

25.04.2001 - version 1.4 [Armel]

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

13.04.2001 - version 1.3 [Florence DEFAIX]

- Après corrections proposées par Jérôme QUELIN et J-P VIDAL.

4: Initialisation & Nettoyage

Depuis le début de la révolution informatique, la programmation «sans garde-fou» est la principale cause des coûts de développement excessifs.

L'*initialisation* et la *libération* d'éléments sont deux problèmes majeurs. De nombreux bogues en C surviennent lorsque le programmeur oublie d'initialiser une variable. L'utilisation de blibliothèques augmente ce risque car les utilisateurs ne savent pas toujours comment initialiser certains composants, ni même qu'ils le doivent. La phase de nettoyage ou libération pose problème dans la mesure où il est très facile d'oublier l'existence d'un élément dont on n'a plus besoin, car justement il ne nous intéresse plus. Dans ce cas, certaines ressources utilisées par un élément oublié sont conservées. Ce phénomène peut entraîner un manque de ressources (dans la majorité des cas, un manque de mémoire).

C++ a introduit la notion de *constructeur*, une méthode appelée automatiquement à la création d'un objet. Java utilise aussi les constructeurs, associé à un ramasse-miettes qui libère les ressources mémoire lorsqu'elles ne sont plus utilisées. Ce chapitre décrit les concepts d'initialisation et de libération, ainsi que leur support dans Java.

Garantie d'initialisation grâce au constructeur

Pour chaque classe il serait possible de créer une méthode **initialise()**. Ce nom inciterait à exécuter la méthode avant d'utiliser l'objet. Malheureusement ce serait à l'utilisateur de se souvenir d'appeler cette méthode pour chaque instance. En Java, le concepteur d'une classe peut garantir son initialisation grâce à une méthode spéciale que l'on dénomme *constructeur*. Quand une classe possède un constructeur, Java l'appelle automatiquement à toute création d'objets, avant qu'ils ne puissent être utilisés. L'initialisation est donc bien garantie.

Le premier problème consiste à trouver un nom pour cette méthode ce qui entraîne deux nouveaux problèmes. Tout d'abord il pourrait y avoir un conflit avec le nom d'un attribut. Ensuite c'est à la compilation que l'appel du constructeur est vérifié. Il faut donc que le compilateur puisse décider du nom du constructeur. La solution de C++ paraît la plus simple et la plus logique, elle est donc aussi utilisée en Java. Il faut donc donner au constructeur le nom de sa classe. Il semble naturel qu'une telle méthode soit en charge de l'initialisation de la classe.

Voici une classe avec un constructeur :

```
///  
//  
  
class Rock {
```

```
Rock() { // Ceci est un constructeur
    System.out.println("Creating Rock");
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
} ///:~
```

Quand un objet est créé :

```
new Rock();
```

de l'espace mémoire est alloué et le constructeur est appelé. L'objet sera obligatoirement initialisé avant qu'il ne puisse être manipulé.

Notez que la convention de nommage qui impose une minuscule pour la première lettre des noms de méthode ne s'applique pas aux constructeurs, leur nom devant *exactement* coïncider avec celui de la classe.

Comme les autres méthodes, un constructeur peut prendre des paramètres. Cela permet de préciser *comment* l'objet va être créé. Notre premier exemple peut facilement être modifié pour que le constructeur prenne un unique paramètre :

```
//: c04:SimpleConstructor2.java
// Les constructeurs peuvent prendre des paramètres.
```

```
class Rock2 {
    Rock2(int i) {
        System.out.println(
            "Creating Rock number " + i);
    }
}

public class SimpleConstructor2 {
    public static void main(String[] args) {
```

```
        for(int i = 0; i < 10; i++)

            new Rock2(i);

    }

} ///:~
```

Les paramètres des constructeurs permettent de personnaliser la création des objets. Par exemple, si la classe **Tree** (arbre) a un constructeur avec un paramètre de type **int** qui détermine la hauteur de l'arbre, un objet **Tree** se crée de la façon suivante :

```
Tree t = new Tree(12); // arbre de 12 pieds
```

De plus, si **Tree(int)** est le seul constructeur, le compilateur ne permettra pas de créer un objet **Tree** d'une autre façon.

La notion de constructeur élimine toute une catégorie d'erreurs et rend plus aisée la lecture du code. Dans le fragment de code précédent, par exemple, il n'y a pas d'appel explicite à une certaine méthode **initialise()** qui serait conceptuellement séparée de la définition. En Java, définition et initialisation sont des concepts unifiés - il est impossible d'avoir l'un sans l'autre.

Un constructeur est une méthode très spéciale de par le fait qu'elle n'a pas de valeur de retour. Cela n'a absolument rien à voir avec le type de retour **void**, qui signifie qu'une méthode ne renvoie rien mais qu'il aurait tout à fait été possible de lui faire renvoyer autre chose. Les constructeurs ne retournent rien et on n'a pas le choix. S'il y avait une valeur de retour, et si l'on pouvait choisir son type, le compilateur devrait trouver une utilisation à cette valeur.

Surcharge de méthodes

L'un des points les plus importants de tout langage de programmation est le nommage. Créer un objet revient à donner un nom à un emplacement mémoire. Une méthode est un nom d'action. En utilisant des noms pour décrire un système, on simplifie la lecture et la modification des programmes. Cela s'apparente à l'écriture en prose dont le but est de communiquer avec le lecteur.

On se réfère à tous les objets et méthodes en utilisant leurs noms. Des noms bien choisis rendent la compréhension du code plus aisée, tant pour le développeur que pour les relecteurs.

Les difficultés commencent lorsque l'on essaie d'exprimer les nuances subtiles du langage humain dans un langage de programmation. Très souvent, un même mot a plusieurs sens, on parle de *surcharge*. Cette notion est très pratique pour exprimer les différences triviales de sens. On dit « laver la chemise », « laver la voiture » et « laver le chien ». Cela paraîtrait absurde d'être obligé de dire « laverChemise la chemise », « laverVoiture la voiture » et « laverChien le chien » pour que l'auditoire puisse faire la distinction entre ces actions. La plupart des langages humains sont redondants à tel point que même sans entendre tous les mots, il est toujours possible de comprendre le sens d'une phrase. Nous n'avons aucunement besoin d'identifiants uniques, le sens peut être déduit du contexte.

La plupart des langages de programmation (C en particulier) imposent un nom unique pour chaque fonction. Ils ne permettent pas d'appeler une fonction **affiche()** pour afficher des entiers et une autre appelée **affiche()** pour afficher des flottants, chaque fonction doit avoir un nom unique.

En Java (et en C++), un autre facteur impose la surcharge de noms de méthodes : les constructeurs. Comme le nom d'un constructeur est déterminé par le nom de la classe, il ne peut y avoir qu'un seul nom de constructeur. Mais que se passe-t-il quand on veut créer un objet de différentes façons ? Par exemple, supposons que l'on construise une classe qui peut s'initialiser de façon standard ou en lisant des informations depuis un fichier. Nous avons alors besoin de deux constructeurs, l'un ne prenant pas de paramètre (le constructeur *par défaut*, aussi appelé le constructeur *sans paramètre / no-arg*), et un autre prenant une **Chaîne / String** comme paramètre, qui représente le nom du fichier depuis lequel on souhaite initialiser l'objet. Tous les deux sont des constructeurs, ils doivent donc avoir le même nom, le nom de la classe. Cela montre que la *surcharge de méthode* est essentielle pour utiliser le même nom de méthode pour des utilisations sur différents types de paramètres. Et si la surcharge de méthode est obligatoire pour les constructeurs, elle est aussi très pratique pour les méthodes ordinaires.

L'exemple suivant montre à la fois une surcharge de constructeur et une surcharge de méthode ordinaire :

```
///  
// c04:Overloading.java  
// Exemple de surcharge de constructeur  
// et de méthode ordinaire.  
import java.util.*;  
  
class Tree {  
    int height;  
    Tree() {  
        prt("Planting a seedling"); // Planter une jeune pousse  
        height = 0;  
    }  
    Tree(int i) {  
        prt("Creating new Tree that is " // Création d'un Arbre  
            + i + " feet tall"); // de i pieds de haut  
        height = i;  
    }  
    void info() {  
        prt("Tree is " + height // L'arbre mesure x pieds  
            + " feet tall");  
    }  
    void info(String s) {  
        prt(s + ": Tree is " // valeur de s : L'arbre mesure x pieds  
            + height + " feet tall");  
    }  
}
```

```

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

public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);

            t.info();

            t.info("overloaded method");
        }

        // constructeur surchargé :

        new Tree();
    }
} ///:~

```

Un objet **Tree** peut être créé soit en tant que jeune pousse, sans fournir de paramètre, soit en tant que plante poussée en pépinière, en donnant une hauteur initiale. Pour permettre ceci, il y a deux constructeurs, l'un ne prend pas de paramètre (on appelle les constructeurs sans paramètre des *constructeurs par défaut* [\[27\]](#)) et un deuxième qui prend la hauteur initiale de l'arbre.

Il est aussi possible d'appeler la méthode **info()** de plusieurs façons. Par exemple, avec un paramètre **String** si un message supplémentaire est désiré, ou sans paramètre lorsqu'il n'y a rien d'autre à dire. Cela paraîtrait étrange de donner deux noms distincts à ce qui est manifestement le même concept. Heureusement, la surcharge de méthode permet l'utilisation du même nom pour les deux.

Différencier les méthodes surchargées

Quand deux méthodes ont le même nom, comment Java peut-il décider quelle méthode est demandée ? Il y a une règle toute simple : chaque méthode surchargée doit prendre une liste unique de types de paramètres.

Lorsqu'on y pense, cela paraît tout à fait sensé : comment le développeur lui-même pourrait-il choisir entre deux méthodes du même nom, autrement que par le type des paramètres ?

Une différence dans l'ordre des paramètres est suffisante pour distinguer deux méthodes (cette approche n'est généralement pas utilisée car elle donne du code difficile à maintenir.) :

```

///: c04:OverloadingOrder.java

// Surcharge basée sur l'ordre

// des paramètres.

```

```
public class OverloadingOrder {  
    static void print(String s, int i) {  
        System.out.println(  
            "String: " + s +  
            ", int: " + i);  
    }  
    static void print(int i, String s) {  
        System.out.println(  
            "int: " + i +  
            ", String: " + s);  
    }  
    public static void main(String[] args) {  
        print("String first", 11);  
        print(99, "Int first");  
    }  
} ///:~
```

Les deux méthodes **print()** ont les mêmes paramètres, mais dans un ordre différent, et c'est ce qui les différencie.

