

28.04.01 - version 5.4 :

- Nettoyage du code html (titres, paragraphes, tableaux, images, ancres) par Armel.

17.07.2000 - version 5.3 :

- Corrections apportées par Olivier Thomann.

26.06.2000 - version 5.2 :

- Insertion du journal de log.

19.06.2000 - version 5.1 :

- Première publication sur eGroups.

A : Passage & et Retour d'Objets

Vous devriez maintenant être conscient que lorsque vous « passez » un objet, vous passez en fait une référence sur cet objet.

Presque tous les langages de programmation possèdent une façon « normale » de passer des objets, et la plupart du temps tout se passe bien. Mais il arrive toujours un moment où on doit faire quelque chose d'un peu hors-norme, et alors les choses se compliquent un peu (voire beaucoup dans le cas du C++). Java ne fait pas exception à la règle, et il est important de comprendre exactement les mécanismes du passage d'arguments et de la manipulation des objets passés. Cette annexe fournit des précisions quant à ces mécanismes.

Ou si vous préférez, si vous provenez d'un langage de programmation qui en disposait, cette annexe répond à la question « Est-ce que Java utilise des pointeurs ? ». Nombreux sont ceux qui ont affirmé que les pointeurs sont difficiles à manipuler et dangereux, donc à proscrire, et qu'en tant que langage propre et pur destiné à alléger le fardeau quotidien de la programmation, Java ne pouvait décemment contenir de telles choses. Cependant, il serait plus exact de dire que Java dispose de pointeurs ; en fait, chaque identifiant d'objet en Java (les scalaires exceptés) est un pointeur, mais leur utilisation est restreinte et surveillée non seulement par le compilateur mais aussi par le système d'exécution. Autrement dit, Java utilise les pointeurs, mais pas les pointeurs arithmétiques. C'est ce que j'ai appelé les « références » ; et vous pouvez y penser comme à des « pointeurs sécurisés », un peu comme des ciseaux de cours élémentaire - ils ne sont pas pointus, on ne peut donc se faire mal avec qu'en le cherchant bien, mais ils peuvent être lents et ennuyeux.

Passage de références

Quand on passe une référence à une méthode, on pointe toujours sur le même objet. Un simple test le démontre :

```
//: appendixa:PassReferences.java
// Le passage de références.

public class PassReferences {
    static void f(PassReferences h) {
        System.out.println("h inside f(): " + h);
    }
}
```

```

    }

    public static void main(String[] args) {
        PassReferences p = new PassReferences();
        System.out.println("p inside main(): " + p);
        f(p);
    }
} ///:~

```

La méthode **toString()** est automatiquement appelée dans l'instruction `print`, dont **PassReferences** hérite directement de **Object** comme la méthode **toString()** n'est pas redéfinie. La version **toString()** de **Object** est donc utilisée, qui affiche la classe de l'objet suivie de l'adresse mémoire où se trouve l'objet (non pas la référence, mais bien là où est stocké l'objet). La sortie ressemble à ceci :

```

p inside main(): PassReferences@1653748
h inside f(): PassReferences@1653748

```

On peut constater que **p** et **h** référencent bien le même objet. Ceci est bien plus efficace que de créer un nouvel objet **PassReferences** juste pour envoyer un argument à une méthode. Mais ceci amène une importante question.

Aliasing

L'aliasing veut dire que plusieurs références peuvent être attachées au même objet, comme dans l'exemple précédent. Le problème de l'aliasing survient quand quelqu'un *modifie* cet objet. Si les propriétaires des autres références ne s'attendent pas à ce que l'objet change, ils vont avoir des surprises. Ceci peut être mis en évidence avec un simple exemple :

```

//: appendixa:Alias1.java
// Aliasing : deux références sur un même objet.

public class Alias1 {
    int i;

    Alias1(int ii) { i = ii; }

    public static void main(String[] args) {
        Alias1 x = new Alias1(7);
        Alias1 y = x; // Assigne la référence.
        System.out.println("x: " + x.i);
        System.out.println("y: " + y.i);
        System.out.println("Incrementing x");
    }
}

```

```

        x.i++;

        System.out.println("x: " + x.i);

        System.out.println("y: " + y.i);
    }
} ///:~

```

Dans la ligne :

```
Alias1 y = x; // Assigne la référence.
```

une nouvelle référence **Alias1** est créée, mais au lieu de se voir assigner un nouvel objet créé avec **new**, elle reçoit une référence existante. Le contenu de la référence **x**, qui est l'adresse de l'objet sur lequel pointe **x**, est assigné à **y** ; et donc **x** et **y** sont attachés au même objet. Donc quand on incrémente le **i** de **x** dans l'instruction :

```
x.i++
```

le **i** de **y** sera modifié lui aussi. On peut le vérifier dans la sortie :

```

x: 7
y: 7
Incrementing x
x: 8
y: 8

```

Une bonne solution dans ce cas est tout simplement de ne pas le faire : ne pas aliaser plus d'une référence à un même objet dans la même portée. Le code en sera d'ailleurs plus simple à comprendre et à déboguer. Cependant, quand on passe une référence en argument - de la façon dont Java est supposé le faire - l'aliasing entre automatiquement en jeu, et la référence locale créée peut modifier « l'objet extérieur » (l'objet qui a été créé en dehors de la portée de la méthode). En voici un exemple :

```

///: appendixa:Alias2.java

// Les appels de méthodes aliasent implicitement
// leurs arguments.

public class Alias2 {

    int i;

    Alias2(int ii) { i = ii; }

    static void f(Alias2 reference) {

```

```
        reference.i++;
    }

    public static void main(String[] args) {
        Alias2 x = new Alias2(7);

        System.out.println("x: " + x.i);

        System.out.println("Calling f(x)");

        f(x);

        System.out.println("x: " + x.i);
    }
} ///:~
```

Le résultat est :

```
x: 7
Calling f(x)
x: 8
```

La méthode modifie son argument, l'objet extérieur. Dans ce genre de situations, il faut décider si cela a un sens, si l'utilisateur s'y attend, et si cela peut causer des problèmes. En général, on appelle une méthode afin de produire une valeur de retour et/ou une modification de l'état de l'objet *sur lequel est appelée la méthode* (une méthode consiste à « envoyer un message » à cet objet). Il est bien moins fréquent d'appeler une méthode afin de modifier ses arguments ; on appelle cela « appeler une méthode pour ses *effets de bord* ». Une telle méthode qui modifie ses arguments doit être clairement documentée et prévenir à propos de ses surprises potentielles. A cause de la confusion et des chausse-trappes engendrés, il vaut mieux s'abstenir de modifier les arguments. S'il y a besoin de modifier un argument durant un appel de méthode sans que cela ne se répercute sur l'objet extérieur, alors il faut protéger cet argument en en créant une copie à l'intérieur de la méthode. Cette annexe traite principalement de ce sujet.

Création de copies locales

En résumé : tous les passages d'arguments en Java se font par référence. C'est à dire que quand on passe « un objet », on ne passe réellement qu'une référence à un objet qui vit en dehors de la méthode ; et si des modifications sont faites sur cette référence, on modifie l'objet extérieur. De plus :

- l'aliasing survient automatiquement durant le passage d'arguments ;
- il n'y a pas d'objets locaux, que des références locales ;
- les références ont une portée, les objets non ;
- la durée de vie d'un objet n'est jamais un problème en Java ;
- le langage ne fournit pas d'aide (tel que « const ») pour éviter qu'un objet ne soit modifié (c'est à dire pour se prémunir contre les effets négatifs de l'aliasing).

Si on ne fait que lire les informations d'un objet et qu'on ne le modifie pas, la forme la plus efficace de passage d'arguments consiste à passer une référence. C'est bien, car la manière de faire par défaut est aussi la plus efficace. Cependant, on peut avoir besoin de traiter l'objet comme s'il était « local » afin que les modifications

apportées n'affectent qu'une copie locale et ne modifient pas l'objet extérieur. De nombreux langages proposent de créer automatiquement une copie locale de l'objet extérieur, à l'intérieur de la méthode [79]. Java ne dispose pas de cette fonctionnalité, mais il permet tout de même de mettre en oeuvre cet effet.

Passage par valeur

Ceci nous amène à discuter terminologie, ce qui est toujours bon dans un débat. Le sens de l'expression « passage par valeur » dépend de la perception qu'on a du fonctionnement du programme. Le sens général est qu'on récupère une copie locale de ce qu'on passe, mais cela est tempéré par notre façon de penser à propos de ce qu'on passe. Deux camps bien distincts s'affrontent quant au sens de « passage par valeur » :

1. Java passe tout par valeur. Quand on passe un scalaire à une méthode, on obtient une copie distincte de ce scalaire. Quand on passe une référence à une méthode, on obtient une copie de la référence. Ainsi, tous les passages d'arguments se font par valeur. Bien sûr, cela suppose qu'on raisonne en terme de références, mais Java a justement été conçu afin de vous permettre d'ignorer (la plupart du temps) que vous travaillez avec une référence. C'est à dire qu'il permet d'assimiler la référence à « l'objet », car il la déréférence automatiquement lorsqu'on fait un appel à une méthode.
2. Java passe les scalaires par valeur (pas de contestations sur ce point), mais les objets sont passés par référence. La référence est considérée comme un alias sur l'objet ; on ne pense donc *pas* passer une référence, mais on se dit plutôt « je passe l'objet ». Comme on n'obtient pas une copie locale de l'objet quand il est passé à une méthode, il est clair que les objets ne sont pas passés par valeur. Sun semble plutôt soutenir ce point de vue, puisque l'un des mot-clefs « réservés mais non implémentés » est **byvalue** (bien que rien ne précise si ce mot-clef verra le jour).

Après avoir présenté les deux camps et précisé que « cela dépend de la façon dont on considère une référence », je vais tenter de mettre le problème de côté. En fin de compte, ce n'est pas *si* important que cela - ce qui est important, c'est de comprendre que passer une référence permet de modifier l'objet passé en argument.

