** comme un type plus spécifique), les **Lists**, les **Sets** et les **Maps**. Cet exemple montre la bonne manière pour construire des versions en lecture seule des collections :

```

//: c09:ReadOnly.java

// Utilisation des méthodes Collections.unmodifiable.

import java.util.*;

import com.bruceeckel.util.*;

public class ReadOnly {

    static Collections2.StringGenerator gen =

        Collections2.countries;

    public static void main(String[] args) {

        Collection c = new ArrayList();

        Collections2.fill(c, gen, 25); // Insertion des données

        c = Collections.unmodifiableCollection(c);

        System.out.println(c); // L'accès en lecture est OK

        c.add("one"); // Modification impossible

        List a = new ArrayList();

```

```

Collections2.fill(a, gen.reset(), 25);

a = Collections.unmodifiableList(a);

ListIterator lit = a.listIterator();

System.out.println(lit.next()); // L'accès en lecture est OK

lit.add("one"); // Modification impossible


Set s = new HashSet();

Collections2.fill(s, gen.reset(), 25);

s = Collections.unmodifiableSet(s);

System.out.println(s); // L'accès en lecture est OK

//! s.add("one"); // Modification impossible


Map m = new HashMap();

Collections2.fill(m,

    Collections2.geography, 25);

m = Collections.unmodifiableMap(m);

System.out.println(m); // L'accès en lecture est OK

//! m.put("Ralph", "Howdy!");

}

} ///:~

```

Dans chaque cas, le conteneur doit être rempli de données *avant* de le rendre non-modifiable. Une fois rempli, la meilleure approche consiste à remplacer la référence existante par la référence renvoyée par l'appel à « unmodifiable ». De cette façon, on ne risque pas d'en altérer accidentellement le contenu une fois qu'on l'a rendu non modifiable. D'un autre côté, cet outil permet de garder un conteneur modifiable **private** dans la classe et de renvoyer une référence en lecture seule sur ce conteneur à partir d'une méthode. Ainsi on peut le changer depuis la classe, tandis qu'on ne pourra y accéder qu'en lecture depuis l'extérieur de cette classe.

Appeler la méthode « unmodifiable » pour un type particulier n'implique pas de contrôle lors de la compilation, mais une fois la transformation effectuée, tout appel à une méthode tentant de modifier le contenu du conteneur provoquera une **UnsupportedOperationException**.

Synchroniser une Collection ou une Map

Le mot-clef **synchronized** constitue une partie importante du *multithreading*, un sujet plus compliqué qui ne sera abordé qu'à partir du Chapitre 14. Ici, je noterai juste que la classe **Collections** permet de synchroniser automatiquement un conteneur entier. La syntaxe en est similaire aux méthodes « unmodifiable » :

```

//: c09:Synchronization.java

// Utilisation des méthodes Collections.synchronized.

```

```
import java.util.*;

public class Synchronization {

    public static void main(String[] args) {

        Collection c =

            Collections.synchronizedCollection(

                new ArrayList());

        List list = Collections.synchronizedList(

            new ArrayList());

        Set s = Collections.synchronizedSet(

            new HashSet());

        Map m = Collections.synchronizedMap(

            new HashMap());

    }

} ///:~
```

Dans ce cas, on passe directement le nouveau conteneur à la méthode « synchronisée » appropriée ; ainsi la version non synchronisée ne peut être accédée de manière accidentelle.

Echec rapide

Les conteneurs Java possèdent aussi un mécanisme qui permet d'empêcher que le contenu d'un conteneur ne soit modifié par plus d'un processus. Le problème survient si on itère à travers un conteneur alors qu'un autre processus insère, supprime ou change un objet de ce conteneur. Cet objet peut déjà avoir été passé, ou on ne l'a pas encore rencontré, peut-être que la taille du conteneur a diminué depuis qu'on a appelé **size()** - il existe de nombreux scénarios catastrophes. La bibliothèque de conteneurs Java incorpore un mécanisme d'*échec-rapide* qui traque tous les changements effectués sur le conteneur autres que ceux dont notre processus est responsable. S'il détecte que quelqu'un d'autre tente de modifier le conteneur, il produit immédiatement une **ConcurrentModificationException**. C'est l'aspect « échec-rapide » - il ne tente pas de détecter le problème plus tard en utilisant un algorithme plus complexe.