Surcharge avec types de base

Un type de base peut être promu automatiquement depuis un type plus petit vers un plus grand ; ceci peut devenir déconcertant dans certains cas de surcharge. L'exemple suivant montre ce qui se passe lorsqu'un type de base est passé à une méthode surchargée :

```
///: c04:PrimitiveOverloading.java  
// Promotion des types de base et surcharge.  
  
public class PrimitiveOverloading {  
    // boolean ne peut pas être converti automatiquement  
    static void prt(String s) {  
        System.out.println(s);  
    }  
  
    void fl(char x) { prt("fl(char)"); }
```

```
void f1(byte x) { prt("f1(byte)"); }
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }

void f2(byte x) { prt("f2(byte)"); }
void f2(short x) { prt("f2(short)"); }
void f2(int x) { prt("f2(int)"); }
void f2(long x) { prt("f2(long)"); }
void f2(float x) { prt("f2(float)"); }
void f2(double x) { prt("f2(double)"); }

void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }
void f3(float x) { prt("f3(float)"); }
void f3(double x) { prt("f3(double)"); }

void f4(int x) { prt("f4(int)"); }
void f4(long x) { prt("f4(long)"); }
void f4(float x) { prt("f4(float)"); }
void f4(double x) { prt("f4(double)"); }

void f5(long x) { prt("f5(long)"); }
void f5(float x) { prt("f5(float)"); }
void f5(double x) { prt("f5(double)"); }

void f6(float x) { prt("f6(float)"); }
void f6(double x) { prt("f6(double)"); }

void f7(double x) { prt("f7(double)"); }
```

```
void testConstVal() {
    prt("Testing with 5");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5);
}

void testChar() {
    char x = 'x';
    prt("char argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testByte() {
    byte x = 0;
    prt("byte argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testShort() {
    short x = 0;
    prt("short argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testInt() {
    int x = 0;
    prt("int argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testLong() {
    long x = 0;
    prt("long argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

void testFloat() {
    float x = 0;
    prt("float argument:");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}
```



```

void testDouble() {
    double x = 0;

    prt("double argument:");

    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
}

public static void main(String[] args) {
    PrimitiveOverloading p =
        new PrimitiveOverloading();

    p.testConstVal();
    p.testChar();
    p.testByte();
    p.testShort();
    p.testInt();
    p.testLong();
    p.testFloat();
    p.testDouble();
}
} ///:~

```

En regardant la sortie du programme, on voit que la constante 5 est considérée comme un **int**. Lorsqu'une méthode surchargée utilisant un **int** est disponible, elle est utilisée. Dans tous les autres cas, si un type de données est plus petit que l'argument de la méthode, le type est promu. **char** est légèrement différent, comme il ne trouve pas une correspondance exacte, il est promu vers un **int**.

Qu'arrive-t'il lorsque le paramètre est *plus grand* que celui attendu par la méthode surchargée ? Une modification du programme précédent donne la réponse :

```

///: c04:Demotion.java

// Types de base déchus et surcharge.

public class Demotion {
    static void prt(String s) {
        System.out.println(s);
    }

    void f1(char x) { prt("f1(char)"); }
    void f1(byte x) { prt("f1(byte)"); }
}

```

```
void f1(short x) { prt("f1(short)"); }
void f1(int x) { prt("f1(int)"); }
void f1(long x) { prt("f1(long)"); }
void f1(float x) { prt("f1(float)"); }
void f1(double x) { prt("f1(double)"); }

void f2(char x) { prt("f2(char)"); }
void f2(byte x) { prt("f2(byte)"); }
void f2(short x) { prt("f2(short)"); }
void f2(int x) { prt("f2(int)"); }
void f2(long x) { prt("f2(long)"); }
void f2(float x) { prt("f2(float)"); }

void f3(char x) { prt("f3(char)"); }
void f3(byte x) { prt("f3(byte)"); }
void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }

void f4(char x) { prt("f4(char)"); }
void f4(byte x) { prt("f4(byte)"); }
void f4(short x) { prt("f4(short)"); }
void f4(int x) { prt("f4(int)"); }

void f5(char x) { prt("f5(char)"); }
void f5(byte x) { prt("f5(byte)"); }
void f5(short x) { prt("f5(short)"); }

void f6(char x) { prt("f6(char)"); }
void f6(byte x) { prt("f6(byte)"); }

void f7(char x) { prt("f7(char)"); }

void testDouble() {
```

```

    double x = 0;

    prt("double argument:");

    f1(x);f2((float)x);f3((long)x);f4((int)x);

    f5((short)x);f6((byte)x);f7((char)x);
}

public static void main(String[] args) {

    Demotion p = new Demotion();

    p.testDouble();

}

} ///:~

```

Ici, les méthodes prennent des types de base plus restreints. Si les paramètres sont d'un type plus grand, il faut les *caster* (*convertir*) vers le type requis en utilisant le nom du type entre parenthèses. Sinon, le compilateur donnera un message d'erreur.

Il est important de noter qu'il s'agit d'une *conversion vers un type plus petit*, ce qui signifie que des informations peuvent être perdues pendant la conversion. C'est d'ailleurs pour cette raison que le compilateur force une conversion explicite.

Surcharge sur la valeur de retour

Il est fréquent de se demander «Pourquoi seulement les noms de classes et la liste des paramètres des méthodes ? Pourquoi ne pas aussi distinguer entre deux méthodes en se basant sur leur type de retour ?» Par exemple, ces deux méthodes, qui ont le même nom et les mêmes arguments, peuvent facilement être distinguées l'une de l'autre :

```

void f() {}

int f() {}

```

Cela fonctionne bien lorsque le compilateur peut déterminer le sens sans équivoque depuis le contexte, comme dans **int x = f()**. Par contre, on peut utiliser une méthode et ignorer sa valeur de retour. On se réfère souvent à cette action comme *appeler une méthode pour ses effets de bord* puisqu'on ne s'intéresse pas à la valeur de retour mais aux autres effets que cet appel de méthode génère. Donc, si on appelle la méthode comme suit :

```

f();

```

Comment Java peut-il déterminer quelle méthode **f()** doit être exécutée ? Et comment quelqu'un lisant ce code pourrait-il le savoir ? A cause de ce genre de difficultés, il est impossible d'utiliser la valeur de retour pour différencier deux méthodes Java surchargées.

Constructeurs par défaut

Comme mentionné précédemment, un constructeur par défaut (c.a.d un constructeur «no-arg») est un

constructeur sans argument, utilisé pour créer des « objets de base ». Si une classe est créée sans constructeur, le compilateur crée automatiquement un constructeur par défaut. Par exemple :

```
///  
c04:DefaultConstructor.java  
  
class Bird {  
    int i;  
}  
  
public class DefaultConstructor {  
    public static void main(String[] args) {  
        Bird nc = new Bird(); // défaut !  
    }  
} ///:~
```

La ligne

```
new Bird();
```

crée un nouvel objet et appelle le constructeur par défaut, même s'il n'était pas défini explicitement. Sans lui, il n'y aurait pas de méthode à appeler pour créer cet objet. Par contre, si au moins un constructeur est défini (avec ou sans argument), le compilateur n'en synthétisera *pas* un :

```
class Bush {  
    Bush(int i) {}  
    Bush(double d) {}  
}
```

Maintenant si on écrit :

```
new Bush();
```

le compilateur donnera une erreur indiquant qu'aucun constructeur ne correspond. C'est comme si lorsqu'aucun constructeur n'est fourni, le compilateur dit «Il faut un constructeur, je vais en créer un.» Alors que s'il existe un constructeur, le compilateur dit «Il y a un constructeur donc le développeur sait se qu'il fait; s'il n'a pas défini de constructeur par défaut c'est qu'il ne désirait pas qu'il y en ait un.»

Le mot-clé this

Lorsqu'il existe deux objets **a** et **b** du même type , il est intéressant de se demander comment on peut appeler

une méthode **f()** sur ces deux objets :

```
class Banana { void f(int i) { /* ... */ } }

Banana a = new Banana(), b = new Banana();

a.f(1);

b.f(2);
```

S'il y a une unique méthode **f()**, comment cette méthode peut-elle savoir si elle a été appelée sur l'objet **a** ou **b** ?

Pour permettre au développeur d'écrire le code dans une syntaxe pratique et orienté objet dans laquelle on «envoie un message vers un objet,» le compilateur effectue un travail secret pour le développeur. Il y a un premier paramètre caché passé à la méthode **f()**, et ce paramètre est une référence vers l'objet en train d'être manipulé. Les deux appels de méthode précédents correspondent donc à ceci :

```
Banana.f(a,1);

Banana.f(b,2);
```

Ce travail est interne et il est impossible d'écrire des expressions de ce type directement en espérant que le compilateur les acceptera, mais cela donne une idée de ce qui se passe.

Supposons maintenant que l'on est à l'intérieur d'une méthode et que l'on désire obtenir une référence sur l'objet courant. Comme cette référence est passée en tant que paramètre *caché* par le compilateur, il n'y a pas d'identificateur pour elle. Cette pour cette raison que le mot clé **this** existe. **this** - qui ne peut être utilisé qu'à l'intérieur d'une méthode - est une référence sur l'objet pour lequel cette méthode à été appelée. On peut utiliser cette référence comme tout autre référence vers un objet. Il n'est toutefois pas nécessaire d'utiliser **this** pour appeler une méthode de la classe courante depuis une autre méthode de la classe courante ; il suffit d'appeler cette méthode. La référence **this** est automatiquement utilisée pour l'autre méthode. On peut écrire :

```
class Apricot {

    void pick() { /* ... */ }

    void pit() { pick(); /* ... */ }

}
```

A l'intérieur de **pit()**, on *pourrait* écrire **this.pick()** mais ce n'est pas nécessaire. Le compilateur le fait automatiquement pour le développeur. Le mot-clé **this** est uniquement utilisé pour les cas spéciaux dans lesquels on doit utiliser explicitement une référence sur l'objet courant. Par exemple, il est couramment utilisé en association avec **return** quand on désire renvoyer une référence sur l'objet courant :

```
//: c04:Leaf.java

// Utilisation simple du mot-clé "this".

public class Leaf {
```

```

    int i = 0;

    Leaf increment() {

        i++;

        return this;
    }

    void print() {

        System.out.println("i = " + i);
    }

    public static void main(String[] args) {

        Leaf x = new Leaf();

        x.increment().increment().increment().print();
    }
} ///:~

```

Puisque **increment()** renvoie une référence vers l'objet courant par le biais du mot-clé **this**, on peut facilement appeler plusieurs opérations successivement sur le même objet.

Appeler un constructeur depuis un autre constructeur

Quand une classe possède plusieurs constructeurs, il peut être utile d'appeler un constructeur depuis un autre pour éviter de la duplication de code. C'est possible grâce au mot-clé **this**.