Clonage d'objets

La raison la plus courante de créer une copie locale d'un objet est qu'on veut modifier cet objet sans impacter l'objet de l'appelant. Si on décide de créer une copie locale, la méthode **clone()** permet de réaliser cette opération. C'est une méthode définie comme **protected** dans la classe de base **Object**, et qu'il faut redéfinir comme **public** dans les classes dérivées qu'on veut cloner. Par exemple, la classe **ArrayList** de la bibliothèque standard redéfinit **clone()**, on peut donc appeler **clone()** sur une **ArrayList** :

```
//: appendixA:Cloning.java
// L'opération clone() ne marche que pour quelques
// composants de la bibliothèque Java standard.
import java.util.*;

class Int {
    private int i;

    public Int(int ii) { i = ii; }

    public void increment() { i++; }
```

```

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

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

        for(int i = 0; i < 10; i++ )
            v.add(new Int(i));

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

        ArrayList v2 = (ArrayList)v.clone();

        // Incrémente tous les éléments de v2 :
        for(Iterator e = v2.iterator();
            e.hasNext(); )
            ((Int)e.next()).increment();

        // Vérifie si les éléments de v ont été modifiés :
        System.out.println("v: " + v);
    }
} ///:~

```

La méthode **clone()** produit un **Object**, qu'il faut alors retrans typer dans le bon type. Cet exemple montre que la méthode **clone()** de **ArrayList** *n'essaie pas* de cloner chacun des objets que l'**ArrayList** contient - l'ancienne **ArrayList** et l'**ArrayList** clonée référencent les mêmes objets. On appelle souvent cela une *copie superficielle*, puisque seule est copiée la « surface » d'un objet. L'objet réel est en réalité constitué de cette « surface », plus les objets sur lesquels les références pointent, plus tous les objets sur lesquels *ces* objets pointent, etc... On s'y réfère souvent en parlant de « réseau d'objets ». On appelle *copie profonde* le fait de copier la totalité de ce fouillis. On peut voir les effets de la copie superficielle dans la sortie, où les actions réalisées sur **v2** affectent **v** :

```

v: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
v: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

Ne pas essayer d'appeler **clone()** sur les objets contenus dans l'**ArrayList** est vraisemblablement une hypothèse raisonnable, car rien ne garantit que ces objets *sont* cloneables [\[80\]](#).

Rendre une classe cloneable

Bien que le méthode clone soit définie dans la classe **Object**, base de toutes les classes, le clonage n'est pas disponible dans toutes les classes [\[81\]](#). Cela semble contraire à l'idée que les méthodes de la classe de base sont

toujours disponibles dans les classes dérivées. Le clonage dans Java va contre cette idée ; si on veut le rendre disponible dans une classe, il faut explicitement ajouter du code pour que le clonage fonctionne.

Utilisation d'une astuce avec `protected`

Afin d'éviter de rendre chaque classe qu'on crée cloneable par défaut, la méthode `clone()` est **protected** dans la classe de base **Object**. Cela signifie non seulement qu'elle n'est pas disponible par défaut pour le programmeur client qui ne fait qu'utiliser la classe (sans en hériter), mais cela veut aussi dire qu'on ne peut pas appeler `clone()` via une référence à la classe de base (bien que cela puisse être utile dans certaines situations, comme le clonage polymorphique d'un ensemble d'**Objects**). C'est donc une manière de signaler, lors de la compilation, que l'objet n'est pas cloneable - et bizarrement, la plupart des classes de la bibliothèque standard Java ne le sont pas. Donc, si on écrit :

```
Integer x = new Integer(1);  
  
x = x.clone();
```

On aura un message d'erreur lors de la compilation disant que `clone()` n'est pas accessible (puisque **Integer** ne la redéfinit pas et qu'elle se réfère donc à la version **protected**). Si, par contre, on se trouve dans une classe dérivée d'**Object** (comme le sont toutes les classes), alors on a la permission d'appeler **Object.clone()** car elle est **protected** et qu'on est un héritier. La méthode `clone()` de la classe de base fonctionne - elle duplique effectivement bit à bit *l'objet de la classe dérivée*, réalisant une opération de clonage classique. Cependant, il faut tout de même rendre *sa propre* méthode de clonage **public** pour la rendre accessible. Donc, les deux points capitaux quand on clone sont :

- Toujours appeler **super.clone()**
- Rendre sa méthode clone **public**

On voudra probablement redéfinir `clone()` dans de futures classes dérivées, sans quoi le `clone()` (maintenant **public**) de la classe actuelle sera utilisé, et pourrait ne pas marcher (cependant, puisque **Object.clone()** crée une copie de l'objet, ça pourrait marcher). L'astuce **protected** ne marche qu'une fois - la première fois qu'on crée une classe dont on veut qu'elle soit cloneable héritant d'une classe qui ne l'est pas. Dans chaque classe dérivée de cette classe la méthode `clone()` sera accessible puisqu'il n'est pas possible en Java de réduire l'accès à une méthode durant la dérivation. C'est à dire qu'une fois qu'une classe est cloneable, tout ce qui en est dérivé est cloneable à moins d'utiliser les mécanismes (décrits ci-après) pour « empêcher » le clonage.

Implémenter l'interface Cloneable

Il y a une dernière chose à faire pour rendre un objet cloneable : implémenter l'**interface Cloneable**. Cette **interface** est un peu spéciale, car elle est vide !

```
interface Cloneable {}
```

La raison d'implémenter cette **interface** vide n'est évidemment pas parce qu'on va surtyper jusqu'à **Cloneable** et appeler une de ses méthodes. L'utilisation d'**interface** dans ce contexte est considérée par certains comme une « astuce » car on utilise une de ses fonctionnalités dans un but autre que celui auquel on pensait originellement. Implémenter l'**interface Cloneable** agit comme une sorte de flag, codé en dur dans le type de la classe. L'**interface Cloneable** existe pour deux raisons. Premièrement, on peut avoir une référence transtypée à un type

de base et ne pas savoir s'il est possible de cloner cet objet. Dans ce cas, on peut utiliser le mot-clef **instanceof** (décrit au chapitre 12) pour savoir si la référence est connectée à un objet qui peut être cloné :

```
if(myReference instanceof Cloneable) // ...
```

La deuxième raison en est qu'on ne veut pas forcément que tous les types d'objets soient cloneables. Donc **Object.clone()** vérifie qu'une classe implémente l'interface **Cloneable**, et si ce n'est pas le cas, elle génère une exception **CloneNotSupportedException**. Donc en général, on est forcé d'implémenter **Cloneable** comme partie du mécanisme de clonage.

Pour un clonage réussi

Une fois les détails d'implémentation de **clone()** compris, il est facile de créer des classes facilement duplicables pour produire des copies locales :

```
//: appendixA:LocalCopy.java
// Créer des copies locales avec clone().
import java.util.*;

class MyObject implements Cloneable {
    int i;

    MyObject(int ii) { i = ii; }

    public Object clone() {
        Object o = null;

        try {
            o = super.clone();
        } catch(CloneNotSupportedException e) {
            System.err.println("MyObject can't clone");
        }

        return o;
    }

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

public class LocalCopy {
    static MyObject g(MyObject v) {
```



```
// Passage par référence, modifie l'objet extérieur :  
  
v.i++;  
  
return v;  
}  
  
static MyObject f(MyObject v) {  
    v = (MyObject)v.clone(); // Copie locale  
    v.i++;  
    return v;  
}  
  
public static void main(String[] args) {  
    MyObject a = new MyObject(11);  
    MyObject b = g(a);  
  
    // On teste l'équivalence des références,  
    // non pas l'équivalence des objets :  
    if(a == b)  
        System.out.println("a == b");  
    else  
        System.out.println("a != b");  
    System.out.println("a = " + a);  
    System.out.println("b = " + b);  
    MyObject c = new MyObject(47);  
    MyObject d = f(c);  
    if(c == d)  
        System.out.println("c == d");  
    else  
        System.out.println("c != d");  
    System.out.println("c = " + c);  
    System.out.println("d = " + d);  
}  
} ///:~
```

Tout d'abord, **clone()** doit être accessible, il faut donc la rendre **public**. Ensuite, il faut que **clone()** commence par appeler la version de **clone()** de la classe de base. La méthode **clone()** appelée ici est celle prédéfinie dans **Object**, et on peut l'appeler car elle est **protected** et donc accessible depuis les classes dérivées.

Object.clone() calcule la taille de l'objet, réserve assez de mémoire pour en créer un nouveau, et copie tous les bits de l'ancien dans le nouveau. On appelle cela une *copie bit à bit*, et c'est typiquement ce qu'on attend d'une méthode **clone()**. Mais avant que **Object.clone()** ne réalise ces opérations, elle vérifie d'abord que la classe est **Cloneable** - c'est à dire, si elle implémente l'interface **Cloneable**. Si ce n'est pas le cas, **Object.clone()** génère une exception **CloneNotSupportedException** pour indiquer qu'on ne peut la cloner. C'est pourquoi il faut entourer l'appel à **super.clone()** dans un bloc try-catch, pour intercepter une exception qui théoriquement ne devrait jamais arriver (parce qu'on a implémenté l'interface **Cloneable**).

Dans **LocalCopy**, les deux méthodes **g()** et **f()** démontrent la différence entre les deux approches concernant le passage d'arguments. **g()** montre le passage par référence en modifiant l'objet extérieur et en retournant une référence à cet objet extérieur, tandis que **f()** clone l'argument, se détachant de lui et laissant l'objet original inchangé. Elle peut alors faire ce qu'elle veut, et même retourner une référence sur ce nouvel objet sans impacter aucunement l'original. À noter l'instruction quelque peu curieuse :

```
v = (MyObject)v.clone();
```

C'est ici que la copie locale est créée. Afin d'éviter la confusion induite par une telle instruction, il faut se rappeler que cet idiome plutôt étrange est tout à fait légal en Java parce que chaque identifiant d'objet est en fait une référence. La référence **v** est donc utilisée pour réaliser une copie de l'objet qu'il référence grâce à **clone()**, qui renvoie une référence au type de base **Object** (car c'est ainsi qu'est définie **Object.clone()**) qui doit ensuite être transtypée dans le bon type.

Dans **main()**, la différence entre les effets des deux approches de passage d'arguments est testée. La sortie est :