Il est relativement aisé de voir le mécanisme d'échec rapide en action - tout ce qu'on a à faire est de créer un itérateur et d'ajouter alors un item à la collection sur laquelle l'itérateur pointe, comme ceci :

```
///: c09:FailFast.java

// Illustre la notion d'« échec rapide ».

import java.util.*;

public class FailFast {

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



```

    Collection c = new ArrayList();

    Iterator it = c.iterator();

    c.add("An object");

    // Génère une exception :

    String s = (String)it.next();

}

} ///:~

```

L'exception survient parce que quelque chose est stocké dans le conteneur *après* que l'itérateur ait été produit par le conteneur. La possibilité que deux parties du programme puissent modifier le même conteneur mène à un état incertain, l'exception notifie donc qu'il faut modifier le code - dans ce cas, récupérer l'itérateur *après* avoir ajouté tous les éléments au conteneur.

Notez qu'on ne peut bénéficier de ce genre de surveillance quand on accède aux éléments d'une **List** en utilisant **get()**.

Opérations non supportées

Il est possible de transformer un tableau en **List** grâce à la méthode **Arrays.asList()** :

```

///: c09:Unsupported.java

// Quelquefois les méthodes définies dans les
// interfaces Collection ne fonctionnent pas!

import java.util.*;

public class Unsupported {

    private static String[] s = {

        "one", "two", "three", "four", "five",

        "six", "seven", "eight", "nine", "ten",

    };

    static List a = Arrays.asList(s);

    static List a2 = a.subList(3, 6);

    public static void main(String[] args) {

        System.out.println(a);

        System.out.println(a2);

        System.out.println(

            "a.contains(" + s[0] + ") = " +

```

```

        a.contains(s[0]));
System.out.println(
    "a.containsAll(a2) = " +
    a.containsAll(a2));
System.out.println("a.isEmpty() = " +
    a.isEmpty());
System.out.println(
    "a.indexOf(" + s[5] + ") = " +
    a.indexOf(s[5]));

// Traversée à reculons :
ListIterator lit = a.listIterator(a.size());
while(lit.hasPrevious())
    System.out.print(lit.previous() + " ");
System.out.println();

// Modification de la valeur des éléments :
for(int i = 0; i < a.size(); i++)
    a.set(i, "47");
System.out.println(a);