En temps normal, **this** signifie «cet objet» ou «l'objet courant,» et renvoie une référence sur l'objet courant. Dans un constructeur, le mot-clé **this** prend un sens différent quand on lui passe une liste de paramètres : il signifie un appel explicite au constructeur qui correspond à cette liste de paramètres. Cela donne un moyen très simple d'appeler d'autres constructeurs :

```

///: c04:Flower.java

// Appel de constructeurs avec "this."

public class Flower {

    int petalCount = 0;

    String s = new String("null");

    Flower(int petals) {

        petalCount = petals;

        // Constructeur avec un unique paramètre int

        System.out.println(

            "Constructor w/ int arg only, petalCount= "

            + petalCount);
    }
}

```

```
}

Flower(String ss) {

    // Constructeur avec un unique paramètre String

    System.out.println(

        "Constructor w/ String arg only, s=" + ss);

    s = ss;

}

Flower(String s, int petals) {

    this(petals);

    //!    this(s); // Impossible d'en appeler deux !

    this.s = s; // Autre usage de "this"

    System.out.println("String & int args");

}


// Constructeur par défaut

Flower() {

    this("hi", 47);

    System.out.println(

        "default constructor (no args)");

}

void print() {

    //!    this(11); // Pas à l'intérieur d'une méthode normale !

    System.out.println(

        "petalCount = " + petalCount + " s = " + s);

}

public static void main(String[] args) {

    Flower x = new Flower();

    x.print();

}

} ///:~
```

Le constructeur **Flower(String s, int petals)** montre qu'on peut appeler un constructeur en utilisant **this**, mais pas deux. De plus, l'appel au constructeur doit absolument être la première instruction sinon le compilateur donnera un message d'erreur.

Cet exemple montre aussi un usage différent du mot-clé **this**. Les noms du paramètre **s** et du membre de données **s** étant les mêmes, il y a ambiguïté. On la résoud en utilisant **this.s** pour se référer au membre de données. Cette forme est très courante en Java et utilisée fréquemment dans ce livre.

Dans la méthode **print()** on peut voir que le compilateur ne permet pas l'appel d'un constructeur depuis toute autre méthode qu'un constructeur.

La signification de static

En pensant au mot-clé **this**, on comprend mieux le sens de rendre une méthode **static**. Cela signifie qu'il n'y a pas de **this** pour cette méthode. Il est impossible d'appeler une méthode non-**static** depuis une méthode **static** [28] (par contre, l'inverse est possible), et il est possible d'appeler une méthode **static** sur la classe elle-même, sans aucun objet. En fait, c'est principalement la raison de l'existence des méthodes **static**. C'est l'équivalent d'une fonction globale en C. Sauf que les fonctions globales sont interdites en Java, et ajouter une méthode **static** dans une classe lui permet d'accéder à d'autres méthodes **static** ainsi qu'aux membres **static**.

Certaines personnes argumentent que les méthodes **static** ne sont pas orientées objet puisqu'elles ont la sémantique des fonctions globales ; avec une méthode **static** on n'envoie pas un message vers un objet, puisqu'il n'y a pas de **this**. C'est probablement un argument valable, et si vous utilisez *beaucoup* de méthodes statiques vous devriez repenser votre stratégie. Pourtant, les méthodes **static** sont utiles et il y a des cas où on en a vraiment besoin. On peut donc laisser les théoriciens décider si oui ou non il s'agit de vraie programmation orientée objet. D'ailleurs, même Smalltalk a un équivalent avec ses «méthodes de classe.»

Nettoyage : finalisation et ramasse-miettes

Les programmeurs connaissent l'importance de l'initialisation mais oublient souvent celle du nettoyage. Après tout, qui a besoin de nettoyer un **int** ? Cependant, avec des bibliothèques, simplement oublier un objet après son utilisation n'est pas toujours sûr. Bien entendu, Java a un ramasse-miettes pour récupérer la mémoire prise par des objets qui ne sont plus utilisés. Considérons maintenant un cas très particulier. Supposons que votre objet alloue une zone de mémoire spéciale sans utiliser **new**. Le ramasse-miettes ne sait récupérer que la mémoire allouée *avec new*, donc il ne saura pas comment récupérer la zone «speciale» de mémoire utilisée par l'objet. Pour gérer ce cas, Java fournit une méthode appelée **finalize()** qui peut être définie dans votre classe. Voici comment c'est *supposé* marcher. Quand le ramasse-miettes est prêt à libérer la mémoire utilisée par votre objet, il va d'abord appeler **finalize()** et ce n'est qu'à la prochaine passe du ramasse-miettes que la mémoire de l'objet est libérée. En choisissant d'utiliser **finalize()**, on a la possibilité d'effectuer d'importantes tâches de nettoyage à l'exécution du ramasse-miettes.

C'est un piège de programmation parce que certains programmeurs, particulièrement les programmeurs C++, risquent au début de confondre **finalize()** avec le *destructeur* de C++ qui est une fonction toujours appelée quand un objet est détruit. Cependant il est important ici de faire la différence entre C++ et Java, car en C++ *les objets sont toujours détruits* (dans un programme sans bug), alors qu'en Java les objets ne sont pas toujours récupérés par le ramasse-miettes. Dit autrement :

Le mécanisme de ramasse-miettes n'est pas un mécanisme de destruction.

Si vous vous souvenez de cette règle de base, il n'y aura pas de problème. Cela veut dire que si une opération doit être effectuée avant la disparition d'un objet, celle-ci est à la charge du développeur. Java n'a pas de mécanisme équivalent au destructeur, il est donc nécessaire de créer une méthode ordinaire pour réaliser ce nettoyage. Par exemple, supposons qu'un objet se dessine à l'écran pendant sa création. Si son image n'est pas effacée explicitement de l'écran, il se peut qu'elle ne le soit jamais. Si l'on ajoute une fonctionnalité d'effacement dans **finalize()**, alors l'image sera effacée de l'écran si l'objet est récupéré par le ramasse-miette, sinon l'image

restera. Il y a donc une deuxième règle à se rappeler :

Les objets peuvent ne pas être récupérés par le ramasse-miettes.

Il se peut que la mémoire prise par un objet ne soit jamais libérée parce que le programme n'approche jamais la limite de mémoire qui lui a été attribuée. Si le programme se termine sans que le ramasse-miettes n'ait jamais libéré la mémoire prise par les objets, celle-ci sera rendue *en masse* (NDT : en français dans le texte) au système d'exploitation au moment où le programme s'arrête. C'est une bonne chose, car le ramasse-miettes implique un coût supplémentaire et s'il n'est jamais appelé, c'est autant d'économisé.

A quoi sert `finalize()` ?

A ce point, on peut croire qu'il ne faudrait pas utiliser **`finalize()`** comme méthode générale de nettoyage. A quoi sert-elle alors ?

Une troisième règle stipule :

Le ramasse-miettes ne s'occupe que de la mémoire.

C'est à dire que la seule raison d'exister du ramasse-miettes est de récupérer la mémoire que le programme n'utilise plus. Par conséquent, toute activité associée au ramasse-miettes, la méthode **`finalize()`** en particulier, doit se concentrer sur la mémoire et sa libération.

Est-ce que cela veut dire que si un objet contient d'autres objets, **`finalize()`** doit libérer ces objets explicitement ? La réponse est... non. Le ramasse-miettes prend soin de libérer tous les objets quelle que soit la façon dont ils ont été créés. Il se trouve que l'on a uniquement besoin de **`finalize()`** dans des cas bien précis où un objet peut allouer de la mémoire sans créer un autre objet. Cependant vous devez vous dire que tout est objet en Java, donc comment est-ce possible ?

Il semblerait que **`finalize()`** ait été introduit parce qu'il est possible d'allouer de la mémoire à-la-C en utilisant un mécanisme autre que celui proposé normalement par Java. Cela arrive généralement avec des *méthodes natives*, qui sont une façon d'appeler du code non-Java en Java (les méthodes natives sont expliquées en Appendice B). C et C++ sont les seuls langages actuellement supportés par les méthodes natives, mais comme elles peuvent appeler des routines écrites avec d'autres langages, il est en fait possible d'appeler n'importe quoi. Dans ce code non-Java, on peut appeler des fonctions de la famille de **`malloc()`** en C pour allouer de la mémoire, et à moins qu'un appel à **`free()`** ne soit effectué cette mémoire ne sera pas libérée, provoquant une «fuite». Bien entendu, **`free()`** est une fonction C et C++, ce qui veut dire qu'elle doit être appelée dans une méthode native dans le **`finalize()`** correspondant.

Maintenant, vous vous dites probablement que vous n'allez pas beaucoup utiliser **`finalize()`**. Vous avez raison : ce n'est pas l'endroit approprié pour effectuer des opérations normales de nettoyage. Dans ce cas, où celles-ci doivent-elles se passer ?

Le nettoyage est impératif

Pour nettoyer un objet, son utilisateur doit appeler une méthode de nettoyage au moment où celui-ci est nécessaire. Cela semble assez simple, mais se heurte au concept de destructeur de C++. En C++, tous les objets sont, ou plutôt *devraient être*, détruits. Si l'objet C++ est créé localement (c'est à dire sur la pile, ce qui n'est pas possible en Java), alors la destruction se produit à la fermeture de la portée dans laquelle l'objet a été créé. Si l'objet a été créé par **`new`** (comme en Java) le destructeur est appelé quand le programmeur appelle l'opérateur C++ **`delete`** (cet opérateur n'existe pas en Java). Si le programmeur C++ oublie d'appeler **`delete`**, le destructeur

n'est jamais appelé et l'on obtient une fuite mémoire. De plus les membres de l'objet ne sont jamais nettoyés non plus. Ce genre de bogue peut être très difficile à repérer.

Contrairement à C++, Java ne permet pas de créer des objets locaux, **new** doit toujours être utilisé. Cependant Java n'a pas de «delete» pour libérer l'objet car le ramasse-miettes se charge automatiquement de récupérer la mémoire. Donc d'un point de vue simplistique, on pourrait dire qu'à cause du ramasse-miettes, Java n'a pas de destructeur. Cependant à mesure que la lecture de ce livre progresse, on s'aperçoit que la présence d'un ramasse-miettes ne change ni le besoin ni l'utilité des destructeurs (de plus, **finalize()** ne devrait jamais être appelé directement, ce n'est donc pas une bonne solution pour ce problème). Si l'on a besoin d'effectuer des opérations de nettoyage autre que libérer la mémoire, il est *toujours* nécessaire d'appeler explicitement la méthode correspondante en Java, ce qui correspondra à un destructeur C++ sans être aussi pratique.