```
a == b
a = 12
b = 12
c != d
c = 47
d = 48
```

Il est important de noter que les tests d'équivalence en Java ne regardent pas à l'intérieur des objets comparés pour voir si leurs valeurs sont les mêmes. Les opérateurs **==** et **!=** comparent simplement les références. Si les adresses à l'intérieur des références sont les mêmes, les références pointent sur le même objet et sont donc « égales ». Les opérateurs testent donc si les références sont aliasées sur le même objet !

Le mécanisme de **Object.clone()**

Que se passe-t-il réellement quand **Object.clone()** est appelé, qui rend si essentiel d'appeler **super.clone()** quand on redéfinit **clone()** dans une classe ? La méthode **clone()** dans la classe racine (ie, **Object**) est chargée de la réservation de la mémoire nécessaire au stockage et de la copie bit à bit de l'objet original dans le nouvel espace de stockage. C'est à dire, elle ne crée pas seulement l'emplacement et copie un **Object** - elle calcule précisément la taille de l'objet copié et le duplique. Puisque tout cela se passe dans le code de la méthode **clone()** définie dans la classe de base (qui n'a aucune idée de ce qui est dérivé à partir d'elle), vous pouvez deviner que le processus implique RTTI pour déterminer quel est réellement l'objet cloné. De cette façon, la méthode **clone()** peut réserver la bonne quantité de mémoire et réaliser une copie bit à bit correcte pour ce type.

Quoi qu'on fasse, la première partie du processus de clonage devrait être un appel à **super.clone()**. Ceci pose les fondations de l'opération de clonage en créant une copie parfaite. On peut alors effectuer les autres opérations nécessaires pour terminer le clonage.

Afin de savoir exactement quelles sont ces autres opérations, il faut savoir ce que **Object.clone()** nous fournit. En particulier, clone-t-il automatiquement la destination de toutes les références ? L'exemple suivant teste cela :

```
//: appendixA:Snake.java
// Teste le clonage pour voir si la destination
// des références sont aussi clonées.
```

```
public class Snake implements Cloneable {
    private Snake next;
    private char c;
    // Valeur de i == nombre de segments
    Snake(int i, char x) {
        c = x;
        if(--i > 0)
            next = new Snake(i, (char)(x + 1));
    }
    void increment() {
        c++;
        if(next != null)
            next.increment();
    }
    public String toString() {
        String s = ":" + c;
        if(next != null)
            s += next.toString();
        return s;
    }
    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch(CloneNotSupportedException e) {
```

```

        System.err.println("Snake can't clone");
    }

    return o;
}

public static void main(String[] args) {
    Snake s = new Snake(5, 'a');
    System.out.println("s = " + s);

    Snake s2 = (Snake)s.clone();
    System.out.println("s2 = " + s2);

    s.increment();
    System.out.println(
        "after s.increment, s2 = " + s2);
}
} ///:~

```

Un **Snake** est composé d'un ensemble de segments, chacun de type **Snake**. C'est donc une liste chaînée simple. Les segments sont créés récursivement, en décrémentant le premier argument du constructeur pour chaque segment jusqu'à ce qu'on atteigne zéro. Afin de donner à chaque segment une étiquette unique, le deuxième argument, un **char**, est incrémenté pour chaque appel récursif au constructeur. La méthode **increment()** incrémente récursivement chaque étiquette afin de pouvoir observer les modifications, et **toString()** affiche récursivement chaque étiquette. La sortie est la suivante :

```

s = :a:b:c:d:e
s2 = :a:b:c:d:e
after s.increment, s2 = :a:c:d:e:f

```

Ceci veut dire que seul le premier segment est dupliqué par **Object.clone()**, qui ne réalise donc qu'une copie superficielle. Si on veut dupliquer tout le **Snake** - une copie profonde - il faut réaliser d'autres opérations dans la méthode **clone()** redéfinie.

Typiquement il faudra donc faire un appel à **super.clone()** dans chaque classe dérivée d'une classe cloneable pour s'assurer que toutes les opérations de la classe de base (y compris **Object.clone()**) soient effectuées. Puis cela sera suivi par un appel explicite à **clone()** pour chaque référence contenue dans l'objet ; sinon ces références seront aliées sur celles de l'objet original. Le mécanisme est le même que lorsque les constructeurs sont appelés - constructeur de la classe de base d'abord, puis constructeur de la classe dérivée suivante, et ainsi de suite jusqu'au constructeur de la classe dérivée la plus lointaine de la classe de base. La différence est que **clone()** n'est pas un constructeur, il n'y a donc rien qui permette d'automatiser le processus. Il faut s'assurer de le faire soi-même.

Cloner un objet composé

Il se pose un problème quand on essaye de faire une copie profonde d'un objet composé. Il faut faire l'hypothèse

que la méthode **clone()** des objets membres va à son tour réaliser une copie profonde de *leurs* références, et ainsi de suite. Il s'agit d'un engagement. Cela veut dire que pour qu'une copie profonde fonctionne il faut soit contrôler tout le code dans toutes les classes, soit en savoir suffisamment sur les classes impliquées dans la copie profonde pour être sûr qu'elles réalisent leur propre copie profonde correctement.

Cet exemple montre ce qu'il faut accomplir pour réaliser une copie profonde d'un objet composé :

```
//: appendixA:DeepCopy.java
// Clonage d'un objet composé.

class DepthReading implements Cloneable {
    private double depth;

    public DepthReading(double depth) {
        this.depth = depth;
    }

    public Object clone() {
        Object o = null;
        try {
            o = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        return o;
    }
}

class TemperatureReading implements Cloneable {
    private long time;
    private double temperature;

    public TemperatureReading(double temperature) {
        time = System.currentTimeMillis();
        this.temperature = temperature;
    }

    public Object clone() {
        Object o = null;
        try {
```

```
        o = super.clone();
    } catch(CloneNotSupportedException e) {
        e.printStackTrace(System.err);
    }
    return o;
}
}
```

```
class OceanReading implements Cloneable {
    private DepthReading depth;
    private TemperatureReading temperature;
    public OceanReading(double tdata, double ddata){
        temperature = new TemperatureReading(tdata);
        depth = new DepthReading(ddata);
    }
    public Object clone() {
        OceanReading o = null;
        try {
            o = (OceanReading)super.clone();
        } catch(CloneNotSupportedException e) {
            e.printStackTrace(System.err);
        }
        // On doit cloner les références :
        o.depth = (DepthReading)o.depth.clone();
        o.temperature =
            (TemperatureReading)o.temperature.clone();
        return o; // Transtypage en Object
    }
}
```

```
public class DeepCopy {
    public static void main(String[] args) {
        OceanReading reading =
            new OceanReading(33.9, 100.5);
```

```

        // Maintenant on le clone :

        OceanReading r =

            (OceanReading)reading.clone();

    }

} ///:~

```

DepthReading et **TemperatureReading** sont quasi identiques ; elles ne contiennent toutes les deux que des scalaires. La méthode **clone()** est donc relativement simple : elle appelle **super.clone()** et renvoie le résultat. Notez que le code de **clone()** des deux classes est identique.

OceanReading est composée d'objets **DepthReading** et **TemperatureReading** ; pour réaliser une copie profonde, sa méthode **clone()** doit donc cloner les références à l'intérieur de **OceanReading**. Pour réaliser ceci, le résultat de **super.clone()** doit être transtypé dans un objet **OceanReading** (afin de pouvoir accéder aux références **depth** et **temperature**).

Copie profonde d'une ArrayList

Reprenons l'exemple **ArrayList** exposé plus tôt dans cette annexe. Cette fois-ci la classe **Int2** est cloneable, on peut donc réaliser une copie profonde de l'**ArrayList** :

```

//: appendixa:AddingClone.java

// Il faut apporter quelques modifications
// pour que vos classes soient cloneables.

import java.util.*;

class Int2 implements Cloneable {

    private int i;

    public Int2(int ii) { i = ii; }

    public void increment() { i++; }

    public String toString() {

        return Integer.toString(i);

    }

    public Object clone() {

        Object o = null;

        try {

            o = super.clone();

        } catch(CloneNotSupportedException e) {

            System.err.println("Int2 can't clone");

        }

    }

}

```

```
        return o;
    }
}

// Une fois qu'elle est cloneable, l'héritage
// ne supprime pas cette propriété :
class Int3 extends Int2 {
    private int j; // Automatiquement dupliqué
    public Int3(int i) { super(i); }
}

public class AddingClone {
    public static void main(String[] args) {
        Int2 x = new Int2(10);
        Int2 x2 = (Int2)x.clone();
        x2.increment();
        System.out.println(
            "x = " + x + ", x2 = " + x2);
        // Tout objet hérité est aussi cloneable :
        Int3 x3 = new Int3(7);
        x3 = (Int3)x3.clone();

        ArrayList v = new ArrayList();
        for(int i = 0; i < 10; i++)
            v.add(new Int2(i));
        System.out.println("v: " + v);
        ArrayList v2 = (ArrayList)v.clone();
        // Maintenant on clone chaque élément :
        for(int i = 0; i < v.size(); i++)
            v2.set(i, ((Int2)v2.get(i)).clone());
        // Incrémente tous les éléments de v2 :
        for(Iterator e = v2.iterator();
            e.hasNext(); )
            ((Int2)e.next()).increment();
    }
}
```



```

    // Vérifie si les éléments de v ont été modifiés :

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

    System.out.println("v2: " + v2);

}

} ///:~

```

Int3 est dérivée de **Int2** et un nouveau membre scalaire **int j** a été ajouté. On pourrait croire qu'il faut redéfinir **clone()** pour être sûr que **j** soit copié, mais ce n'est pas le cas. Lorsque la méthode **clone()** de **Int2** est appelée à la place de la méthode **clone()** de **Int3**, elle appelle **Object.clone()**, qui détermine qu'elle travaille avec un **Int3** et duplique tous les bits de **Int3**. Tant qu'on n'ajoute pas de références qui ont besoin d'être clonées, l'appel à **Object.clone()** réalise toutes les opérations nécessaires au clonage, sans se préoccuper de la profondeur hiérarchique où **clone()** a été définie.

Pour réaliser une copie profonde d'une **ArrayList**, il faut donc la cloner, puis la parcourir et cloner chacun des objets pointés par l'**ArrayList**. Un mécanisme similaire serait nécessaire pour réaliser une copie profonde d'un **HashMap**.

Le reste de l'exemple prouve que le clonage s'est bien passé en montrant qu'une fois cloné, un objet peut être modifié sans que l'objet original n'en soit affecté.

Copie profonde via la sérialisation

Quand on examine la sérialisation d'objets dans Java (présentée au Chapitre 11), on se rend compte qu'un objet sérialisé puis désérialisé est, en fait, cloné. Pourquoi alors ne pas utiliser la sérialisation pour réaliser une copie profonde ? Voici un exemple qui compare les deux approches en les chronométrant :

```

//: appendixa:Compete.java

import java.io.*;

class Thing1 implements Serializable {}

class Thing2 implements Serializable {

    Thing1 o1 = new Thing1();