// Compiles, mais ne marche pas :
lit.add("X"); // Opération non supportée
a.clear(); // Non supporté
a.add("eleven"); // Non supporté
a.addAll(a2); // Non supporté
a.retainAll(a2); // Non supporté
a.remove(s[0]); // Non supporté
a.removeAll(a2); // Non supporté
}
} ///:~

```

En fait, seule une partie des interfaces **Collection** et **List** est implémentée. Le reste des méthodes provoque l'apparition déplaisante d'une chose appelée **UnsupportedOperationException**. Vous en apprendrez plus sur les exceptions dans le chapitre suivant, mais pour résumer l'**interface Collection** - de même que certaines autres **interfaces** de la bibliothèque de conteneurs Java - contient des méthodes « optionnelles », qui peuvent ou non être « supportées » dans la classe concrète qui implémente cette **interface**. Appeler une méthode non supportée provoque une **UnsupportedOperationException** pour indiquer une erreur de programmation.

« Comment !?! » vous exclamez-vous, incrédule. « Tout l'intérêt des **interfaces** et des classes de base provient du fait qu'elles promettent que ces méthodes feront quelque chose d'utile ! Cette affirmation rompt cette promesse - non seulement ces méthodes ne réalisent *pas* d'opération intéressante, mais en plus elles arrêtent le programme ! Qu'en est-il du contrôle de type ? »

Ce n'est pas si grave que ça. Avec une **Collection**, une **List**, un **Set** ou une **Map**, le compilateur empêche toujours d'appeler des méthodes autres que celles présentes dans cette **interface**, ce n'est donc pas comme Smalltalk (qui permet d'appeler n'importe quelle méthode sur n'importe quel objet, appel dont on ne verra la pertinence que lors de l'exécution du programme). De plus, la plupart des méthodes acceptant une **Collection** comme argument ne font que lire cette **Collection** - et les méthodes de « lecture » de **Collection** ne sont *pas* optionnelles.

Cette approche empêche l'explosion du nombre d'interfaces dans la conception. Les autres conceptions de bibliothèques de conteneurs semblent toujours se terminer par une pléthore d'interfaces pour décrire chacune des variations sur le thème principal et sont donc difficiles à appréhender. Il n'est même pas possible de capturer tous les cas spéciaux dans les **interfaces**, car n'importe qui peut toujours créer une nouvelle **interface**. L'approche « opération non supportée » réalise l'un des buts fondamentaux de la bibliothèque de conteneurs Java : les conteneurs sont simples à apprendre et à utiliser ; les opérations non supportées sont des cas spéciaux qui peuvent être appris par la suite. Pour que cette approche fonctionne, toutefois :

1. Une **UnsupportedOperationException** doit être un événement rare. C'est à dire que toutes les opérations doivent être supportées dans la majorité des classes, et seuls quelques rares cas spéciaux peuvent ne pas supporter une certaine opération. Ceci est vrai dans la bibliothèque de conteneurs Java, puisque les classes que vous utiliserez dans 99 pour cent des cas - **ArrayList**, **LinkedList**, **HashSet** et **HashMap**, de même que les autres implémentations concrètes - supportent toutes les opérations. Cette conception fournit une « porte dérobée » si on veut créer une nouvelle **Collection** sans fournir de définition sensée pour toutes les méthodes de l'**interface Collection**, et s'intégrer tout de même dans la bibliothèque.
2. Quand une opération n'est *pas* supportée, il faut une probabilité raisonnable qu'une **UnsupportedOperationException** apparaisse lors de l'implémentation plutôt qu'une fois le produit livré au client. Après tout, elle indique une erreur de programmation : une implémentation a été utilisée de manière incorrecte. Ce point est moins certain, et c'est là où la nature expérimentale de cette conception entre en jeu. Nous ne nous apercevrons de son intérêt qu'avec le temps.

Dans l'exemple précédent, **Arrays.asList()** produit une **List** supportée par un tableau de taille fixe. Il est donc sensé que les seules opérations supportées soient celles qui ne changent pas la taille du tableau. D'un autre côté, si une nouvelle **interface** était requise pour exprimer ce différent type de comportement (peut-être appelée « **FixedSizeList** »), cela laisserait la porte ouverte à la complexité et bientôt on ne saurait où donner de la tête lorsqu'on voudrait utiliser la bibliothèque.

La documentation d'une méthode qui accepte une **Collection**, une **List**, un **Set** ou une **Map** en argument doit spécifier quelles méthodes optionnelles doivent être implémentées. Par exemple, trier requiert les méthodes **set()** et **Iterator.set()**, mais pas **add()** ou **remove()**.

Les conteneurs Java 1.0 / 1.1

Malheureusement, beaucoup de code a été écrit en utilisant les conteneurs Java 1.0 / 1.1, et aujourd'hui encore ces classes continuent d'être utilisées dans du nouveau code. Bien qu'il faille éviter d'utiliser les anciens conteneurs lorsqu'on produit du code nouveau, il faut toutefois être conscient qu'ils existent. Cependant, les anciens conteneurs étaient plutôt limités, il n'y a donc pas tellement à en dire sur eux (puisque'ils appartiennent au passé, j'évitais de trop me pencher sur certaines horribles décisions de conception).

Vector & Enumeration

Le **Vector** était la seule séquence auto-redimensionnable dans Java 1.0 / 1.1, ce qui favorisa son utilisation. Ses défauts sont trop nombreux pour être décrits ici (se référer à la première version de ce livre, disponible sur le CD ROM de ce livre et téléchargeable sur *www.BruceEckel.com*). Fondamentalement, il s'agit d'une **ArrayList** avec des noms de méthodes longs et maladroits. Dans la bibliothèque de conteneurs Java 2, la classe **Vector** a été adaptée afin de pouvoir s'intégrer comme une **Collection** et une **List**, et donc dans l'exemple suivant la méthode **Collections2.fill()** est utilisée avec succès. Ceci peut mener à des effets pervers, car on pourrait croire que la classe **Vector** a été améliorée alors qu'elle n'a été incluse que pour supporter du code pré-Java2.

La version Java 1.0 / 1.1 de l'itérateur a choisi d'inventer un nouveau nom, « énumération », au lieu d'utiliser un terme déjà familier pour tout le monde. L'interface **Enumeration** est plus petite qu'**Iterator**, ne possédant que deux méthodes, et utilise des noms de méthodes plus longs : **boolean hasMoreElements()** renvoie **true** si l'énumération contient encore des éléments, et **Object nextElement()** renvoie l'élément suivant de cette énumération si il y en a encore (autrement elle génère une exception).

Enumeration n'est qu'une interface, et non une implémentation, et même les nouvelles bibliothèques utilisent encore quelquefois l'ancienne **Enumeration** - ce qui est malheureux mais généralement sans conséquence. Bien qu'il faille utiliser un **Iterator** quand on le peut, on peut encore rencontrer des bibliothèques qui renvoient des **Enumerations**.

Toute **Collection** peut produire une **Enumeration** via la méthode **Collections.enumerations()**, comme le montre cet exemple :