Une des utilisations possibles de **finalize()** est l'observation du ramasse-miettes. L'exemple suivant montre ce qui se passe et résume les descriptions précédentes du ramasse-miettes :

```
//: c04:Garbage.java

// Démonstration du ramasse-miettes
// et de la finalisation

class Chair {
    static boolean gcrun = false;
    static boolean f = false;
    static int created = 0;
    static int finalized = 0;
    int i;
    Chair() {
        i = ++created;
        if(created == 47)
            System.out.println("Created 47");
    }
    public void finalize() {
        if(!gcrun) {
            // Premier appel de finalize() :

            gcrun = true;
            System.out.println(
                "Beginning to finalize after " +
                created + " Chairs have been created");
        }
        if(i == 47) {
```

```
        System.out.println(
            "Finalizing Chair #47, " +
            "Setting flag to stop Chair creation");
        f = true;
    }
    finalized++;
    if(finalized >= created)
        System.out.println(
            "All " + finalized + " finalized");
    }
}

public class Garbage {
    public static void main(String[] args) {
        // Tant que le flag n'a pas été levé,
        // construire des objets Chair et String:
        while(!Chair.f) {
            new Chair();
            new String("To take up space");
        }
        System.out.println(
            "After all Chairs have been created:\n" +
            "total created = " + Chair.created +
            ", total finalized = " + Chair.finalized);
        // Arguments optionnels pour forcer
        // la finalisation et l'exécution du ramasse-miettes :
        if(args.length > 0) {
            if(args[0].equals("gc") ||
                args[0].equals("all")) {
                System.out.println("gc():");
                System.gc();
            }
            if(args[0].equals("finalize") ||
                args[0].equals("all")) {
```

```

        System.out.println("runFinalization()");

        System.runFinalization();
    }
}

System.out.println("bye!");
}

} ///:~

```

Le programme ci-dessus crée un grand nombre d'objets **Chair** et, à un certain point après que le ramasse-miettes ait commencé à s'exécuter, le programme arrête de créer des **Chairs**. Comme le ramasse-miettes peut s'exécuter n'importe quand, on ne sait pas exactement à quel moment il se lance, il existe donc un *flag* appelé **gerun** qui indique si le ramasse-miettes a commencé son exécution. Un deuxième *flag* **f** est le moyen pour **Chair** de prévenir la boucle **main()** qu'elle devrait arrêter de fabriquer des objets. On lève ces deux *flags* dans **finalize()**, qui est appelé pendant l'exécution du ramasse-miettes.

Deux autres variables **statiques**, **created** and **finalized**, enregistre le nombre d'objets **Chair** créés par rapport au nombre réclamé par le ramasse-miettes. Enfin, chaque objet **Chair** contient sa propre version (non statique) de l'**int i** pour savoir quel est son numéro. Quand l'objet **Chair** numéro 47 est réclamé, le flag est mis à **true** pour arrêter la création des objets **Chair**.

Tout ceci se passe dans le **main()**, dans la boucle

```

while(!Chair.f) {
    new Chair();
    new String("To take up space");
}

```

On peut se demander comment cette boucle va se terminer puisque rien dans la boucle ne change la valeur de **Chair.f**. Cependant, **finalize()** le fera au moment de la réclamation du numéro 47.

La création d'un objet **String** à chaque itération représente simplement de l'espace mémoire supplémentaire pour inciter le ramasse-miettes à s'exécuter, ce qu'il fera dès qu'il se sentira inquiet pour le montant de mémoire disponible.

A l'exécution du programme, l'utilisateur fournit une option sur la ligne de commande : «gc,» «finalize,» ou «all». Le paramètre «gc» permet l'appel de la méthode **System.gc()** (pour forcer l'exécution du ramasse-miettes). «finalize» permet d'appeler **System.runFinalization()** ce qui, en théorie, fait que tout objet non finalisé soit finalisé. Enfin, «all» exécute les deux méthodes.

Le comportement de ce programme et celui de la version de la première édition de cet ouvrage montrent que la question du ramasse-miettes et de la finalisation a évolué et qu'une grosse part de cette évolution s'est passée en coulisse. En fait, il est possible que le comportement du programme soit tout à fait différent lorsque vous lirez ces lignes.

Si **System.gc()** est appelé, alors la finalisation concerne tous les objets. Ce n'était pas forcément le cas avec les

implémentations précédentes du JDK bien que la documentation dise le contraire. De plus, il semble qu'appeler **System.runFinalization()** n'ait aucun effet.

Cependant, on voit que toutes les méthodes de finalisation sont exécutées seulement dans le cas où **System.gc()** est appelé après que tous les objets aient été créés et mis à l'écart. Si **System.gc()** n'est pas appelé, seulement certains objets seront finalisés. En Java 1.1, la méthode **System.runFinalizersOnExit()** fut introduite pour que les programmes puissent exécuter toutes les méthodes de finalisation lorsqu'ils se terminent, mais la conception était boguée et la méthode a été classée *deprecated*. C'est un indice supplémentaire qui montre que les concepteurs de Java ont eu de nombreux démêlés avec le problème du ramasse-miettes et de la finalisation. Il est à espérer que ces questions ont été réglées dans Java 2.

Le programme ci-dessus montre que les méthodes de finalisation sont toujours exécutées mais seulement si le programmeur force lui-même l'appel. Si on ne force pas l'appel de **System.gc()**, le résultat ressemblera à ceci :

```
Created 47
Beginning to finalize after 3486 Chairs have been created
Finalizing Chair #47, Setting flag to stop Chair creation
After all Chairs have been created:
total created = 3881, total finalized = 2684
bye!
```

Toutes les méthodes de finalisation ne sont donc pas appelées à la fin du programme. Ce n'est que quand **System.gc()** est appelé que tous les objets qui ne sont plus utilisés seront finalisés et détruits.

Il est important de se souvenir que ni le ramasse-miettes, ni la finalisation ne sont garantis. Si la machine virtuelle Java (JVM) ne risque pas de manquer de mémoire, elle ne perdra (légitimement) pas de temps à en récupérer grâce au ramasse-miettes.

La «death condition»

En général, on ne peut pas compter sur un appel à **finalize()**, et il est nécessaire de créer des fonctions spéciales de nettoyage et de les appeler explicitement. Il semblerait donc que **finalize()** ne soit utile que pour effectuer des tâches de nettoyage mémoire très spécifiques dont la plupart des programmeurs n'aura jamais besoin. Cependant, il existe une très intéressante utilisation de **finalize()** qui ne nécessite pas que son appel soit garanti. Il s'agit de la vérification de la *death condition* [\[29\]](#) d'un objet (état d'un objet à sa destruction).

Au moment où un objet n'est plus intéressant, c'est à dire lorsqu'il est prêt à être réclamé par le ramasse-miettes, cet objet doit être dans un état où sa mémoire peut être libérée sans problème. Par exemple, si l'objet représente un fichier ouvert, celui-ci doit être fermé par le programmeur avant que la mémoire prise par l'objet ne soit réclamée. Si certaines parties de cet objet n'ont pas été nettoyées comme il se doit, il s'agit d'un bogue du programme qui peut être très difficile à localiser. L'intérêt de **finalize()** est qu'il est possible de l'utiliser pour découvrir cet état de l'objet, même si cette méthode n'est pas toujours appelée. Si une des finalisations trouve le bogue, alors le problème est découvert et c'est ce qui compte vraiment après tout.

Voici un petit exemple pour montrer comment on peut l'utiliser :

```
///  
c04:DeathCondition.java
```

```
// Comment utiliser finalize() pour détecter les objets qui
// n'ont pas été nettoyés correctement.

class Book {
    boolean checkedOut = false;

    Book(boolean checkOut) {
        checkedOut = checkOut;
    }

    void checkIn() {
        checkedOut = false;
    }

    public void finalize() {
        if(checkedOut)
            System.out.println("Error: checked out");
    }
}

public class DeathCondition {
    public static void main(String[] args) {
        Book novel = new Book(true);

        // Nettoyage correct :
        novel.checkIn();

        // Perd la référence et oublie le nettoyage :
        new Book(true);

        // Force l'exécution du ramasse-miettes et de la finalisation :
        System.gc();
    }
} ///:~
```

Ici, la «death condition» est le fait que tous les objets de type **Book** doivent être «rendus» (checked in) avant d'être récupéré par le ramasse-miettes, mais dans la fonction **main()** une erreur de programmation fait qu'un de ces livres n'est pas rendu. Sans **finalize()** pour vérifier la «death condition», cela pourrait s'avérer un bogue difficile à trouver.

Il est important de noter l'utilisation de **System.gc()** pour forcer l'exécution de la finalisation (en fait, il est utile de le faire pendant le développement du programme pour accélérer le débogage). Cependant même si

System.gc() n'est pas appelé, il est très probable que le livre (**Book**) perdu soit découvert par plusieurs exécutions successives du programme (en supposant que suffisamment de mémoire soit alloué pour que le ramasse-miettes se déclenche).

Comment fonctionne un ramasse-miettes ?

Les utilisateurs de langages où l'allocation d'objets sur le tas coûte cher peuvent supposer que la façon qu'a Java de tout allouer sur le tas (à l'exception des types de base) coûte également cher. Cependant, il se trouve que l'utilisation d'un ramasse-miettes peut *accélérer* de manière importante la création d'objets. Ceci peut sembler un peu bizarre à première vue : la réclamation d'objets aurait un effet sur la création d'objets. Mais c'est comme ça que certaines JVMs fonctionnent et cela veut dire, qu'en Java, l'allocation d'objets sur le tas peut être presque aussi rapide que l'allocation sur la pile dans d'autres langages.

Un exemple serait de considérer le tas en C++ comme une pelouse où chaque objet prend et délimite son morceau de gazon. Cet espace peut être abandonné un peu plus tard et doit être réutilisé. Avec certaines JVMs, le tas de Java est assez différent ; il ressemble plus à une chaîne de montage qui avancerait à chaque fois qu'un objet est alloué. Ce qui fait que l'allocation est remarquablement rapide. Le «pointeur du tas» progresse simplement dans l'espace vide, ce qui correspond donc à l'allocation sur la pile en C++ (il y a bien sûr une petite pénalité supplémentaire pour le fonctionnement interne mais ce n'est pas comparable à la recherche de mémoire libre).

On peut remarquer que le tas n'est en fait pas vraiment une chaîne de montage, et s'il est traité de cette manière, la mémoire finira par avoir un taux de «paging» (utiliser toute la mémoire virtuelle incluant la partie sur disque dur) important (ce qui représente un gros problème de performance) et finira par manquer de mémoire. Le ramasse-miettes apporte la solution en s'interposant et, alors qu'il collecte les miettes (les objets inutilisables), il compacte tous les objets du tas. Ceci représente l'action de déplacer le «pointeur du tas» un peu plus vers le début et donc plus loin du «page fault» (interruption pour demander au système d'exploitation des pages de mémoire supplémentaire situées dans la partie de la mémoire virtuelle qui se trouve sur disque dur). Le ramasse-miettes réarrange tout pour permettre l'utilisation de ce modèle d'allocation très rapide et utilisant une sorte de «tas infini».