}

class Thing3 implements Cloneable {

    public Object clone() {

        Object o = null;

        try {

            o = super.clone();

        } catch (CloneNotSupportedException e) {

```

```
        System.err.println("Thing3 can't clone");
    }
    return o;
}
}
```

```
class Thing4 implements Cloneable {
    Thing3 o3 = new Thing3();
    public Object clone() {
        Thing4 o = null;
        try {
            o = (Thing4)super.clone();
        } catch(CloneNotSupportedException e) {
            System.err.println("Thing4 can't clone");
        }
        // Clone aussi la donnée membre :
        o.o3 = (Thing3)o3.clone();
        return o;
    }
}
```

```
public class Compete {
    static final int SIZE = 5000;
    public static void main(String[] args)
        throws Exception {
        Thing2[] a = new Thing2[SIZE];
        for(int i = 0; i < a.length; i++)
            a[i] = new Thing2();
        Thing4[] b = new Thing4[SIZE];
        for(int i = 0; i < b.length; i++)
            b[i] = new Thing4();
        long t1 = System.currentTimeMillis();
        ByteArrayOutputStream buf =
            new ByteArrayOutputStream();
    }
}
```

```

    ObjectOutputStream o =
        new ObjectOutputStream(buf);
    for(int i = 0; i < a.length; i++)
        o.writeObject(a[i]);
    // Récupère les copies:
    ObjectInputStream in =
        new ObjectInputStream(
            new ByteArrayInputStream(
                buf.toByteArray()));
    Thing2[] c = new Thing2[SIZE];
    for(int i = 0; i < c.length; i++)
        c[i] = (Thing2)in.readObject();
    long t2 = System.currentTimeMillis();
    System.out.println(
        "Duplication via serialization: " +
        (t2 - t1) + " Milliseconds");
    // Maintenant on tente le clonage :
    t1 = System.currentTimeMillis();
    Thing4[] d = new Thing4[SIZE];
    for(int i = 0; i < d.length; i++)
        d[i] = (Thing4)b[i].clone();
    t2 = System.currentTimeMillis();
    System.out.println(
        "Duplication via cloning: " +
        (t2 - t1) + " Milliseconds");
}
} ///:~

```

Thing2 et **Thing4** contiennent des objets membres afin qu'une copie profonde soit nécessaire. Il est intéressant de noter que bien que les classes **Serializable** soient plus faciles à implémenter, elles nécessitent plus de travail pour les copier. Le support du clonage demande plus de travail pour créer la classe, mais la duplication des objets est relativement simple. Les résultats sont édifiants. Voici la sortie obtenue pour trois exécutions :

```

Duplication via serialization: 940 Milliseconds
Duplication via cloning: 50 Milliseconds

```

```
Duplication via serialization: 710 Milliseconds
```

```
Duplication via cloning: 60 Milliseconds
```

```
Duplication via serialization: 770 Milliseconds
```

```
Duplication via cloning: 50 Milliseconds
```

Outre la différence significative de temps entre la sérialisation et le clonage, vous noterez aussi que la sérialisation semble beaucoup plus sujette aux variations, tandis que le clonage a tendance à être plus stable.

Supporter le clonage plus bas dans la hiérarchie

Si une nouvelle classe est créée, sa classe de base par défaut est **Object**, qui par défaut n'est pas cloneable (comme vous le verrez dans la section suivante). Tant qu'on n'implémente pas explicitement le clonage, celui-ci ne sera pas disponible. Mais on peut le rajouter à n'importe quel niveau et la classe sera cloneable à partir de ce niveau dans la hiérarchie, comme ceci :

```
//: appendixa:HorrorFlick.java
// On peut implémenter le Clonage
// à n'importe quel niveau de la hiérarchie.
import java.util.*;

class Person {}

class Hero extends Person {}

class Scientist extends Person
    implements Cloneable {
    public Object clone() {
        try {
            return super.clone();
        } catch(CloneNotSupportedException e) {
            // Ceci ne devrait jamais arriver :
            // la classe est Cloneable !
            throw new InternalError();
        }
    }
}

class MadScientist extends Scientist {}

public class HorrorFlick {
```

```
public static void main(String[] args) {  
  
    Person p = new Person();  
  
    Hero h = new Hero();  
  
    Scientist s = new Scientist();  
  
    MadScientist m = new MadScientist();  
  
  
    // p = (Person)p.clone(); // Erreur lors de la compilation  
    // h = (Hero)h.clone(); // Erreur lors de la compilation  
  
    s = (Scientist)s.clone();  
  
    m = (MadScientist)m.clone();  
  
}  
} ///:~
```

Tant que le clonage n'est pas supporté, le compilateur bloque toute tentative de clonage. Si le clonage est ajouté dans la classe **Scientist**, alors **Scientist** et tous ses descendants sont cloneables.

Pourquoi cet étrange design ?

Si tout ceci vous semble étrange, c'est parce que ça l'est réellement. On peut se demander comment on en est arrivé là. Que se cache-t-il derrière cette conception ?

Originellement, Java a été conçu pour piloter des boîtiers, sans aucune pensée pour l'Internet. Dans un langage générique tel que celui-ci, il semblait sensé que le programmeur soit capable de cloner n'importe quel objet. C'est ainsi que **clone()** a été placée dans la classe de base **Object**, *mais* c'était une méthode **public** afin qu'un objet puisse toujours être cloné. Cela semblait l'approche la plus flexible, et après tout, quel mal y avait-il à cela ?

Puis, quand Java s'est révélé comme le langage de programmation idéal pour Internet, les choses ont changé. Subitement, des problèmes de sécurité sont apparus, et bien sûr, ces problèmes ont été réglés en utilisant des objets, et on ne voulait pas que n'importe qui soit capable de cloner ces objets de sécurité. Ce qu'on voit donc est une suite de patches appliqués sur l'arrangement initialement simple : **clone()** est maintenant **protected** dans **Object**. Il faut la redéfinir *et* implémenter **Cloneable** et traiter les exceptions.

Il est bon de noter qu'on n'est obligé d'utiliser l'interface **Cloneable** que si on fait un appel à la méthode **clone()** de **Object**, puisque cette méthode vérifie lors de l'exécution que la classe implémente **Cloneable**. Mais dans un souci de cohérence (et puisque de toute façon **Cloneable** est vide), il vaut mieux l'implémenter.

Contrôler la clonabilité

On pourrait penser que, pour supprimer le support du clonage, il suffit de rendre **private** la méthode **clone()**, mais ceci ne marchera pas car on ne peut prendre une méthode de la classe de base et la rendre moins accessible dans une classe dérivée. Ce n'est donc pas si simple. Et pourtant, il est essentiel d'être capable de contrôler si un objet peut être cloné ou non. Une classe peut adopter plusieurs attitudes à ce propos :

1. L'indifférence. Rien n'est fait pour supporter le clonage, ce qui signifie que la classe ne peut être clonée,

mais qu'une classe dérivée peut implémenter le clonage si elle veut. Ceci ne fonctionne que si **Object.clone()** traite comme il faut tous les champs de la classe.

2. Implémenter **clone()**. Respecter la marche à suivre pour l'implémentation de **Cloneable** et redéfinir **clone()**. Dans la méthode **clone()** redéfinie, appeler **super.clone()** et intercepter toutes les exceptions (afin que la méthode **clone()** redéfinie ne génère pas d'exceptions).
3. Supporter le clonage conditionnellement. Si la classe contient des références sur d'autres objets qui peuvent ou non être cloneables (une classe conteneur, par exemple), la méthode **clone()** peut essayer de cloner tous les objets référencés, et s'ils génèrent des exceptions, relayer ces exceptions au programmeur. Par exemple, prenons le cas d'une sorte d'**ArrayList** qui essaierait de cloner tous les objets qu'elle contient. Quand on écrit une telle **ArrayList**, on ne peut savoir quelle sorte d'objets le programmeur client va pouvoir stocker dans l'**ArrayList**, on ne sait donc pas s'ils peuvent être clonés.
4. Ne pas implémenter **Cloneable** mais redéfinir **clone()** en **protected**, en s'assurant du fonctionnement correct du clonage pour chacun des champs. De cette manière, toute classe dérivée peut redéfinir **clone()** et appeler **super.clone()** pour obtenir le comportement attendu lors du clonage. Cette implémentation peut et doit invoquer **super.clone()** même si cette méthode attend un objet **Cloneable** (elle génère une exception sinon), car personne ne l'invoquera directement sur un objet de la classe. Elle ne sera invoquée qu'à travers une classe dérivée, qui, elle, implémente **Cloneable** si elle veut obtenir le fonctionnement désiré.
5. Tenter de bloquer le clonage en n'implémentant pas **Cloneable** et en redéfinissant **clone()** afin de générer une exception. Ceci ne fonctionne que si toutes les classes dérivées appellent **super.clone()** dans leur redéfinition de **clone()**. Autrement, un programmeur est capable de contourner ce mécanisme.
6. Empêcher le clonage en rendant la classe **final**. Si **clone()** n'a pas été redéfinie par l'une des classes parentes, alors elle ne peut plus l'être. Si elle a déjà été redéfinie, la redéfinir à nouveau et générer une exception **CloneNotSupportedException**. Rendre la classe **final** est la seule façon d'interdire catégoriquement le clonage. De plus, si on manipule des objets de sécurité ou dans d'autres situations dans lesquelles on veut contrôler le nombre d'objets créés, il faut rendre tous les constructeurs **private** et fournir une ou plusieurs méthodes spéciales pour créer les objets. De cette manière, les méthodes peuvent restreindre le nombre d'objets créés et les conditions dans lesquelles ils sont créés (un cas particulier en est le patron *singleton* présenté dans *Thinking in Patterns with Java*, téléchargeable à www.BruceEckel.com).

Voici un exemple qui montre les différentes façons dont le clonage peut être implémenté et interdit plus bas dans la hiérarchie :

```
//: appendixA:CheckCloneable.java
// Vérifie si une référence peut être clonée.