```

//: c09:Enumerations.java
// Vectors et Enumerations de Java 1.0 / 1.1.
import java.util.*;
import com.bruceeckel.util.*;

class Enumerations {
    public static void main(String[] args) {
        Vector v = new Vector();
        Collections2.fill(
            v, Collections2.countries, 100);
        Enumeration e = v.elements();
        while(e.hasMoreElements())
            System.out.println(e.nextElement());

        // Produit une Enumeration à partir d'une Collection:
        e = Collections.enumeration(new ArrayList());
    }
} ///:~

```

La classe **Vector** de Java 1.0 / 1.1 ne dispose que d'une méthode **addElement()**, mais **fill()** utilise la méthode **add()** qui a été copiée quand **Vector** a été transformé en **List**. Pour produire une **Enumeration**, il faut appeler **elements()**, on peut alors l'utiliser pour réaliser une itération en avant.

La dernière ligne crée une **ArrayList** et utilise **enumeration()** pour adapter une **Enumeration** à partir de l'**Iterator** de l'**ArrayList**. Ainsi, si on a du vieux code qui requiert une **Enumeration**, on peut tout de même utiliser les nouveaux conteneurs.

Hashtable

Comme on a pu le voir dans les tests de performances de ce chapitre, la **Hashtable** de base est très similaire au **HashMap**, et ce même jusqu'aux noms des méthodes. Il n'y a aucune raison d'utiliser **Hashtable** au lieu de **HashMap** dans du nouveau code.

Stack

Le concept de la pile a déjà été introduit plus tôt avec les **LinkedLists**. Ce qui est relativement étrange à propos de la classe **Stack** Java 1.0 / 1.1 est qu'elle est *dérivée* de **Vector** au lieu d'utiliser un **Vector** comme composant de base. Elle possède donc toutes les caractéristiques et comportements d'un **Vector** en plus des comportements additionnels d'une **Stack**. Il est difficile de savoir si les concepteurs ont choisi délibérément cette approche en la jugeant particulièrement pratique ou si c'était juste une conception naïve.

Voici une simple illustration de **Stack** qui « pousse » chaque ligne d'un tableau **String** :

```
//: c09:Stacks.java
// Illustration de la classe Stack.

import java.util.*;

public class Stacks {
    static String[] months = {
        "January", "February", "March", "April",
        "May", "June", "July", "August", "September",
        "October", "November", "December" };
    public static void main(String[] args) {
        Stack stk = new Stack();
        for(int i = 0; i < months.length; i++)
            stk.push(months[i] + " ");
        System.out.println("stk = " + stk);
        // Traiter une pile comme un Vector :
        stk.addElement("The last line");
        System.out.println(
```

```

        "element 5 = " + stk.elementAt(5));

    System.out.println("popping elements:");

    while(!stk.empty())

        System.out.println(stk.pop());

    }

} ///:~

```

Chaque ligne du tableau **months** est insérée dans la **Stack** avec **push()**, et récupérée par la suite au sommet de la pile avec un **pop()**. On peut aussi réaliser des opérations de **Vector** sur l'objet **Stack**. Ceci est possible car, de par les propriétés de l'héritage, une **Stack** est un **Vector**. Donc toutes les opérations qui peuvent être effectuées sur un **Vector** peuvent aussi être réalisées sur une **Stack**, comme **elementAt()**.

Ainsi que mentionné précédemment, il vaut mieux utiliser une **LinkedList** si on souhaite un comportement de pile.

BitSet

Un **BitSet** est utile si on veut stocker efficacement un grand nombre d'informations on-off. Cette classe n'est efficace toutefois que sur le plan de la taille ; si le but recherché est un accès performant, mieux vaut se tourner vers un tableau de quelque type natif.

De plus, la taille minimum d'un **BitSet** est celle d'un **long**, soit 64 bits. Ce qui implique que si on stocke n'importe quelle quantité plus petite, telle que 8 bits, un **BitSet** introduira du gaspillage ; il vaudrait donc mieux créer sa propre classe, ou utiliser un tableau, pour stocker les flags si la taille est un problème.

Un conteneur normal se redimensionne si on ajoute de nouveaux éléments, et un **BitSet** n'échappe pas à la règle. L'exemple suivant montre comme le **BitSet** fonctionne :