Pour comprendre comment tout cela fonctionne, il serait bon de donner maintenant une meilleure description de la façon dont un ramasse-miettes fonctionne. Nous utiliserons l'acronyme GC (en anglais, un ramasse-miette est appelé Garbage Collector) dans les paragraphes suivants. Une technique de GC relativement simple mais lente est le compteur de référence. L'idée est que chaque objet contient un compteur de référence et à chaque fois qu'une nouvelle référence sur un objet est créée le compteur est incrémenté. A chaque fois qu'une référence est hors de portée ou que la valeur **null** lui est assignée, le compteur de références est décrémenté. Par conséquent, la gestion des compteurs de références représente un coût faible mais constant tout au long du programme. Le ramasse-miettes se déplace à travers toute la liste d'objets et quand il en trouve un avec un compteur à zéro, il libère la mémoire. L'inconvénient principal est que si des objets se réfèrent de façon circulaire, ils ne peuvent jamais avoir un compteur à zéro tout en étant inaccessible. Pour localiser ces objets qui se réfèrent mutuellement, le ramasse-miettes doit faire un important travail supplémentaire. Les compteurs de références sont généralement utilisés pour expliquer les ramasses-miettes mais ils ne semblent pas être utilisés dans les implémentations de la JVM.

D'autres techniques, plus performantes, n'utilisent pas de compteur de références. Elles sont plutôt basées sur l'idée que l'on est capable de remonter la chaîne de références de tout objet «non-mort» (i.e encore en utilisation) jusqu'à une référence vivant sur la pile ou dans la zone statique. Cette chaîne peut très bien passer par plusieurs niveaux d'objets. Par conséquent, si l'on part de la pile et de la zone statique et que l'on trace toutes les références, on trouvera tous les objets encore en utilisation. Pour chaque référence que l'on trouve, il faut aller jusqu'à l'objet référencé et ensuite suivre toutes les références contenues dans *cet* objet, aller jusqu'aux objets référencés, etc. jusqu'à ce que l'on ait visité tous les objets que l'on peut atteindre depuis la référence sur la pile

ou dans la zone statique. Chaque objet visité doit être encore vivant. Notez qu'il n'y a aucun problème avec les groupes qui s'auto-référencent : ils ne sont tout simplement pas trouvés et sont donc automatiquement morts.

Avec cette approche, la JVM utilise un ramasse-miettes *adaptatif*. Le sort des objets vivants trouvés dépend de la variante du ramasse-miettes utilisée à ce moment-là. Une de ces variantes est le *stop-and-copy*. L'idée est d'arrêter le programme dans un premier temps (ce n'est pas un ramasse-miettes qui s'exécute en arrière-plan). Puis, chaque objet vivant que l'on trouve est copié d'un tas à un autre, délaissant les objets morts. De plus, au moment où les objets sont copiés, ils sont rassemblés les uns à côté des autres, compactant de ce fait le nouveau tas (et permettant d'allouer de la mémoire en la récupérant à l'extrémité du tas comme cela a été expliqué auparavant).

Bien entendu, quand un objet est déplacé d'un endroit à un autre, toutes les références qui pointent (i.e. qui *référencent*) l'objet doivent être mis à jour. La référence qui part du tas ou de la zone statique vers l'objet peut être modifiée sur le champ, mais il y a d'autres références pointant sur cet objet qui seront trouvées «sur le chemin». Elles seront corrigées dès qu'elles seront trouvées (on peut s'imaginer une table associant les anciennes adresses aux nouvelles).

Il existe deux problèmes qui rendent ces «ramasse-miettes par copie» inefficaces. Le premier est l'utilisation de deux tas et le déplacement des objets d'un tas à l'autre, utilisant ainsi deux fois plus de mémoire que nécessaire. Certaines JVMs s'en sortent en allouant la mémoire par morceau et en copiant simplement les objets d'un morceau à un autre.

Le deuxième problème est la copie. Une fois que le programme atteint un état stable, il se peut qu'il ne génère pratiquement plus de miettes (i.e. d'objets morts). Malgré ça, le ramasse-miettes par copie va quand même copier toute la mémoire d'une zone à une autre, ce qui est du gaspillage pur et simple. Pour éviter cela, certaines JVMs détectent que peu d'objets meurent et choisissent alors une autre technique (c'est la partie d'«adaptation»). Cette autre technique est appelée *mark and sweep* (NDT : littéralement *marque et balaye*), et c'est ce que les versions précédentes de la JVM de Sun utilisaient en permanence. En général, le «mark and sweep» est assez lent, mais quand on sait que l'on génère peu ou pas de miettes, la technique est rapide.

La technique de «mark and sweep» suit la même logique de partir de la pile et de la zone de mémoire statique et de suivre toutes les références pour trouver les objets encore en utilisation. Cependant, à chaque fois qu'un objet vivant est trouvé, il est marqué avec un flag, mais rien n'est encore collecté. C'est seulement lorsque la phase de «mark» est terminée que le «sweep» commence. Pendant ce balayage, les objets morts sont libérés. Aucune copie n'est effectuée, donc si le ramasse-miettes décide de compacter la mémoire, il le fait en réarrangeant les objets.

Le «stop-and-copy» correspond à l'idée que ce type de ramasse-miettes ne s'exécute *pas* en tâche de fond, le programme est en fait arrêté pendant l'exécution du ramasse-miettes. La littérature de Sun mentionne assez souvent le ramasse-miettes comme une tâche de fond de basse priorité, mais il se trouve que le ramasse-miettes n'a pas été implémenté de cette manière, tout au moins dans les premières versions de la JVM de Sun. Le ramasse-miettes était plutôt exécuté quand il restait peu de mémoire libre. De plus, le «mark-and-sweep» nécessite l'arrêt du programme.

Comme il a été dit précédemment, la JVM décrite ici alloue la mémoire par blocs. Si un gros objet est alloué, un bloc complet lui est réservé. Le «stop-and-copy» strictement appliqué nécessite la copie de chaque objet vivant du tas d'origine vers un nouveau tas avant de pouvoir libérer le vieux tas, ce qui se traduit par la manipulation de beaucoup de mémoire. Avec des blocs, le ramasse-miettes peut simplement utiliser les blocs vides (et/ou contenant uniquement des objets morts) pour y copier les objets. Chaque bloc possède un *compteur de génération* pour savoir s'il est «mort» (vide) ou non. Dans le cas normal, seuls les blocs créés depuis le ramasse-miettes sont compactés ; les compteurs de générations de tous les autres blocs sont mis à jour s'ils ont été référencés. Cela prend en compte le cas courant des nombreux objets ayant une durée de vie très courte.

Régulièrement, un balayage complet est effectué, les gros objets ne sont toujours pas copiés (leurs compteurs de génération sont simplement mis à jour) et les blocs contenant des petits objets sont copiés et compactés. La JVM évalue constamment l'efficacité du ramasse-miettes et si cette technique devient une pénalité plutôt qu'un avantage, elle la change pour un «mark-and-sweep». De même, la JVM évalue l'efficacité du mark-and-sweep et si le tas se fragmente, le stop-and-copy est réutilisé. C'est là où l'«adaptation» vient en place et finalement on peut utiliser ce terme anglophone à rallonge : «adaptive generational stop-and-copy mark-and-sweep» qui correspondrait à «adaptatif entre marque-et-balaye et stoppe-et-copie de façon générationnelle».

Il existe un certain nombre d'autres optimisations possibles dans une JVM. Une d'entre elles, très importante, implique le module de chargement des classes et le compilateur Just-In-Time (JIT). Quand une classe doit être chargée (généralement la première fois que l'on veut créer un objet de cette classe), le fichier **.class** est trouvé et le byte-code pour cette classe est chargé en mémoire. A ce moment-là, une possibilité est d'utiliser le JIT sur tout le code, mais cela a deux inconvénients. Tout d'abord c'est un peu plus coûteux en temps et, quand on considère toutes les classes chargées sur la durée de vie du programme, cela peut devenir conséquent. De plus, la taille de l'exécutable est augmentée (les byte codes sont bien plus compacts que le code résultant du JIT) et peut donc causer l'utilisation de pages dans la mémoire virtuelle, ce qui ralentit clairement le programme. Une autre approche est l'*évaluation paresseuse* qui n'utilise le JIT que lorsque cela est nécessaire. Par conséquent, si une partie du code n'est jamais exécutée, il est possible qu'elle ne soit jamais compilée par le JIT.

Initialisation de membre

Java prend en charge l'initialisation des variables avant leur utilisation. Dans le cas des variables locales à une méthode, cette garantie prend la forme d'une erreur à la compilation. Donc le code suivant :

```
void f() {  
    int i;  
    i++;  
}
```

générera un message d'erreur disant que la variable **i** peut ne pas avoir été initialisée. Bien entendu, le compilateur aurait pu donner à **i** une valeur par défaut, mais il est plus probable qu'il s'agit d'une erreur de programmation et une valeur par défaut aurait masqué ce problème. En forçant le programmeur à donner une valeur par défaut, il y a plus de chances de repérer un bogue.

Cependant, si une valeur primitive est un membre de données d'une classe, les choses sont un peu différentes. Comme n'importe quelle méthode peut initialiser ou utiliser cette donnée, il ne serait pas très pratique ou faisable de forcer l'utilisateur à l'initialiser correctement avant son utilisation. Cependant, il n'est pas correct de la laisser avec n'importe quoi comme valeur, Java garantit donc de donner une valeur initiale à chaque membre de données avec un type primitif. On peut voir ces valeurs ici :

```
//: c04:InitialValues.java  
// Imprime les valeurs initiales par défaut.  
  
class Measurement {  
    boolean t;
```

```

char c;

byte b;

short s;

int i;

long l;

float f;

double d;

void print() {
    System.out.println(
        "Data type      Initial value\n" +
        "boolean         " + t + "\n" +
        "char              [" + c + "] " + (int)c + "\n" +
        "byte              " + b + "\n" +
        "short             " + s + "\n" +
        "int               " + i + "\n" +
        "long              " + l + "\n" +
        "float             " + f + "\n" +
        "double            " + d);
}
}

public class InitialValues {
    public static void main(String[] args) {
        Measurement d = new Measurement();
        d.print();

        /* Dans ce cas, il est également possible d'écrire :
        new Measurement().print();
        */
    }
} ///:~

```

Voici la sortie de ce programme :

```

Data type      Initial value
boolean        false

```

```
char        [ ] 0
byte        0
short       0
int         0
long        0
float       0.0
double      0.0
```

La valeur pour **char** est zéro, ce qui se traduit par un espace dans la sortie-écran.

Nous verrons plus tard que quand on définit une référence sur un objet dans une classe sans l'initialiser avec un nouvel objet, la valeur spéciale **null** (mot-clé Java) est donnée à cette référence.

On peut voir que même si des valeurs ne sont pas spécifiées, les données sont initialisées automatiquement. Il n'y a donc pas de risque de travailler par inattention avec des variables non-initialisées.