// Ne peut être clonée car ne redéfinit pas clone() :
class Ordinary {}

// Redéfinit clone, mais n'implémente pas
// Cloneable :
class WrongClone extends Ordinary {
    public Object clone()
```

```
        throws CloneNotSupportedException {
    return super.clone(); // Génère une exception
}
}

// Fait le nécessaire pour le clonage :
class IsCloneable extends Ordinary
    implements Cloneable {
    public Object clone()
        throws CloneNotSupportedException {
    return super.clone();
}
}

// Interdit le clonage en générant une exception :
class NoMore extends IsCloneable {
    public Object clone()
        throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
}

class TryMore extends NoMore {
    public Object clone()
        throws CloneNotSupportedException {
    // Appelle NoMore.clone(), génère une excpetion :
    return super.clone();
}
}

class BackOn extends NoMore {
    private BackOn duplicate(BackOn b) {
    // Crée une copie de b d'une façon ou d'une autre
    // et renvoie cette copie. C'est une copie sans
```

```
// intérêt, juste pour l'exemple :
return new BackOn();
}

public Object clone() {
    // N'appelle pas NoMore.clone() :
    return duplicate(this);
}
}

// On ne peut dériver cette classe, donc on ne peut
// redéfinir la méthode clone comme dans BackOn:
final class ReallyNoMore extends NoMore {}

public class CheckCloneable {
    static Ordinary tryToClone(Ordinary ord) {
        String id = ord.getClass().getName();
        Ordinary x = null;
        if(ord instanceof Cloneable) {
            try {
                System.out.println("Attempting " + id);
                x = (Ordinary)((Cloneable)ord).clone();
                System.out.println("Cloned " + id);
            } catch(CloneNotSupportedException e) {
                System.err.println("Could not clone "+id);
            }
        }
        return x;
    }

    public static void main(String[] args) {
        // Transtypage ascendant :
        Ordinary[] ord = {
            new IsCloneable(),
            new WrongClone(),
            new NoMore(),
        }
    }
}
```



```

        new TryMore(),

        new BackOn(),

        new ReallyNoMore(),

    };

    Ordinary x = new Ordinary();

    // Ceci ne compilera pas, puisque clone()
    // est protected dans Object:
    //! x = (Ordinary)x.clone();

    // tryToClone() vérifie d'abord si
    // une classe implémente Cloneable :
    for(int i = 0; i < ord.length; i++)
        tryToClone(ord[i]);
    }
} ///:~

```

La première classe, **Ordinary**, représente le genre de classes que nous avons rencontré tout au long de ce livre : pas de support du clonage, mais pas de contrôle sur la clonabilité non plus. Mais si on dispose d'une référence sur un objet **Ordinary** qui peut avoir été transtypé à partir d'une classe dérivée, on ne peut savoir s'il est peut être cloné ou non.

La classe **WrongClone** montre une implémentation incorrecte du clonage. Elle redéfinit bien **Object.clone()** et rend la méthode **public**, mais elle n'implémente pas **Cloneable**, donc quand **super.clone()** est appelée (ce qui revient à un appel à **Object.clone()**), une exception **CloneNotSupportedException** est générée et le clonage échoue.

La classe **IsCloneable** effectue toutes les actions nécessaires au clonage : **clone()** est redéfinie et **Cloneable** implémentée. Cependant, cette méthode **clone()** et plusieurs autres qui suivent dans cet exemple *n'interceptent pas* **CloneNotSupportedException**, mais la font suivre à l'appelant, qui doit alors l'envelopper dans un bloc try-catch. Dans les méthodes **clone()** typiques il faut intercepter **CloneNotSupportedException** à l'intérieur de **clone()** plutôt que de la propager. Cependant dans cet exemple, il est plus intéressant de propager les exceptions.

La classe **NoMore** tente d'interdire le clonage comme les concepteurs de Java pensaient le faire : en générant une exception **CloneNotSupportedException** dans la méthode **clone()** de la classe dérivée. La méthode **clone()** de la classe **TryMore** appelle **super.clone()**, ce qui revient à appeler **NoMore.clone()**, qui génère une exception et empêche donc le clonage.

Mais que se passe-t-il si le programmeur ne respecte pas la chaîne d'appel « recommandée » et n'appelle pas **super.clone()** à l'intérieur de la méthode **clone()** redéfinie ? C'est ce qui se passe dans la classe **BackOn**. Cette classe utilise une méthode séparée **duplicate()** pour créer une copie de l'objet courant et appelle cette méthode dans **clone()** au lieu d'appeler **super.clone()**. L'exception n'est donc jamais générée et la nouvelle classe est cloneable. La seule solution vraiment sûre est montrée dans **ReallyNoMore**, qui est **final** et ne peut donc être dérivée. Ce qui signifie que si **clone()** génère une exception dans la classe **final**, elle ne peut être modifiée via l'héritage et la prévention du clonage est assurée (on ne peut appeler explicitement **Object.clone()** depuis

une classe qui a un niveau arbitraire d'héritage ; on en est limité à appeler **super.clone()**, qui a seulement accès à sa classe parente directe). Implémenter des objets qui traite de sujets relatifs à la sécurité implique donc de rendre ces classes **final**.

La première méthode qu'on voit dans la classe **CheckCloneable** est **tryToClone()**, qui prend n'importe quel objet **Ordinary** et vérifie s'il est cloneable grâce à **instanceof**. Si c'est le cas, il transtype l'objet en **IsCloneable**, appelle **clone()** et retransype le résultat en **Ordinary**, interceptant toutes les exceptions générées. Remarquez l'utilisation de l'identification dynamique du type (voir Chapitre 12) pour imprimer le nom de la classe afin de suivre le déroulement du programme.

Dans **main()**, différents types d'objets **Ordinary** sont créés et transtypés en **Ordinary** dans la définition du tableau. Les deux premières lignes de code qui suivent créent un objet **Ordinary** et tentent de le cloner. Cependant ce code ne compile pas car **clone()** est une méthode **protected** dans **Object**. Le reste du code parcourt le tableau et essaye de cloner chaque objet, reportant le succès ou l'échec de l'opération. Le résultat est :

```
Attempting IsCloneable
Cloned IsCloneable
Attempting NoMore
Could not clone NoMore
Attempting TryMore
Could not clone TryMore
Attempting BackOn
Cloned BackOn
Attempting ReallyNoMore
Could not clone ReallyNoMore
```

Donc pour résumer, si on veut qu'une classe soit cloneable, il faut :

1. Implémenter l'interface **Cloneable**.
2. Redéfinir **clone()**.
3. Appeler **super.clone()** depuis la méthode **clone()** de la classe.
4. Intercepter les exceptions à l'intérieur de la méthode **clone()**.

Ceci produira l'effet désiré.

Le constructeur de copie

Le clonage peut sembler un processus compliqué à mettre en oeuvre. On se dit qu'il doit certainement exister une autre alternative, et souvent on envisage (surtout les programmeurs C++) de créer un constructeur spécial dont le travail est de dupliquer un objet. En C++, on l'appelle le *constructeur de copie*. Cela semble a priori la solution la plus évidente, mais en fait elle ne fonctionne pas. Voici un exemple.

```
/// appendixa:CopyConstructor.java
```

```
// Un constructeur pour copier un objet du même
// type, dans une tentative de créer une copie locale.
```

```
class FruitQualities {
    private int weight;
    private int color;
    private int firmness;
    private int ripeness;
    private int smell;
    // etc...

    FruitQualities() { // Constucteur par défaut.
        // fait des tas de choses utiles...
    }

    // D'autres constructeurs :
    // ...

    // Constructeur de copie :
    FruitQualities(FruitQualities f) {
        weight = f.weight;
        color = f.color;
        firmness = f.firmness;
        ripeness = f.ripeness;
        smell = f.smell;
        // etc...
    }
}
```

```
class Seed {
    // Membres...

    Seed() { /* Constructeur par défaut */ }

    Seed(Seed s) { /* Constructeur de copie */ }
}
```

```
class Fruit {
    private FruitQualities fq;
```

```
private int seeds;

private Seed[] s;

Fruit(FruitQualities q, int seedCount) {

    fq = q;

    seeds = seedCount;

    s = new Seed[seeds];

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

        s[i] = new Seed();

}

// Autres constructeurs :

// ...

// Constructeur de copie :

Fruit(Fruit f) {

    fq = new FruitQualities(f.fq);

    seeds = f.seeds;

    // Appelle le constructeur de copie sur toutes les Seed :

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

        s[i] = new Seed(f.s[i]);

    // D'autres activités du constructeur de copie...

}

// Pour permettre aux constructeurs dérivés (ou aux

// autres méthodes) de changer les qualités :

protected void addQualities(FruitQualities q) {

    fq = q;

}

protected FruitQualities getQualities() {

    return fq;

}

}

class Tomato extends Fruit {

    Tomato() {

        super(new FruitQualities(), 100);

    }

}
```

```
    Tomato(Tomato t) { // Constructeur de copie.

        super(t); // Transtypage pour le constructeur de copie parent.

        // D'autres activités du constructeur de copie...

    }

}

class ZebraQualities extends FruitQualities {

    private int stripedness;

    ZebraQualities() { // Constructeur par défaut.

        // Fait des tas de choses utiles...

    }

    ZebraQualities(ZebraQualities z) {

        super(z);

        stripedness = z.stripedness;

    }

}

class GreenZebra extends Tomato {

    GreenZebra() {

        addQualities(new ZebraQualities());

    }

    GreenZebra(GreenZebra g) {

        super(g); // Appelle Tomato(Tomato)

        // Restitue les bonnes qualités :

        addQualities(new ZebraQualities());

    }

    void evaluate() {

        ZebraQualities zq =

            (ZebraQualities)getQualities();

        // Utilise les qualités

        // ...

    }

}
```

```

public class CopyConstructor {

    public static void ripen(Tomato t) {

        // Utilise le « constructeur de copie » :

        t = new Tomato(t);

        System.out.println("In ripen, t is a " +

            t.getClass().getName());

    }

    public static void slice(Fruit f) {

        f = new Fruit(f); // Hmmm... est-ce que cela va marcher ?

        System.out.println("In slice, f is a " +

            f.getClass().getName());

    }

    public static void main(String[] args) {

        Tomato tomato = new Tomato();

        ripen(tomato); // OK

        slice(tomato); // OOPS!