```

///: c09:Bits.java

// Illustration des BitSets.

import java.util.*;

public class Bits {

    static void printBitSet(BitSet b) {

        System.out.println("bits: " + b);

        String bbits = new String();

        for(int j = 0; j < b.size() ; j++)

            bbits += (b.get(j) ? "1" : "0");

        System.out.println("bit pattern: " + bbits);

    }

    public static void main(String[] args) {

```

```
Random rand = new Random();

// Récupère les LSB (Less Significant Bits - NdT: bit de poids fort) de nextInt()
byte bt = (byte)rand.nextInt();

BitSet bb = new BitSet();
for(int i = 7; i >=0; i--)
    if(((1 << i) & bt) != 0)
        bb.set(i);
    else
        bb.clear(i);

System.out.println("byte value: " + bt);
printBitSet(bb);

short st = (short)rand.nextInt();

BitSet bs = new BitSet();
for(int i = 15; i >=0; i--)
    if(((1 << i) & st) != 0)
        bs.set(i);
    else
        bs.clear(i);

System.out.println("short value: " + st);
printBitSet(bs);

int it = rand.nextInt();

BitSet bi = new BitSet();
for(int i = 31; i >=0; i--)
    if(((1 << i) & it) != 0)
        bi.set(i);
    else
        bi.clear(i);

System.out.println("int value: " + it);
printBitSet(bi);

// Teste les bitsets >= 64 bits:
BitSet b127 = new BitSet();
```

```

        b127.set(127);

        System.out.println("set bit 127: " + b127);

        BitSet b255 = new BitSet(65);

        b255.set(255);

        System.out.println("set bit 255: " + b255);

        BitSet b1023 = new BitSet(512);

        b1023.set(1023);

        b1023.set(1024);

        System.out.println("set bit 1023: " + b1023);