Spécifier une initialisation

Comment peut-on donner une valeur initiale à une variable ? Une manière directe de le faire est la simple affectation au moment de la définition de la variable dans la classe (note : il n'est pas possible de le faire en C++ bien que tous les débutants s'y essayent). Les définitions des champs de la classe **Measurement** sont modifiées ici pour fournir des valeurs initiales :

```
class Measurement {
    boolean b = true;

    char c = 'x';

    byte B = 47;

    short s = 0xff;

    int i = 999;

    long l = 1;

    float f = 3.14f;

    double d = 3.14159;

    // . . .
}
```

On peut initialiser des objets de type non-primitif de la même manière. Si **Depth** (NDT : «profondeur») est une classe, on peut ajouter une variable et l'initialiser de cette façon :

```
class Measurement {
    Depth o = new Depth();

    boolean b = true;
}
```

```
// . . .
```

Si `o` ne reçoit pas de valeur initiale et que l'on essaye de l'utiliser malgré tout, on obtient une erreur à l'exécution appelée *exception* (explications au chapitre 10).

Il est même possible d'appeler une méthode pour fournir une valeur d'initialisation :

```
class CInit {
    int i = f();
    //...
}
```

Bien sûr cette méthode peut avoir des arguments, mais ceux-ci ne peuvent pas être d'autres membres non encore initialisés, de la classe. Par conséquent ce code est valide :

```
class CInit {
    int i = f();
    int j = g(i);
    //...
}
```

Mais pas celui-ci :

```
class CInit {
    int j = g(i);
    int i = f();
    //...
}
```

C'est un des endroits où le compilateur *se plaint* avec raison du forward referencing (référence à un objet déclaré plus loin dans le code), car il s'agit d'une question d'ordre d'initialisation et non pas de la façon dont le programme est compilé.

Cette approche par rapport à l'initialisation est très simple. Elle est également limitée dans le sens où *chaque* objet de type **Measurement** aura les mêmes valeurs d'initialisation. Quelquefois c'est exactement ce dont on a besoin, mais d'autres fois un peu plus de flexibilité serait nécessaire.

Initialisation par constructeur

On peut utiliser le constructeur pour effectuer les initialisations. Cela apporte plus de flexibilité pour le programmeur car il est possible d'appeler des méthodes et effectuer des actions à l'exécution pour déterminer les

valeurs initiales. Cependant il y a une chose à se rappeler : cela ne remplace pas l'initialisation automatique qui est faite avant l'exécution du constructeur. Donc par exemple :

```
class Counter {  
    int i;  
  
    Counter() { i = 7; }  
  
    // . . .
```

Dans ce cas, `i` sera d'abord initialisé à 0 puis à 7. C'est ce qui se passe pour tous les types primitifs et les références sur objet, même pour ceux qui ont été initialisés explicitement au moment de leur définition. Pour cette raison, le compilateur ne force pas l'utilisateur à initialiser les éléments dans le constructeur à un endroit donné, ni avant leur utilisation : l'initialisation est toujours garantie [\[30\]](#).

Ordre d'initialisation

Dans une classe, l'ordre d'initialisation est déterminé par l'ordre dans lequel les variables sont définies. Les définitions de variables peuvent être disséminées n'importe où et même entre les définitions des méthodes, mais elles sont initialisées avant tout appel à une méthode, même le constructeur. Par exemple :

```
///  
// c04:OrderOfInitialization.java  
// Montre l'ordre d'initialisation.  
  
// Quand le constructeur est appelé pour créer  
// un objet Tag, un message s'affichera :  
class Tag {  
    Tag(int marker) {  
        System.out.println("Tag(" + marker + ")");  
    }  
}  
  
class Card {  
    Tag t1 = new Tag(1); // Avant le constructeur  
    Card() {  
        // Montre que l'on est dans le constructeur :  
        System.out.println("Card()");  
        t3 = new Tag(33); // Réinitialisation de t3  
    }  
    Tag t2 = new Tag(2); // Après le constructeur
```

```

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

Tag t3 = new Tag(3); // la fin
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        Card t = new Card();
        t.f(); // Montre que la construction a été effectuée
    }
} ///:~

```

Dans la classe **Card**, les définitions des objets **Tag** sont intentionnellement dispersées pour prouver que ces objets seront tous initialisés avant toute action (y compris l'appel du constructeur). De plus, **t3** est réinitialisé dans le constructeur. La sortie-écran est la suivante :

```

Tag(1)
Tag(2)
Tag(3)
Card()
Tag(33)
f()

```

La référence sur **t3** est donc initialisée deux fois, une fois avant et une fois pendant l'appel au constructeur (on jette le premier objet pour qu'il soit récupéré par le ramasse-miettes plus tard). A première vue, cela ne semble pas très efficace, mais cela garantit une initialisation correcte ; que se passerait-il si l'on surchargeait le constructeur avec un autre constructeur qui *n'initialiserait pas t3* et qu'il n'y avait pas d'initialisation «par défaut» dans la définition de **t3** ?

Initialisation de données statiques

Quand les données sont statiques (**static**) la même chose se passe ; s'il s'agit d'une donnée de type primitif et qu'elle n'est pas initialisée, la variable reçoit une valeur initiale standard. Si c'est une référence sur un objet, c'est la valeur **null** qui est utilisée à moins qu'un nouvel objet ne soit créé et sa référence donnée comme valeur à la variable.

Pour une initialisation à l'endroit de la définition, les mêmes règles que pour les variables non-statiques sont appliquées. Il n'y a qu'une seule version (une seule zone mémoire) pour une variable statique quel que soit le nombre d'objets créés. Mais une question se pose lorsque cette zone statique est initialisée. Un exemple va rendre cette question claire :

```
//: c04:StaticInitialization.java
// Préciser des valeurs initiales dans une
// définition de classe.
```

```
class Bowl {
    Bowl(int marker) {
        System.out.println("Bowl(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}
```

```
class Table {
    static Bowl b1 = new Bowl(1);
    Table() {
        System.out.println("Table()");
        b2.f(1);
    }
    void f2(int marker) {
        System.out.println("f2(" + marker + ")");
    }
    static Bowl b2 = new Bowl(2);
}
```

```
class Cupboard {
    Bowl b3 = new Bowl(3);
    static Bowl b4 = new Bowl(4);
    Cupboard() {
        System.out.println("Cupboard()");
        b4.f(2);
    }
    void f3(int marker) {
        System.out.println("f3(" + marker + ")");
    }
}
```

```

    }

    static Bowl b5 = new Bowl(5);
}

public class StaticInitialization {
    public static void main(String[] args) {
        System.out.println(
            "Creating new Cupboard() in main");
        new Cupboard();
        System.out.println(
            "Creating new Cupboard() in main");
        new Cupboard();
        t2.f2(1);
        t3.f3(1);
    }

    static Table t2 = new Table();
    static Cupboard t3 = new Cupboard();
} ///:~

```

Bowl permet de visionner la création d'une classe. **Table**, ainsi que **Cupboard**, créent des membres **static** de **Bowl** partout au travers de leur définition de classe. Il est à noter que **Cupboard** crée un **Bowl b3** non-**statique** avant les définitions **statiques**. La sortie montre ce qui se passe :

```

Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creating new Cupboard() in main
Bowl(3)
Cupboard()

```



```

f(2)

Creating new Cupboard() in main

Bowl(3)

Cupboard()

f(2)

f2(1)

f3(1)

```

L'initialisation **statique** intervient seulement si c'est nécessaire. Si on ne crée jamais d'objets **Table** et que **Table.b1** ou **Table.b2** ne sont jamais référencés, les membres **statiques Bowl b1** et **b2** ne seront jamais créés. Cependant, ils ne sont initialisés que lorsque le *premier* objet **Table** est créé (ou le premier accès **statique** est effectué). Après cela, les objets **statiques** ne sont pas réinitialisés.

Dans l'ordre d'initialisation, les membres **static** viennent en premier, s'ils n'avaient pas déjà été initialisés par une précédente création d'objet, les objets non **static** sont traités. On peut le voir clairement dans la sortie du programme.

Il peut être utile de résumer le processus de création d'un objet. Considérons une classe appelée **Dog** :

1. La première fois qu'un objet de type **Dog** est créé, *ou* la première fois qu'on utilise une méthode déclarée **static** ou un champ **static** de la classe **Dog**, l'interpréteur Java doit localiser **Dog.class**, ce qu'il fait en cherchant dans le classpath ;
2. Au moment où **Dog.class** est chargée (créant un objet **Class**, que nous verrons plus tard), toutes les fonctions d'initialisation **statiques** sont exécutées. Par conséquent, l'initialisation **statique** n'arrive qu'une fois, au premier chargement de l'objet **Class** ;
3. Lorsque l'on exécute **new Dog()** pour créer un nouvel objet de type **Dog**, le processus de construction commence par allouer suffisamment d'espace mémoire sur le tas pour contenir un objet **Dog** ;
4. Cet espace est mis à zéro, donnant automatiquement à tous les membres de type primitif dans cet objet **Dog** leurs valeurs par défaut (zéro pour les nombres et l'équivalent pour les **boolean** et les **char**) et aux références la valeur **null** ;
5. Toute initialisation effectuée au moment de la définition des champs est exécutée ;
6. Les constructeurs sont exécutés. Comme nous le verrons au chapitre 6, ceci peut en fait déclencher beaucoup d'activité, surtout lorsqu'il y a de l'héritage.

Initialisation statique explicite

Java permet au programmeur de grouper toute autre initialisation statique dans une «clause de construction» **static** (quelquefois appelé *bloc statique*) dans une classe. Cela ressemble à ceci :

```

class Spoon {
    static int i;

    static {
        i = 47;
    }
}

```

```
// . . .
```

On dirait une méthode, mais il s'agit simplement du mot-clé **static** suivi d'un corps de méthode. Ce code, comme les autres initialisations statiques, est exécuté une seule fois, à la création du premier objet de cette classe *ou* au premier accès à un membre déclaré **static** de cette classe (même si on ne crée jamais d'objet de cette classe). Par exemple :

```
//: c04:ExplicitStatic.java
// Initialisation statique explicite
// avec l'instruction "static".

class Cup {
    Cup(int marker) {
        System.out.println("Cup(" + marker + ")");
    }
    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

class Cups {
    static Cup c1;
    static Cup c2;
    static {
        c1 = new Cup(1);
        c2 = new Cup(2);
    }
    Cups() {
        System.out.println("Cups()");
    }
}

public class ExplicitStatic {
    public static void main(String[] args) {
        System.out.println("Inside main()");
        Cups.c1.f(99);    // (1)
    }
}
```

```

    }

    // static Cups x = new Cups(); // (2)

    // static Cups y = new Cups(); // (2)

} ///:~

```

Les instructions statiques d'initialisation pour **Cups** sont exécutées soit quand l'accès à l'objet **static c1** intervient à la ligne (1), soit si la ligne (1) est mise en commentaire et les lignes (2) ne le sont pas. Si (1) et (2) sont en commentaire, l'initialisation **static** pour **Cups** n'intervient jamais. De plus, que l'on enlève les commentaires pour les deux lignes (2) ou pour une seule n'a aucune importance : l'initialisation statique n'est effectuée qu'une seule fois.