        GreenZebra g = new GreenZebra();

        ripen(g); // OOPS!

        slice(g); // OOPS!

        g.evaluate();

    }

} ///:~

```

Ceci semble un peu étrange à première vue. Bien sûr, un fruit a des qualités, mais pourquoi ne pas mettre les données membres représentant ces qualités directement dans la classe **Fruit** ? Deux raisons à cela. La première est qu'on veut pouvoir facilement insérer ou changer les qualités. Notez que **Fruit** possède une méthode **protected addQualities()** qui permet aux classes dérivées de le faire (on pourrait croire que la démarche logique serait d'avoir un constructeur **protected** dans **Fruit** qui accepte un argument **FruitQualities**, mais les constructeurs ne sont pas hérités et ne seraient pas disponibles dans les classes dérivées). En créant une classe séparée pour la qualité des fruits, on dispose d'une plus grande flexibilité, incluant la possibilité de changer les qualités d'un objet **Fruit** pendant sa durée de vie.

La deuxième raison pour laquelle on a décidé de créer une classe **FruitQualities** est dans le cas où on veut ajouter de nouvelles qualités ou en changer le comportement via héritage ou polymorphisme. Notez que pour les **GreenZebra** (qui sont *réellement* un type de tomates - j'en ai cultivé et elles sont fabuleuses), le constructeur appelle **addQualities()** et lui passe un objet **ZebraQualities**, qui est dérivé de **FruitQualities** et peut donc être attaché à la référence **FruitQualities** de la classe de base. Bien sûr, quand **GreenZebra** utilise les **FruitQualities** il doit le transtyper dans le type correct (comme dans **evaluate()**), mais il sait que le type est toujours **ZebraQualities**.

Vous noterez aussi qu'il existe une classe **Seed**, et qu'un **Fruit** (qui par définition porte ses propres graines) [\[82\]](#) contient un tableau de **Seeds**.

Enfin, vous noterez que chaque classe dispose d'un constructeur de copie, et que chaque constructeur de copie doit s'occuper d'appeler le constructeur de copie de la classe de base et des objets membres pour réaliser une copie profonde. Le constructeur de copie est testé dans la classe **CopyConstructor**. La méthode **ripen()** accepte un argument **Tomato** et réalise une construction de copie afin de dupliquer l'objet :

```
t = new Tomato(t);
```

tandis que **slice()** accepte un objet **Fruit** plus générique et le duplique aussi :

```
f = new Fruit(f);
```

Ces deux méthodes sont testées avec différents types de **Fruit** dans **main()**. Voici la sortie produite :

```
In ripen, t is a Tomato
In slice, f is a Fruit
In ripen, t is a Tomato
In slice, f is a Fruit
```

C'est là que le problème survient. Après la construction de copie réalisée dans **slice()** sur l'objet **Tomato**, l'objet résultant n'est plus un objet **Tomato**, mais seulement un **Fruit**. Il a perdu toute sa tomaticité. De même, quand on prend une **GreenZebra**, **ripen()** et **slice()** la transforment toutes les deux en **Tomato** et **Fruit**, respectivement. La technique du constructeur de copie ne fonctionne donc pas en Java pour créer une copie locale d'un objet.

Pourquoi cela fonctionne-t-il en C++ et pas en Java ?

Le constructeur de copie est un mécanisme fondamental en C++, puisqu'il permet de créer automatiquement une copie locale d'un objet. Mais l'exemple précédent prouve que cela ne fonctionne pas en Java. Pourquoi ? En Java, toutes les entités manipulées sont des références, tandis qu'en C++ on peut manipuler soit des références sur les objets soit les objets directement. C'est le rôle du constructeur de copie en C++ : prendre un objet et permettre son passage par valeur, donc dupliquer l'objet. Cela fonctionne donc très bien en C++, mais il faut garder présent à l'esprit que ce mécanisme est à proscrire en Java.

Classes en lecture seule

Bien que la copie locale produite par **clone()** donne les résultats escomptés dans les cas appropriés, c'est un exemple où le programmeur (l'auteur de la méthode) est responsable des effets secondaires indésirables de l'aliasing. Que se passe-t-il dans le cas où on construit une bibliothèque tellement générique et utilisée qu'on ne peut supposer qu'elle sera toujours clonée aux bons endroits ? Ou alors, que se passe-t-il si on *veut* permettre l'aliasing dans un souci d'efficacité - afin de prévenir la duplication inutile d'un objet - mais qu'on n'en veut pas les effets secondaires négatifs ?

Une solution est de créer des *objets immuables* appartenant à des classes en lecture seule. On peut définir une classe telle qu'aucune méthode de la classe ne modifie l'état interne de l'objet. Dans une telle classe, l'aliasing n'a aucun impact puisqu'on peut seulement lire son état interne, donc même si plusieurs portions de code utilisent le même objet cela ne pose pas de problèmes.

Par exemple, la bibliothèque standard Java contient des classes « wrapper » pour tous les types fondamentaux. Vous avez peut-être déjà découvert que si on veut stocker un **int** dans un conteneur tel qu'une **ArrayList** (qui n'accepte que des références sur un **Object**), on peut insérer l'**int** dans la classe **Integer** de la bibliothèque standard :

```
///  
// La classe Integer ne peut pas être modifiée.  
import java.util.*;  
  
public class ImmutableInteger {  
    public static void main(String[] args) {  
        ArrayList v = new ArrayList();  
        for(int i = 0; i < 10; i++)  
            v.add(new Integer(i));  
        // Mais comment changer l'int à  
        // l'intérieur de Integer?  
    }  
} ///:~
```

La classe **Integer** (de même que toutes les classes « wrapper » pour les scalaires) implémentent l'immuabilité d'une manière simple : elles ne possèdent pas de méthodes qui permettent de modifier l'objet. Si on a besoin d'un objet qui contient un scalaire qui peut être modifié, il faut la créer soi-même. Heureusement, ceci se fait facilement :

```
///  
// Une classe wrapper modifiable.  
import java.util.*;  
  
class IntValue {  
    int n;  
    IntValue(int x) { n = x; }  
    public String toString() {  
        return Integer.toString(n);  
    }  
}
```



```

    }

    public class MutableInteger {
        public static void main(String[] args) {
            ArrayList v = new ArrayList();
            for(int i = 0; i < 10; i++)
                v.add(new IntValue(i));
            System.out.println(v);
            for(int i = 0; i < v.size(); i++)
                ((IntValue)v.get(i)).n++;
            System.out.println(v);
        }
    } //::~

```

Notez que **n** est amical pour simplifier le codage. **IntValue** peut même être encore plus simple si l'initialisation à zéro est acceptable (auquel cas on n'a plus besoin du constructeur) et qu'on n'a pas besoin d'imprimer cet objet (auquel cas on n'a pas besoin de **toString()**) :

```
class IntValue { int n; }
```

La recherche de l'élément et son transtypage par la suite est un peu lourd et maladroit, mais c'est une particularité de **ArrayList** et non de **IntValue**.

Créer des classes en lecture seule

Il est possible de créer ses propres classes en lecture seule. Voici un exemple :

```

//: appendixA:Immutable1.java
// Des objets qu'on ne peut modifier
// ne craignent pas l'aliasing.

public class Immutable1 {
    private int data;

    public Immutable1(int initVal) {
        data = initVal;
    }

    public int read() { return data; }
}

```

```

public boolean nonzero() { return data != 0; }

public Immutable1 quadruple() {
    return new Immutable1(data * 4);
}

static void f(Immutable1 i1) {
    Immutable1 quad = i1.quadruple();
    System.out.println("i1 = " + i1.read());
    System.out.println("quad = " + quad.read());
}

public static void main(String[] args) {
    Immutable1 x = new Immutable1(47);
    System.out.println("x = " + x.read());
    f(x);
    System.out.println("x = " + x.read());
}
} ///:~

```

Toutes les données sont **private**, et aucune méthode **public** ne modifie les données. En effet, la méthode qui semble modifier l'objet, **quadruple()**, crée en fait un nouvel objet **Immutable1** sans modifier l'objet original.

La méthode **f()** accepte un objet **Immutable1** et effectue diverses opérations avec, et la sortie de **main()** démontre que **x** ne subit aucun changement. Ainsi, l'objet **x** peut être aliasé autant qu'on le veut sans risque puisque la classe **Immutable1** a été conçue afin de garantir que les objets ne puissent être modifiés.

L'inconvénient de l'immuabilité

Créer une classe immuable semble à première vue une solution élégante. Cependant, dès qu'on a besoin de modifier un objet de ce nouveau type, il faut supporter le coût supplémentaire de la création d'un nouvel objet, ce qui implique aussi un passage plus fréquent du ramasse-miettes. Cela n'est pas un problème pour certaines classes, mais cela est trop coûteux pour certaines autres (telles que la classes **String**).

La solution est de créer une classe compagnon qui, elle, *peut* être modifiée. Ainsi, quand on effectue beaucoup de modifications, on peut basculer sur la classe compagnon modifiable et revenir à la classe immuable une fois qu'on en a terminé.

L'exemple ci-dessus peut être modifié pour montrer ce mécanisme :

```

///appendixa:Immutable2.java
// Une classe compagnon pour modifier
// des objets immuables.

```

```
class Mutable {
    private int data;
    public Mutable(int initVal) {
        data = initVal;
    }
    public Mutable add(int x) {
        data += x;
        return this;
    }
    public Mutable multiply(int x) {
        data *= x;
        return this;
    }
    public Immutable2 makeImmutable2() {
        return new Immutable2(data);
    }
}

public class Immutable2 {
    private int data;
    public Immutable2(int initVal) {
        data = initVal;
    }
    public int read() { return data; }
    public boolean nonzero() { return data != 0; }
    public Immutable2 add(int x) {
        return new Immutable2(data + x);
    }
    public Immutable2 multiply(int x) {
        return new Immutable2(data * x);
    }
    public Mutable makeMutable() {
        return new Mutable(data);
    }
}
```

```

    }

    public static Immutable2 modify1(Immutable2 y){
        Immutable2 val = y.add(12);
        val = val.multiply(3);
        val = val.add(11);
        val = val.multiply(2);
        return val;
    }