    }

} ///:~

```

Le générateur de nombres aléatoires est utilisé pour générer un **byte**, un **short** et un **int** au hasard, et chacun est transformé en motif de bits dans un **BitSet**. Ceci fonctionne bien puisque la taille d'un **BitSet** est de 64 bits, et donc aucune de ces opérations ne nécessite un changement de taille. Un **BitSet** de 512 bits est alors créé. Le constructeur alloue de la place pour deux fois ce nombre de bits. Cependant, on peut tout de même positionner le bit 1024 ou même au delà.

Résumé

Pour résumer les conteneurs fournis dans la bibliothèque standard Java :

1. Un tableau associe des indices numériques à des objets. Il stocke des objets d'un type connu afin de ne pas avoir à transtyper le résultat quand on récupère un objet. Il peut être multidimensionnel, et peut stocker des scalaires. Cependant, sa taille ne peut être modifiée une fois créé.
2. Une **Collection** stocke des éléments, alors qu'une **Map** stocke des paires d'éléments associés.
3. Comme un tableau, une **List** associe aussi des indices numériques à des objets - on peut penser aux tableaux et aux **Lists** comme à des conteneurs ordonnés. Une **List** se redimensionne automatiquement si on y ajoute des éléments. Mais une **List** ne peut stocker que des références sur des **Objects**, elle ne peut donc stocker des scalaires et il faut toujours transtyper le résultat quand on récupère une référence sur un **Object** du conteneur.
4. Utiliser une **ArrayList** si on doit effectuer un grand nombre d'accès aléatoires, et une **LinkedList** si on doit réaliser un grand nombre d'insertions et de suppressions au sein de la liste.
5. Le comportement de files, files doubles et piles est fourni via les **LinkedLists**.
6. Une **Map** est une façon d'associer non pas des nombres, mais des *objets* à d'autres objets. La conception d'un **HashMap** est focalisée sur les temps d'accès, tandis qu'un **TreeMap** garde ses clefs ordonnées, et n'est donc pas aussi rapide qu'un **HashMap**.
7. Un **Set** n'accepte qu'une instance de valeur de chaque objet. Les **HashSets** fournissent des temps d'accès optimaux, alors que les **TreeSets** gardent leurs éléments ordonnés.
8. Il n'y a aucune raison d'utiliser les anciennes classes **Vector**, **Hashtable** et **Stack** dans du nouveau code.

Les conteneurs sont des outils qu'on peut utiliser jour après jour pour rendre les programmes plus simples, plus puissants et plus efficaces.

Exercices

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

1. Créez un tableau de **doubles** et remplissez-le avec **fill()** en utilisant **RandDoubleGenerator**. Affichez les résultats.
2. Créez une nouvelle classe **Gerbil** possédant un **int gerbilNumber** initialisé dans le constructeur (similaire à l'exemple **Mouse** de ce chapitre). Donnez-lui une méthode **hop()** qui affiche son numéro de gerboise et un message indiquant qu'elle est en train de sauter. Créez une **ArrayList** et ajoutez-y un ensemble d'objets **Gerbil**. Maintenant utilisez la méthode **get()** pour parcourir la **List** et appelez **hop()** pour chaque **Gerbil**.
3. Modifiez l'exercice 2 afin d'utiliser un **Iterator** lors du parcours de la **List** pour appeler **hop()**.
4. Prenez la classe **Gerbil** de l'exercice 2 et placez-la dans une **Map** à la place, en associant le nom (une **String**) de la **Gerbil** à chaque **Gerbil** (la valeur) stockée dans la table. Récupérer un **Iterator** sur l'ensemble des clefs (obtenu via **keySet()**) et utilisez-le pour parcourir la **Map**, récupérez la **Gerbil** pour chaque clef, imprimez son nom et dites-lui de sauter avec la méthode **hop()**.
5. Créez une **List** (essayez avec une **ArrayList** et une **LinkedList**) et remplissez-la en utilisant **Collections2.countries**. Triez cette liste et affichez-la, puis appliquez-lui **Collections.shuffle()** plusieurs fois, en l'imprimant à chaque fois pour voir les effets de cette méthode.
6. Montrez que vous ne pouvez ajouter que des objets **Mouse** dans une **MouseListener**.
7. Modifiez **MouseListener.java** de manière à ce qu'elle hérite de **ArrayList** au lieu d'utiliser la composition. Illustrez le problème de cette approche.
8. Réparez **CatsAndDogs.java** en créant un conteneur **Cats** (en utilisant **ArrayList**) qui n'accepte que des objets **Cats**.
9. Créez un conteneur qui encapsule un tableau de **Strings**, qui ne stocke et ne renvoie que des **Strings**, afin de ne pas avoir de transtypage à faire lorsqu'on l'utilise. Si le tableau interne n'est pas assez grand pour l'insertion suivante, le conteneur devra se redimensionner automatiquement. Dans **main()**, comparez les performances de votre conteneur avec une **ArrayList** contenant des **Strings**.