Initialisation d'instance non statique

Java offre une syntaxe similaire pour initialiser les variables non **static** pour chaque objet. Voici un exemple :

```

///: c04:Mugs.java

// Java "Instance Initialization." (Initialisation d'instance de Java)

```

```

class Mug {
    Mug(int marker) {
        System.out.println("Mug(" + marker + ")");
    }

    void f(int marker) {
        System.out.println("f(" + marker + ")");
    }
}

```

```

public class Mugs {
    Mug c1;
    Mug c2;
    {
        c1 = new Mug(1);
        c2 = new Mug(2);
        System.out.println("c1 & c2 initialized");
    }
    Mugs() {
        System.out.println("Mugs()");
    }
}

```

```

    }

    public static void main(String[] args) {
        System.out.println("Inside main()");

        Mugs x = new Mugs();
    }
} ///:~

```

On peut voir que la clause d'initialisation d'instance :

```

{
    c1 = new Mug(1);
    c2 = new Mug(2);

    System.out.println("c1 & c2 initialized");
}

```

ressemble exactement à la clause d'initialisation statique moins le mot-clé **static**. Cette syntaxe est nécessaire pour permettre l'initialisation de *classes internes anonymes* (voir Chapitre 8).

Initialisation des tableaux

L'initialisation des tableaux en C est laborieuse et source d'erreurs. C++ utilise *l'initialisation d'aggregats* pour rendre cette opération plus sûre [\[31\]](#). Java n'a pas d'«aggregats» comme C++, puisque tout est objet en Java. Java possède pourtant des tableaux avec initialisation.

Un tableau est simplement une suite d'objets ou de types de base, tous du même type et réunis ensemble sous un même nom. Les tableaux sont définis et utilisés avec l'*opérateur d'indexation* `[]` (crochets ouvrant et fermant). Pour définir un tableau il suffit d'ajouter des crochets vides après le nom du type :

```
int[] a1;
```

Les crochets peuvent également être placé après le nom de la variable :

```
int a1[];
```

Cela correspond aux attentes des programmeurs C et C++. Toutefois, la première syntaxe est probablement plus sensée car elle annonce le type comme un «tableau de **int**.» Ce livre utilise cette syntaxe.

Le compilateur ne permet pas de spécifier la taille du tableau à sa définition. Cela nous ramène à ce problème de «référence.» A ce point on ne dispose que d'une référence sur un tableau, et aucune place n'a été allouée pour ce tableau. Pour créer cet espace de stockage pour le tableau, il faut écrire une expression d'initialisation. Pour les tableaux, l'initialisation peut apparaître à tout moment dans le code, mais on peut également utiliser un type spécial d'initialisation qui doit alors apparaître à la déclaration du tableau. Cette initialisation spéciale est un ensemble de valeurs entre accolades. L'allocation de l'espace de stockage pour le tableau (l'équivalent de **new**)

est prise en charge par le compilateur dans ce cas. Par exemple :

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Mais pourquoi voudrait-on définir une référence sur tableau sans tableau ?

```
int[] a2;
```

Il est possible d'affecter un tableau à un autre en Java, on peut donc écrire :

```
a2 = a1;
```

Cette expression effectue en fait une copie de référence, comme le montre la suite :

```
///  
// Tableau de types primitifs.  
  
public class Arrays {  
    public static void main(String[] args) {  
        int[] a1 = { 1, 2, 3, 4, 5 };  
        int[] a2;  
        a2 = a1;  
        for(int i = 0; i < a2.length; i++)  
            a2[i]++;  
        for(int i = 0; i < a1.length; i++)  
            System.out.println(  
                "a1[" + i + "] = " + a1[i]);  
    }  
} ///:~
```

On peut voir que **a1** a une valeur initiale tandis qu' **a2** n'en a pas ; **a2** prend une valeur plus tard— dans ce cas, vers un autre tableau.

Maintenant voyons quelque chose de nouveau : tous les tableaux ont un membre intrinsèque (qu'ils soient tableaux d'objets ou de types de base) que l'on peut interroger — mais pas changer — ; il donne le nombre d'éléments dans le tableau. Ce membre s'appelle **length** (longueur). Comme les tableaux en Java , comme C et C++, commencent à la case zero, le plus grand nombre d'éléments que l'on peut indexer est **length - 1**. Lorsqu'on dépasse ces bornes, C et C++ acceptent cela tranquillement et la mémoire peut être corrompue ; ceci est la cause de bogues infâmes. Par contre, Java empêche ce genre de problèmes en générant une erreur d'exécution (une *exception*, le sujet du Chapitre 10) lorsque le programme essaye d'accéder à une valeur en

dehors des limites. Bien sûr, vérifier ainsi chaque accès coûte du temps et du code ; comme il n'y a aucun moyen de désactiver ces vérifications, les accès tableaux peuvent être une source de lenteur dans un programme s'ils sont placés à certains points critiques de l'exécution. Les concepteurs de Java ont pensé que cette vitesse légèrement réduite était largement contrebalancée par les aspects de sécurité sur Internet et la meilleure productivité des programmeurs.

Que faire quand on ne sait pas au moment où le programme est écrit, combien d'éléments vont être requis à l'exécution ? Il suffit d'utiliser **new** pour créer les éléments du tableau. Dans ce cas, **new** fonctionne même pour la création d'un tableau de types de base (**new** ne peut pas créer un type de base) :

```
///  
// Créer des tableaux avec new.  
import java.util.*;  
  
public class ArrayNew {  
    static Random rand = new Random();  
    static int pRand(int mod) {  
        return Math.abs(rand.nextInt()) % mod + 1;  
    }  
    public static void main(String[] args) {  
        int[] a;  
        a = new int[pRand(20)];  
        System.out.println(  
            "length of a = " + a.length);  
        for(int i = 0; i < a.length; i++)  
            System.out.println(  
                "a[" + i + "] = " + a[i]);  
    }  
} ///:~
```

Comme la taille du tableau est choisie aléatoirement (en utilisant la méthode **pRand()**), il est clair que la création du tableau se passe effectivement à l'exécution. De plus, on peut voir en exécutant le programme que les tableaux de types primitifs sont automatiquement initialisés avec des valeurs "vides" (pour les nombres et les **char**, cette valeur est zéro, pour les **boolean**, cette valeur est **false**).

Bien sûr le tableau pourrait aussi avoir été défini et initialisé sur la même ligne :

```
int[] a = new int[pRand(20)];
```

Lorsque l'on travaille avec un tableau d'objets non primitifs, il faut toujours utiliser **new**. Encore une fois, le

problème des références revient car ce que l'on crée est un tableau de références. Considérons le type englobant **Integer**, qui est une classe et non un type de base :

```

//: c04:ArrayClassObj.java
// Création d'un tableau d'objets (types de base exclus).

import java.util.*;

public class ArrayClassObj {

    static Random rand = new Random();

    static int pRand(int mod) {

        return Math.abs(rand.nextInt()) % mod + 1;

    }

    public static void main(String[] args) {

        Integer[] a = new Integer[pRand(20)];

        System.out.println(

            "length of a = " + a.length);

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

            a[i] = new Integer(pRand(500));

            System.out.println(

                "a[" + i + "] = " + a[i]);

        }

    }

} ///:~

```

Ici, même après que **new** ait été appelé pour créer le tableau :

```
Integer[] a = new Integer[pRand(20)];
```

c'est uniquement un tableau de références, et l'initialisation n'est pas complète tant que cette référence n'a pas elle-même été initialisée en créant un nouvel objet **Integer** :

```
a[i] = new Integer(pRand(500));
```

Oublier de créer l'objet produira une exception d'exécution dès que l'on accédera à l'emplacement.

Regardons la formation de l'objet **String** à l'intérieur de print. On peut voir que la référence vers l'objet **Integer** est automatiquement convertie pour produire une **String** représentant la valeur à l'intérieur de l'objet.

Il est également possible d'initialiser des tableaux d'objets en utilisant la liste délimitée par des accolades. Il y a deux formes :

```
///  
// Initialisation de tableaux.  
  
public class ArrayInit {  
    public static void main(String[] args) {  
        Integer[] a = {  
            new Integer(1),  
            new Integer(2),  
            new Integer(3),  
        };  
  
        Integer[] b = new Integer[] {  
            new Integer(1),  
            new Integer(2),  
            new Integer(3),  
        };  
    }  
} ///:~
```

C'est parfois utile, mais d'un usage plus limité car la taille du tableau est déterminée à la compilation. La virgule finale dans la liste est optionnelle. (Cette fonctionnalité permet une gestion plus facile des listes longues.)

La deuxième forme d'initialisation de tableaux offre une syntaxe pratique pour créer et appeler des méthodes qui permet de donner le même effet que *les listes à nombre d'arguments variable* de C ("varargs" en C). Ces dernières permettent le passage d'un nombre quelconque de paramètres, chacun de type inconnu.

Comme toutes les classes héritent d'une classe racine **Object** (un sujet qui sera couvert en détail tout au long du livre), on peut créer une méthode qui prend un tableau d'**Object** et l'appeler ainsi :

```
///  
// Utilisation de la syntaxe des tableaux pour créer  
// des listes à nombre d'argument variable.  
  
class A { int i; }  
  
public class VarArgs {
```



```

static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
        System.out.println(x[i]);
}

public static void main(String[] args) {
    f(new Object[] {
        new Integer(47), new VarArgs(),
        new Float(3.14), new Double(11.11) });
    f(new Object[] {"one", "two", "three" });
    f(new Object[] {new A(), new A(), new A()});
}
} ///:~

```

A ce niveau, il n'y a pas grand chose que l'on peut faire avec ces objets inconnus, et ce programme utilise la conversion automatique vers **String** afin de faire quelque chose d'utile avec chacun de ces **Objects**. Au chapitre 12, qui explique l'*identification dynamique de types* (RTTI), nous verrons comment découvrir le type exact de tels objets afin de les utiliser à des fins plus intéressantes.