    // Ceci produit le même résultat :
    public static Immutable2 modify2(Immutable2 y){
        Mutable m = y.makeMutable();
        m.add(12).multiply(3).add(11).multiply(2);
        return m.makeImmutable2();
    }

    public static void main(String[] args) {
        Immutable2 i2 = new Immutable2(47);
        Immutable2 r1 = modify1(i2);
        Immutable2 r2 = modify2(i2);
        System.out.println("i2 = " + i2.read());
        System.out.println("r1 = " + r1.read());
        System.out.println("r2 = " + r2.read());
    }
} ///:~

```

Immutable2 contient des méthodes qui, comme précédemment, préservent l'immutabilité des objets en créant de nouveaux objets dès qu'une modification est demandée. Ce sont les méthodes **add()** et **multiply()**. La classe compagnon est appelée **Mutable**, et possède aussi des méthodes **add()** et **multiply()**, mais ces méthodes modifient l'objet **Mutable** au lieu d'en créer un nouveau. De plus, **Mutable** possède une méthode qui utilise ses données pour créer un objet **Immutable2** et vice-versa.

Les deux méthodes static **modify1()** et **modify2()** montrent deux approches différentes pour arriver au même résultat. Dans **modify1()**, tout est réalisé dans la classe **Immutable2** et donc quatre nouveaux objets **Immutable2** sont créés au cours du processus (et chaque fois que **val** est réassignée, l'instance précédente est récupérée par le ramasse-miettes).

Dans la méthode **modify2()**, on peut voir que la première action réalisée est de prendre l'objet **Immutable2** **y** et d'en produire une forme **Mutable** (c'est comme si on appelait **clone()** vue précédemment, mais cette fois un différent type d'objet est créé). L'objet **Mutable** est alors utilisé pour réaliser un grand nombre d'opérations *sans* nécessiter la création de nombreux objets. Puis il est retransformé en objet **Immutable2**. On n'a donc créé que

deux nouveaux objets (l'objet **Mutable** et le résultat **Immutable2**) au lieu de quatre.

Cette approche est donc sensée quand :

1. On a besoin d'objets immuables et
2. On a souvent besoin de faire beaucoup de modifications sur ces objets ou
3. Il est prohibitif de créer de nouveaux objets immuables.

Chaînes immuables

Examinons le code suivant :

```
///  
appendixa:Stringer.java  
  
public class Stringer {  
    static String upcase(String s) {  
        return s.toUpperCase();  
    }  
    public static void main(String[] args) {  
        String q = new String("howdy");  
        System.out.println(q); // howdy  
        String qq = upcase(q);  
        System.out.println(qq); // HOWDY  
        System.out.println(q); // howdy  
    }  
} ///  
~
```

Quand **q** est passé à **upcase()** c'est en fait une copie de la référence sur **q**. L'objet auquel cette référence est connectée reste dans la même localisation physique. Les références sont copiées quand elles sont passées en argument.

En regardant la définition de **upcase()**, on peut voir que la référence passée en argument porte le nom **s**, et qu'elle existe seulement pendant que le corps de **upcase()** est exécuté. Quand **upcase()** se termine, la référence locale **s** disparaît. **upcase()** renvoie le résultat, qui est la chaîne originale avec tous ses caractères en majuscules. Bien sûr, elle renvoie en fait une référence sur le résultat. Mais la référence qu'elle renvoie porte sur un nouvel objet, et l'original **q** est laissé inchangé. Comment cela se fait-il ?

Constantes implicites

Si on écrit :

```
String s = "asdf";

String x = Stringer.upcase(s);
```

est-ce qu'on veut réellement que la méthode **upcase()** *modifie* l'argument ? En général, non, car pour le lecteur du code, un argument est une information fournie à la méthode et non quelque chose qui puisse être modifié. C'est une garantie importante, car elle rend le code plus facile à lire et à comprendre.

En C++, cette garantie a été jugée suffisamment importante pour justifier l'introduction d'un mot-clef spécial, **const**, pour permettre au programmeur de s'assurer qu'une référence (pointeur ou référence en C++) ne pouvait être utilisée pour modifier l'objet original. Mais le programmeur C++ devait alors faire attention et se rappeler d'utiliser **const** de partout. Cela peut être déroutant et facile à oublier.

Surcharge de l'opérateur « + » et les StringBuffer

Les objets de la classe **String** sont conçus pour être immuables, en utilisant les techniques montrées précédemment. Lorsqu'on examine la documentation en ligne de la classe **String** (qui est résumée un peu plus loin dans cette annexe), on se rend compte que chaque méthode de la classe qui semble modifier un objet **String** crée et renvoie en fait un nouvel objet **String** contenant la modification ; la **String** originale reste inchangée. Il n'y a donc pas de fonctionnalité en Java telle que le **const** du C++ pour s'assurer de l'immuabilité des objets lors de la compilation. Si on en a besoin, il faut l'implémenter soi-même, comme le fait la classe **String**.

Comme les objets **String** sont immuables, on peut aliaser un objet **String** autant de fois qu'on veut. Puisqu'il est en lecture seule, une référence ne peut rien modifier qui puisse affecter les autres références. Un objet en lecture seule résoud donc élégamment le problème de l'aliasing.

Il semble également possible de gérer tous les cas dans lesquels on a besoin de modifier un objet en créant une nouvelle version de l'objet avec les modifications, comme le fait la classe **String**. Ce n'est toutefois pas efficace pour certaines opérations. C'est le cas de l'opérateur « + » qui a été surchargé pour les objets **String**. Surchargé veut dire qu'un sens supplémentaire lui a été attribué s'il est utilisé avec une classe particulière (les opérateurs « + » et « += » pour les **String** sont les seuls opérateurs surchargés en Java, et Java ne permet pas au programmeur d'en surcharger d'autres) [\[83\]](#).

Quand il est utilisé avec des objets **String**, l'opérateur « + » permet de concaténer des **String** :

```
String s = "abc" + foo + "def" + Integer.toString(47);
```

On peut imaginer comment ceci *pourrait* être implémenté : la **String** « abc » pourrait avoir une méthode **append()** qui créerait un nouvel objet **String** contenant la chaîne « abc » concaténée avec le contenu de **foo**. Le nouvel objet **String** devrait alors créer une nouvelle **String** à laquelle serait ajoutée la chaîne « def », et ainsi de suite.

Cela marcherait bien sûr, mais cela nécessiterait la création d'un grand nombre d'objets **String** rien que pour assembler la nouvelle **String**, et donc on se retrouverait avec un ensemble d'objets **String** intermédiaires qui devraient être réclamés par le ramasse-miettes. Je suspecte les concepteurs de Java d'avoir tenté cette approche en premier (une leçon de la conception de logiciel - on ne sait réellement rien d'un système tant qu'on n'a pas essayé de le coder et qu'on ne dispose pas d'un système fonctionnel). Je suspecte aussi qu'ils ont découvert que les performances étaient inacceptables.

La solution consiste en une classe compagnon modifiable similaire à celle exposée précédemment. Pour la classe **String**, cette classe compagnon est appelée **StringBuffer**, et le compilateur crée automatiquement un objet **StringBuffer** pour évaluer certaines expressions, en particulier quand les opérateurs surchargés `+` et `+=` sont utilisés avec des objets **String**. L'exemple suivant montre ce qui se passe :

```

//: appendixA:ImmutableStrings.java

// Démonstration de la classe StringBuffer.

public class ImmutableStrings {

    public static void main(String[] args) {

        String foo = "foo";

        String s = "abc" + foo +

            "def" + Integer.toString(47);

        System.out.println(s);

        // L'« équivalent » en utilisant la classe StringBuffer :

        StringBuffer sb =

            new StringBuffer("abc"); // Crée une String!

        sb.append(foo);

        sb.append("def"); // Crée une String!

        sb.append(Integer.toString(47));

        System.out.println(sb);