10. Répétez l'exercice 9 pour un conteneur d'**ints**, et comparez les performances avec une **ArrayList** contenant des objets **Integer**. Dans le test de performances, incluez en plus le fait d'incrémenter chaque objet du conteneur.
11. Créez un tableau pour chaque type primitif et un tableau de **Strings**, remplissez chaque tableau en utilisant un générateur fourni parmi les utilitaires de **com.bruceeckel.util**, et affichez chaque tableau en utilisant la méthode **print()** appropriée.
12. Créez un générateur qui produise des noms de personnages de vos films préférés (que pensez-vous de *Snow White* ou *Star Wars*), et revienne au début quand il n'a plus de noms à proposer. Utilisez les utilitaires de **com.bruceeckel.util** pour remplir un tableau, une **ArrayList**, une **LinkedList** et les deux types de **Set** disponibles, puis imprimez chaque conteneur.
13. Créez une classe contenant deux objets **String**, et rendez-la **Comparable** afin que la comparaison ne porte que sur la première **String**. Remplissez un tableau et une **ArrayList** avec des objets de cette classe, en utilisant le générateur **geography**. Montrez que le tri fonctionne correctement. Créez maintenant un **Comparator** qui porte sur la deuxième **String**, et montrez que le tri fonctionne aussi ; effectuez une recherche en utilisant votre **Comparator**.
14. Modifiez l'exercice 13 afin d'utiliser un tri alphabétique.
15. Utilisez **Arrays2.RandStringGenerator** pour remplir un **TreeSet** utilisant un tri alphabétique. Imprimez le **TreeSet** pour vérifier l'ordre.
16. Créez une **ArrayList** et une **LinkedList**, et remplissez-les en utilisant le générateur **Collections2.capitals**. Imprimez chaque liste en utilisant un **Iterator** ordinaire, puis insérer une liste dans l'autre en utilisant un **ListIterator**, en insérant un élément de la deuxième liste entre chaque élément de la première liste. Réalisez maintenant cette insertion en partant de la fin de la première liste et en la parcourant à l'envers.

17. Ecrivez une méthode qui utilise un **Iterator** pour parcourir une **Collection** et imprime le code de hachage de chaque objet du conteneur. Remplissez tous les types de **Collections** existants avec des objets et utilisez votre méthode avec chaque conteneur.
18. Réparez le problème d'**InfiniteRecursion.java**.
19. Créez une classe, puis créez un tableau d'objets de votre classe. Remplissez une **List** à partir du tableau. Créez un sous-ensemble de la **List** en utilisant **subList()**, puis supprimez cet ensemble de la **List** avec **removeAll()**.
20. Modifiez l'exercice 6 du Chapitre 7 afin de stocker les **Rodents** dans une **ArrayList** et utilisez un **Iterator** pour parcourir la séquence des **Rodents**. Rappelez-vous qu'une **ArrayList** ne stocke que des **Objects** et qu'on doit donc les transtyper pour retrouver le comportement d'un **Rodent**.
21. En vous basant sur **Queue.java**, créez une classe **DoubleFile** et testez-la.
22. Utilisez un **TreeMap** dans **Statistics.java**. Ajoutez maintenant du code afin de tester les différences de performances entre **HashMap** et **TreeMap** dans ce programme.
23. Créez une **Map** et un **Set** contenant tous les pays dont le nom commence par un 'A'.
24. Remplissez un **Set** à l'aide de **Collections2.countries**, utilisez plusieurs fois les mêmes données et vérifiez que le **Set** ne stocke qu'une seule instance de chaque donnée. Testez ceci avec les deux types de **Set**.
25. A partir de **Statistics.java**, créez un programme qui lance le test plusieurs fois et vérifie si un nombre a tendance à apparaître plus souvent que les autres dans les résultats.
26. Réécrivez **Statistics.java** en utilisant un **HashSet** d'objets **Counter** (vous devrez modifier la classe **Counter** afin qu'elle fonctionne avec un **HashSet**). Quelle approche semble la meilleure ?
27. Modifiez la classe de l'exercice 13 afin qu'elle puisse être stockée dans un **HashSet** et comme clef dans un **HashMap**.
28. Créez un **SlowSet** en vous inspirant de **SlowMap.java**.
29. Appliquez les tests de **Map1.java** à **SlowMap** pour vérifier que cette classe fonctionne. Corrigez dans **SlowMap** tout ce qui ne marche pas correctement.
30. Implémentez le reste de l'interface **Map** pour **SlowMap**.
31. Modifiez **MapPerformance.java** afin d'inclure les tests pour **SlowMap**.
32. Modifiez **SlowMap** afin qu'elle utilise une seule **ArrayList** d'objets **MPair** au lieu de deux **ArrayLists**. Vérifiez que la version modifiée fonctionne correctement. Utilisez **MapPerformance.java** pour tester cette nouvelle **Map**. Modifiez maintenant la méthode **put()** afin qu'elle effectue un **sort()** après chaque insertion, et modifiez **get()** afin d'utiliser **Collections.binarySearch()** pour récupérer la clef. Comparez les performances de la nouvelle version avec les anciennes.