Tableaux multidimensionnels

Java permet de créer facilement des tableaux multidimensionnels :

```

//: c04:MultiDimArray.java
// Création de tableaux multidimensionnels.

import java.util.*;

public class MultiDimArray {
    static Random rand = new Random();

    static int pRand(int mod) {
        return Math.abs(rand.nextInt()) % mod + 1;
    }

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

    public static void main(String[] args) {
        int[][] a1 = {
            { 1, 2, 3, },

```

```
    { 4, 5, 6, },
};

for(int i = 0; i < a1.length; i++)
    for(int j = 0; j < a1[i].length; j++)
        prt("a1[" + i + "][" + j +
            "] = " + a1[i][j]);

// tableau 3-D avec taille fixe :
int[][][] a2 = new int[2][2][4];
for(int i = 0; i < a2.length; i++)
    for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length;
            k++)
            prt("a2[" + i + "][" + j +
                "] = " + a2[i][j][k]);

// tableau 3-D avec vecteurs de taille variable :
int[][][] a3 = new int[pRand(7)][][];
for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[pRand(5)][];
    for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
}
for(int i = 0; i < a3.length; i++)
    for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length;
            k++)
            prt("a3[" + i + "][" + j +
                "] = " + a3[i][j][k]);

// Tableau d'objets non primitifs :
Integer[][] a4 = {
    { new Integer(1), new Integer(2) },
    { new Integer(3), new Integer(4) },
    { new Integer(5), new Integer(6) },
}
```

```

};

for(int i = 0; i < a4.length; i++)
    for(int j = 0; j < a4[i].length; j++)
        prt("a4[" + i + "][" + j +
            "]" = " + a4[i][j]);

Integer[][] a5;
a5 = new Integer[3][];
for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];
    for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
}
for(int i = 0; i < a5.length; i++)
    for(int j = 0; j < a5[i].length; j++)
        prt("a5[" + i + "][" + j +
            "]" = " + a5[i][j]);
}
} ///:~

```

Le code d'affichage utilise **length** ; de cette façon il ne force pas une taille de tableau fixe.

Le premier exemple montre un tableau multidimensionnel de type primitifs. Chaque vecteur du tableau est délimité par des accolades :

```

int[][] a1 = {
    { 1, 2, 3, },
    { 4, 5, 6, },
};

```

Chaque paire de crochets donne accès à la dimension suivante du tableau.

Le deuxième exemple montre un tableau à trois dimensions alloué par **new**. Ici le tableau entier est alloué en une seule fois :

```

int[][][] a2 = new int[2][2][4];

```

Par contre, le troisième exemple montre que les vecteurs dans les tableaux qui forment la matrice peuvent être de longueurs différentes :

```

int[][][] a3 = new int[pRand(7)][][];

for(int i = 0; i < a3.length; i++) {
    a3[i] = new int[pRand(5)][];

    for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
}

```

Le premier **new** crée un tableau avec un longueur aléatoire pour le premier élément et le reste de longueur indéterminée. Le deuxième **new** à l'intérieur de la boucle **for** remplit les éléments mais laisse le troisième index indéterminé jusqu'au troisième **new**.

On peut voir à l'exécution que les valeurs des tableaux sont automatiquement initialisées à zéro si on ne leur donne pas explicitement de valeur initiale.

Les tableaux d'objets non primitifs fonctionnent exactement de la même manière, comme le montre le quatrième exemple, qui présente la possibilité d'utiliser **new** dans les accolades d'initialisation :

```

Integer[][] a4 = {
    { new Integer(1), new Integer(2)},
    { new Integer(3), new Integer(4)},
    { new Integer(5), new Integer(6)},
};

```

Le cinquième exemple montre comment un tableau d'objets non primitifs peut être construit pièce par pièce :

```

Integer[][] a5;

a5 = new Integer[3][];

for(int i = 0; i < a5.length; i++) {
    a5[i] = new Integer[3];

    for(int j = 0; j < a5[i].length; j++)
        a5[i][j] = new Integer(i*j);
}

```

L'expression **i*j** est là uniquement pour donner une valeur intéressante à l' **Integer**.

Résumé

Le mécanisme apparemment sophistiqué d'initialisation que l'on appelle constructeur souligne l'importance donnée à l'initialisation dans ce langage. Quand Stroustrup était en train de créer C++, une des premières

observations qu'il fit à propos de la productivité en C était qu'une initialisation inappropriée des variables cause de nombreux problèmes de programmation. Ce genre de bogues est difficile à trouver. Des problèmes similaires se retrouvent avec un mauvais nettoyage. Parce que les constructeurs permettent de *garantir* une initialisation et un nettoyage correct (le compilateur n'autorisera pas la création d'un objet sans un appel valide du constructeur), le programmeur a un contrôle complet en toute sécurité.

En C++, la destruction est importante parce que les objets créés avec **new** doivent être détruits explicitement. En Java, le ramasse-miettes libère automatiquement la mémoire pour tous les objets, donc la méthode de nettoyage équivalente en Java n'est pratiquement jamais nécessaire. Dans les cas où un comportement du style destructeur n'est pas nécessaire, le ramasse-miettes de Java simplifie grandement la programmation et ajoute une sécurité bien nécessaire à la gestion mémoire. Certains ramasse-miettes peuvent même s'occuper du nettoyage d'autres ressources telles que les graphiques et les fichiers. Cependant, le prix du ramasse-miettes est payé par une augmentation du temps d'exécution, qu'il est toutefois difficile d'évaluer à cause de la lenteur globale des interpréteurs Java au moment de l'écriture de cet ouvrage. Lorsque cela changera, il sera possible de savoir si le coût du ramasse-miettes posera des barrières à l'utilisation de Java pour certains types de programmes (un des problèmes est que le ramasse-miettes est imprévisible).

Parce que Java garantit la construction de tous les objets, le constructeur est, en fait, plus conséquent que ce qui est expliqué ici. En particulier, quand on crée de nouvelles classes en utilisant soit *la composition*, soit *l'héritage* la garantie de construction est maintenue et une syntaxe supplémentaire est nécessaire. La composition, l'héritage et leurs effets sur les constructeurs sont expliqués un peu plus loin dans cet ouvrage.

Exercices

Les solutions aux exercices choisis peuvent être trouvées dans le document électronique *The Thinking in Java Annotated Solution Guide*, disponible pour une modeste somme à l'adresse www.BruceEckel.com.

1. Créez une classe avec un constructeur par défaut (c'est à dire sans argument) qui imprime un message. Créez un objet de cette classe.
2. Ajoutez à la classe de l'exercice 1 un constructeur surchargé qui prend une **String** en argument et qui l'imprime avec votre message.
3. Créez un tableau de références sur des objets de la classe que vous avez créée à l'exercice 2. Mais ne créez pas les objets eux-même. Quand le programme s'exécute, voyez si les messages d'initialisation du constructeur sont imprimés.
4. Terminez l'exercice 3 en créant les objets pour remplir le tableau de références.
5. Créez un tableau d'objets **String** et affectez une chaîne de caractères à chaque élément. Imprimez le tableau en utilisant une boucle **for**.
6. Créez une classe **Dog** avec une méthode **bark()** (NDT: to bark = aboyer) surchargée. Cette méthode sera surchargée en utilisant divers types primitifs de données et devra imprimer différents types d'aboiement, hurlement, ... suivant la version surchargée qui est appelée. Ecrivez également une méthode **main()** qui appellera toutes les versions.
7. Modifiez l'exercice 6 pour que deux des méthodes surchargées aient deux paramètres (de deux types différents), mais dans l'ordre inverse l'une par rapport à l'autre. Vérifiez que cela fonctionne.
8. Créez une classe sans constructeur et créez ensuite un objet de cette classe dans **main()** pour vérifier que le constructeur par défaut est construit automatiquement.
9. Créez une classe avec deux méthodes. Dans la première méthode, appelez la seconde méthode deux fois : la première fois sans utiliser **this** et la seconde fois en l'utilisant.
10. Créez une classe avec deux constructeurs (surchargés). En utilisant **this**, appelez le second constructeur dans le premier.
11. Créez une classe avec une méthode **finalize()** qui imprime un message. Dans **main()**, créez un objet de cette classe. Expliquez le comportement de ce programme.

12. Modifiez l'exercice 11 pour que votre **finalize()** soit toujours appelé.
13. Créez une classe **Tank** (NDT: citerne) qui peut être remplie et vidée et qui a une *death condition* qui est que la citerne doit être vide quand l'objet est nettoyé. Ecrivez une méthode **finalize()** qui vérifie cette *death condition*. Dans **main()**, testez tous les scénarii possibles d'utilisation de **Tank**.
14. Créez une classe contenant un **int** et un **char** non initialisés et imprimez leurs valeurs pour vérifier que Java effectue leurs initialisations par défaut.
15. Créez une classe contenant une référence non-initialisée à une **String**. Montrez que cette référence est initialisée à **null** par Java.
16. Créez une classe avec un champ **String** qui est initialisé à l'endroit de sa définition et un autre qui est initialisé par le constructeur. Quelle est la différence entre les deux approches ?
17. Créez une classe avec un champ **static String** qui est initialisé à l'endroit de la définition et un autre qui est initialisé par un bloc **static**. Ajoutez une méthode statique qui imprime les deux champs et montre qu'ils sont initialisés avant d'être utilisés.
18. Créez une classe avec un champ **String** qui est initialisé par une «initialisation d'instance». Décrire une utilisation de cette fonctionnalité (autre que celle spécifiée dans cet ouvrage).
19. Ecrivez une méthode qui crée et initialise un tableau de **double** à deux dimensions. La taille de ce tableau est déterminée par les arguments de la méthode. Les valeurs d'initialisation sont un intervalle déterminé par des valeurs de début et de fin également donné en paramètres de la méthode. Créez une deuxième méthode qui imprimera le tableau généré par la première. Dans **main()**, testez les méthodes en créant et en imprimant plusieurs tableaux de différentes tailles.
20. Recommencez l'exercice 19 pour un tableau à trois dimensions.
21. Mettez en commentaire la ligne marquée (1) dans **ExplicitStatic.java** et vérifiez que la clause d'initialisation statique n'est pas appelée. Maintenant décommentez une des lignes marquées (2) et vérifiez que la clause d'initialisation statique *est* appelée. Décommentez maintenant l'autre ligne marquée (2) et vérifiez que l'initialisation statique n'est effectuée qu'une fois.
22. Faites des expériences avec **Garbage.java** en exécutant le programme avec les arguments «gc», «finalize,» ou «all». Recommencez le processus et voyez si vous détectez des motifs répétitifs dans la sortie écran. Modifiez le code pour que **System.runFinalization()** soit appelé *avant* **System.gc()** et regardez les résultats.

[27] Dans certains articles écrits par Sun relatifs à Java, il est plutôt fait référence au terme maladroit bien que descriptif «no-arg constructors». Le terme «constructeur par défaut» est utilisé depuis des années et c'est donc celui que j'utiliserai.

[28] Le seul cas dans lequel cela est possible, est si l'on passe une référence à un objet dans la méthode **statique**. Ensuite, en utilisant la référence (qui est en fait **this** maintenant), on peut appeler des méthodes non-**statiques** et accéder à des champs non-**statiques**. Mais, en général, lorsque l'on veut faire quelque chose comme cela, on crée tout simplement une méthode non-statique.

[29] Un terme créé par Bill Venners (www.artima.com) pendant le séminaire que lui et moi avons donné ensemble.

[30] En comparaison, C++ possède la *liste d'initialisation du constructeur* qui déclenche l'initialisation avant d'entrer dans le corps du constructeur. Voir *Thinking in C++, 2^{nde} édition* (disponible sur le CDROM de cet ouvrage et à www.BruceEckel.com).

[31] Voir *Thinking in C++, 2^{nde} édition* pour une description complète de l'initialisation par agrégat en C++.