    }

} ///:~

```

Durant la création de la **String** `s`, le compilateur fait à peu près l'équivalent du code suivant qui utilise `sb` : une **StringBuffer** est créée et `append()` est appelée pour ajouter les nouveaux caractères directement dans l'objet **StringBuffer** (plutôt que de créer de nouvelles copies à chaque fois). Bien que ceci soit beaucoup plus efficace, il est bon de noter que chaque fois qu'on crée une chaîne de caractères quotée telle que « **abc** » ou « **def** », le compilateur les transforme en objets **String**. Il peut donc y avoir plus d'objets créés que ce qu'on pourrait croire, en dépit de l'efficacité permise par la classe **StringBuffer**.

Les classes String et StringBuffer

Voici un aperçu des méthodes disponibles pour les deux classes **String** et **StringBuffer** afin de se faire une idée de la manière dont elles interagissent. Ces tables ne référencent pas toutes les méthodes disponibles, mais seulement celles qui sont importantes pour notre sujet. Les méthodes surchargées sont résumées sur une ligne.

Méthode	Arguments, Surcharges	Utilisation
Constructeur	Surchargés : Défaut, String , StringBuffer , tableau de char , tableau	Création d'objets String .

	de byte .	
length()		Nombre de caractères dans la String .
charAt()	int Index	Le n-ième caractère dans la String .
getChars(), getBytes()	Le début et la fin de la zone à copier, le tableau dans lequel effectuer la copie, un index dans le tableau destination.	Copie des chars ou des bytes dans un tableau externe.
toCharArray()		Produit un char[] contenant les caractères de la String .
equals(), equals-IgnoreCase()	Une String avec laquelle comparer.	Un test d'égalité sur le contenu de deux String .
compareTo()	Une String avec laquelle comparer.	Le résultat est négatif, zéro ou positif suivant la comparaison lexicographique de la String et l'argument. Majuscules et minuscules sont différenciées !
regionMatches()	Déplacement dans la String , une autre String ainsi que son déplacement et la longueur à comparer. Surchargé : ajoute l'insensibilisation à la casse.	Résultat boolean indiquant si les régions correspondent.
startsWith()	La String avec laquelle la String pourrait débuter. Surchargé : ajoute un déplacement pour l'argument.	Résultat boolean indiquant si la String débute avec l'argument.
endsWith()	La String qui pourrait être un suffixe de cette String .	Résultat boolean indiquant si l'argument est un suffixe de la String .
indexOf(), lastIndexOf()	Surchargés : char , char et index de départ, String , String , et index de départ.	Renvoie -1 si l'argument n'est pas trouvé dans la String , sinon renvoie l'index où l'argument commence. lastIndexOf() recherche en partant de la fin.
substring()	Surchargé : index de départ, index de départ et index de fin.	Renvoie un nouvel objet String contenant l'ensemble des caractères spécifiés.
concat()	La String à concaténer.	Renvoie un nouvel objet String contenant les caractères de l'objet String initial suivis des caractères de l'argument.
replace()	L'ancien caractère à rechercher, le nouveau caractère avec lequel le remplacer.	Renvoie un nouvel objet String avec les remplacements effectués. Utilise l'ancienne String si aucune correspondance n'est trouvée.
toLowerCase(), toUpperCase()		Renvoie un nouvel objet String avec la casse de tous les caractères modifiée. Utilise l'ancienne String si aucun changement n'a été effectué.
trim()		Renvoie un nouvel objet String avec les espaces enlevés à chaque extrémité. Utilise l'ancienne String si aucun

		changement n'a été effectué.
valueOf()	Surchargés : Object , char[] , char[] et déplacement et compte, boolean , char , int , long , float , double .	Renvoie une String contenant une représentation sous forme de chaîne de l'argument.
intern()		Produit une et une seule référence String pour chaque séquence unique de caractères.

On peut voir que chaque méthode de la classe **String** renvoie un nouvel objet **String** quand il est nécessaire d'en modifier le contenu. Remarquez aussi que si le contenu n'a pas besoin d'être modifié la méthode renverra juste une référence sur la **String** originale. Cela permet d'économiser sur le stockage et le surcoût d'une création.

Et voici la classe **StringBuffer** :

Méthode	Arguments, Surcharges	Utilisation
Constructeur	Surchargés : défaut, longueur du buffer à créer, String à partir de laquelle créer.	Crée un nouvel objet StringBuffer .
toString()		Crée un objet String à partir de l'objet StringBuffer .
length()		Nombre de caractères dans l'objet StringBuffer .
capacity()		Renvoie l'espace actuellement alloué.
ensure-Capacity()	Entier indiquant la capacité désirée.	Fait que l'objet StringBuffer alloue au minimum un certain espace.
setLength()	Entier indiquant la nouvelle longueur de la chaîne de caractères dans le buffer.	Tronque ou accroît la chaîne de caractères. Si accroissement, complète avec des caractères nulls.
charAt()	Entier indiquant la localisation de l'élément désiré.	Renvoie le char à l'endroit spécifié dans le buffer.
setCharAt()	Entier indiquant la localisation de l'élément désiré et la nouvelle valeur char de cet élément.	Modifie la valeur à un endroit précis.
getChars()	Le début et la fin à partir de laquelle copier, le tableau dans lequel copier, un index dans le tableau de destination.	Copie des chars dans un tableau extérieur. Il n'existe pas de getBytes() comme dans la classe String .
append()	Surchargés : Object , String , char[] , char[] avec déplacement et longueur, boolean , char , int , long , float , double .	L'argument est converti en chaîne de caractères et concaténé à la fin du buffer courant, en augmentant la taille du buffer si nécessaire.
insert()	Surchargés, chacun avec un premier argument contenant le déplacement auquel on débute l'insertion : Object , String , char[] , boolean , char , int , long , float , double .	Le deuxième argument est converti en chaîne de caractères et inséré dans le buffer courant au déplacement spécifié. La taille du buffer est augmentée si nécessaire.
reverse()		L'ordre des caractères du buffer est inversé.

La méthode la plus fréquemment utilisée est **append()**, qui est utilisée par le compilateur quand il évalue des expressions **String** contenant les opérateurs « + » et « += ». La méthode **insert()** a une forme similaire, et les deux méthodes réalisent des modifications significatives sur le buffer au lieu de créer de nouveaux objets.

Les Strings sont spéciales

La classe **String** n'est donc pas une simple classe dans Java. La classe **String** est spéciale à bien des égards, en particulier parce que c'est une classe intégrée et fondamentale dans Java. Il y a aussi le fait qu'une chaîne de caractères entre quotes est convertie en **String** par le compilateur et les opérateurs + et +=. Dans cette annexe vous avez pu voir d'autres spécificités : l'immutabilité précautionneusement assurée par la classe compagnon **StringBuffer** et d'autres particularités magiques du compilateur.

Résumé

Puisque dans Java tout est référence, et puisque chaque objet est créé dans le segment et réclamée par le ramasse-miettes seulement quand il n'est plus utilisé, la façon de manipuler les objets change, spécialement lors du passage et du retour d'objets. Par exemple, en C ou en C++, si on veut initialiser un endroit de stockage dans une méthode, il faut demander à l'utilisateur de passer l'adresse de cet endroit de stockage à la méthode, ou alors il faut se mettre d'accord sur qui a la responsabilité de détruire cet espace de stockage. L'interface et la compréhension de telles méthodes sont donc bien plus compliquées. Mais en Java, on n'a pas à se soucier de savoir si un objet existera toujours lorsqu'on en aura besoin, puisque tout est déjà géré pour nous. On peut ne créer un objet que quand on en a besoin, et pas avant, sans se soucier de déléguer les responsabilités sur cet objet : on passe simplement une référence. Quelquefois la simplification que cela engendre passe inaperçue, d'autres fois elle est flagrante.

Les inconvénients de cette magie sous-jacente sont doubles :

1. Il y a toujours une pénalité d'efficacité engendrée pour cette gestion supplémentaire de la mémoire (bien qu'elle puisse être relativement faible), et il y a toujours une petite incertitude sur le temps d'exécution (puisque le ramasse-miettes peut être obligé d'entrer en action si la quantité de mémoire disponible n'est pas suffisante). Pour la plupart des applications, les bénéfices compensent largement les inconvénients, et les parties de l'application critiques quant au temps peuvent être écrites en utilisant des méthodes **native** (voir l'annexe B).
2. Aliasing : on peut se retrouver accidentellement avec deux références sur le même objet, ce qui est un problème si les deux références sont supposées pointer sur deux objets *distincts*. Cela demande de faire un peu plus attention, et, si nécessaire, **clone()** l'objet pour empêcher toute modification intempestive de l'objet par l'autre référence. Cependant on peut supporter l'aliasing pour l'efficacité qu'il procure en évitant cet inconvénient en créant des objets immuables dont les opérations renvoient un nouvel objet du même type ou d'un type différent, mais ne changent pas l'objet original afin que toute référence liée à cet objet ne voie pas de changements.

Certaines personnes insinuent que la conception du clonage a été bâclée en Java, et pour ne pas s'embêter avec, implémentent leur propre version du clonage [84], sans jamais appeler la méthode **Object.clone()**, éliminant ainsi le besoin d'implémenter **Cloneable** et d'intercepter l'exception **CloneNotSupportedException**. Ceci est certainement une approche raisonnable et comme **clone()** est très peu implémentée dans la bibliothèque standard Java, elle est aussi relativement sûre. Mais tant qu'on n'appelle pas **Object.clone()** on n'a pas besoin d'implémenter **Cloneable** ou d'intercepter l'exception, donc cela semble acceptable aussi.

Exercises

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

1. Démontrer un second niveau d'aliasing. Créer une méthode qui crée une référence sur un objet mais ne modifie pas cet objet. Par contre, la méthode appelle une seconde méthode, lui passant la référence, et cette seconde méthode modifie l'objet.
2. Créer une classe **myString** contenant un objet **String** qu'on initialisera dans le constructeur en utilisant l'argument du constructeur. Ajouter une méthode **toString()** et une méthode **concatenate()** qui ajoute un objet **String** à la chaîne interne. Implémenter **clone()** dans **myString**. Créer deux méthodes **static** acceptant chacune une référence **myString x** comme argument et appelant **x.concatenat("test")**, mais la seconde méthode appelle **clone()** d'abord. Tester les deux méthodes et montrer les différents effets.
3. Créer une classe **Battery** contenant un **int** qui est un numéro de batterie (comme identifiant unique). La rendre cloneable et lui donner une méthode **toString()**. Créer maintenant une classe **Toy** qui contienne un tableau de **Battery** et une méthode **toString()** qui affiche toutes les **Battery**. Ecrire une méthode **clone()** pour la classe **Toy** qui clone automatiquement tous ses objets **Battery**. Tester cette méthode en clonant **Toy** et en affichant le résultat.
4. Changer **CheckCloneable.java** afin que toutes les méthodes **clone()** interceptent l'exception **CloneNotSupportedException** plutôt que de la faire suivre à l'appelant.
5. En utilisant la technique de la classe compagnon modifiable, créer une classe immuable contenant un **int**, un **double** et un tableau de **char**.
6. Modifier **Compete.java** pour ajouter de nouveaux objets membres aux classes **Thing2** et **Thing4**, et voyez si vous pouvez déterminer comment le minutage varie avec la complexité - si c'est une simple relation linéaire ou si cela semble plus compliqué.
7. En reprenant **Snake.java**, créer une version du serpent qui supporte la copie profonde.
8. Hériter d'une **ArrayList** et faire que sa méthode **clone()** réalise une opération de copie profonde.

[79] En C, qui généralement gère des données de petite taille, le passage d'arguments par défaut se fait par valeur. C++ a dû suivre cette norme, mais avec les objets, le passage par valeur n'est pas la façon de faire la plus efficace. De plus, coder une classe afin qu'elle supporte le passage par valeur en C++ est un gros casse-tête.

[80] Ce n'est pas l'orthographe correcte du mot clonable (N.d.T. : en français comme en anglais), mais c'est de cette manière qu'il est utilisé dans la bibliothèque Java ; j'ai donc décidé de l'utiliser ainsi ici aussi, dans l'espoir de réduire la confusion.

[81] On peut apparemment créer un simple contre-exemple de cette affirmation, comme ceci :

```
public class Cloneit implements Cloneable {
    public static void main (String[] args)
        throws CloneNotSupportedException {
        Cloneit a = new Cloneit();
        Cloneit b = (Cloneit)a.clone();
    }
}
```

Toutefois, ceci ne marche que parce que **main()** est une méthode de **Cloneit** et donc a la permission d'appeler la méthode **protected**

clone() de la classe de base. Si on l'appelle depuis une autre classe, la compilation générera des erreurs.

[82] L'avocat excepté, qui a été reclassifié en « aliment gras ».

[83] C++ permet au programmeur de surcharger les opérateurs comme il l'entend. Comme ceci se révèle souvent être un processus compliqué (voir le Chapitre 10 de *Thinking in C++*, 2nd edition, Prentice-Hall, 2000), les concepteurs de Java ont jugé une telle fonctionnalité « non souhaitable » qui ne devait pas donc pas être intégrée dans Java. Elle n'était apparemment pas si indésirable que cela, puisqu'eux-mêmes l'ont finalement utilisée, et ironiquement, la surcharge d'opérateurs devrait être bien plus facile à utiliser en Java qu'en C++. On peut le voir dans Python (voir www.python.org) qui dispose d'un ramasse-miettes et d'un mécanisme de surcharge d'opérateurs très simple à mettre en oeuvre.

[84] Doug Lea, qui m'a été d'une aide précieuse sur ce sujet, m'a suggéré ceci, me disant qu'il créait simplement une fonction **duplicate()** pour chaque classe.