33. Ajoutez un champ **char** à **CountedString** qui soit aussi initialisé dans le constructeur, et modifiez les méthodes **hashCode()** et **equals()** afin d'inclure la valeur de ce **char**.
34. Modifiez **SimpleHashMap** afin de signaler les collisions, et testez ce comportement en ajoutant les mêmes données plusieurs fois afin de voir les collisions.
35. Modifiez **SimpleHashMap** afin de signaler le nombre d'objets testés lorsqu'une collision survient. Autrement dit, combien d'appels à **next()** doivent être effectués sur l'**Iterator** parcourant la **LinkedList** pour trouver l'occurrence recherchée ?
36. Implémentez les méthodes **clear()** et **remove()** pour **SimpleHashMap**.
37. Implémentez le reste de l'interface **Map** pour **SimpleHashMap**.
38. Ajoutez une méthode **private rehash()** à **SimpleHashMap** qui soit invoquée lorsque le facteur de charge dépasse 0,75. Au cours du rehachage, doublez le nombre de seaux et recherchez le premier nombre premier qui soit supérieur à cette valeur pour déterminer le nouveau nombre de seaux.
39. Créez et testez un **SimpleHashSet** en vous inspirant de **SimpleHashMap.java**.
40. Modifiez **SimpleHashMap** afin d'utiliser des **ArrayLists** au lieu de **LinkedLists**. Modifiez **MapPerformance.java** afin de comparer les performances des deux implémentations.
41. Consultez la documentation HTML du JDK (téléchargeable sur java.sun.com) de la classe **HashMap**. Créez un **HashMap**, remplissez-le avec des éléments et déterminez son facteur de charge. Testez la vitesse d'accès aux éléments de ce dictionnaire ; essayez d'augmenter cette vitesse en créant un nouveau

- HashMap** d'une capacité initiale supérieure et en copiant l'ancien dictionnaire dans le nouveau.
42. Dans le Chapitre 8, localisez l'exemple **GreenHouse.java** comportant 3 fichiers. Dans **Controller.java**, la classe **EventSet** n'est qu'un conteneur. Changez le code afin d'utiliser une **LinkedList** au lieu d'un **EventSet**. Remplacer **EventSet** par **LinkedList** ne suffira pas, il vous faudra aussi utiliser un **Iterator** pour parcourir l'ensemble d'événements.
 43. (Difficile). Créez votre propre classe de dictionnaire haché, particularisée pour un type particulier de clefs : **String** par exemple. Ne la faites pas dériver de **Map**. A la place, dupliquez les méthodes afin que les méthodes **put()** et **get()** n'acceptent que des objets **String** comme clef, et non des **Objects**. Tout ce qui touche les clefs ne doit pas impliquer de types génériques, mais uniquement des **Strings** afin d'éviter les surcoûts liés au transtypage. Votre but est de réaliser l'implémentation la plus rapide possible. Modifiez **MapPerformance.java** afin de tester votre implémentation contre un **HashMap**.
 44. (Difficile). Trouvez le code source de **List** dans la bibliothèque de code source Java fournie dans toutes les distributions Java. Copiez ce code et créez une version spéciale appelée **intList** qui ne stocke que des **ints**. Réfléchissez à ce qu'il faudrait pour créer une version spéciale de **List** pour chacun des types scalaires. Réfléchissez maintenant à ce qu'il faudrait si on veut créer une classe de liste chaînée qui fonctionne avec tous les types scalaires. Si les types paramétrés sont implémentés un jour dans Java, ils fourniront un moyen de réaliser ce travail automatiquement (entre autres).
-

[44] Il est toutefois possible de s'enquérir de la taille du **vector**, et la méthode **at()** effectue, *elle*, un contrôle sur les indices.

[45] C'est l'un des points où le C++ est indiscutablement supérieur à Java, puisque le C++ supporte la notion de *types paramétrés* grâce au mot-clef **template**.

[46] Le programmeur C++ remarquera comme le code pourrait être réduit en utilisant des arguments par défaut et des templates. Le programmeur Python, lui, notera que cette bibliothèque est complètement inutile dans ce langage.

[47] Par Joshua Bloch de chez Sun.

[48] Ces données ont été trouvées sur Internet, et parsées ensuite par un programme Python (cf. www.Python.org).

[49] Voici un endroit où la surcharge d'opérateur serait appréciée.

[50] Si ces accélérations ne suffisent pas pour répondre aux besoins de performance du programme, il est toujours possible d'accélérer les accès en implémentant une nouvelle **Map** et en la particularisant pour le type spécifique destiné à être stocké pour éviter les délais dus au transtypage en **Object** et inversement. Pour atteindre des niveaux encore plus élevés de performance, les fanas de vitesse pourront se référer à *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition* de Donald Knuth pour remplacer les listes par des tableaux qui possèdent deux avantages supplémentaires : leurs caractéristiques peuvent être optimisées pour le stockage sur disque et ils peuvent économiser la plus grande partie du temps passée à créer et détruire les enregistrements individuels.