Prog. orientée objet avancée: Java

Jean-Francois Lalande - May 2015

Ce cours présente les aspects avancés de la programmation orientée objet en Java. Il s'agit de couvrir les particularités liées à l'environnement de la machine virtuelle, les aspects de programmation concurrente, les entrées / sorties, l'introspection.

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

1 Plan du module

Plan du module

2 Machine virtuelle Java

Contributeurs

Guillaume Hiet, Jean-Francois Lalande

2.1 Rôle de la machine virtuelle

La machine virtuelle travaille sur le *bytecode*, en général obtenu à partir de fichiers sources Java. Elle interprète le *bytecode* contenu dans les .class ou .jar. Elle peut aussi les compiler à la volée (*just-in-time* compiler, JIT). La plupart des machines virtuelles modernes peuvent interpréter ou compiler le *bytecode*. Enfin, certains outils permettent de compiler du *bytecode* en code natif.

A la différence des langages classiques *write once, compile anywhere*, le langage Java est du type *compile once, run anywhere*. Le code compilé, le *bytecode* peut être exécuté indifférement sur une machine virtuelle implémentée pour fonctionner sur Windows, Linux, Android, etc...

Liste non exhaustive de quelques machines virtuelles:

- Sun Microsystems
- GNU Compiler for the Java Programming Language
- IBM
- ...

2.2 Le bytecode

Le *bytecode* est une séquence d'instruction pour la machine virtuelle. La JVM stocke pour chaque classe chargée le flot de bytecode associé à chaque méthode. Une méthode peut être par exemple constituée du flot ci-dessous [BB]:

```
// Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
iconst_0      // 03
istore_0      // 3b
iinc 0, 1     // 84 00 01
iload_0       // 1a
iconst_2      // 05
imul          // 68
istore_0      // 3b
goto -7       // a7 ff f9
```

Le nombre d'*opcodes* est petit ce qui permet de faire tenir tous les *opcodes* sur un octet. Brièvement, voici une liste des opcodes:

- **iconst_X**: empiler la constante X sur la pile
- **iload_X**: empiler la variable locale n°X
- **istore_X**: dépiler un entier et le stocker dans la variable locale n°X
- **i2f**: convertir un int en float
- **iadd, imul, iinc...**: opérations arithmétiques
- **ireturn**: retourne le résultat

Example de code source et de bytecode

Voici un extrait tiré de [BB]:

```
byte a = 1;
byte b = 1;
byte c = (byte) (a + b);
return c;
```

Qui se retrouve compilé sous la forme:

```
iconst_1  // Push int constant 1.
istore_1  // Pop into local variable 1, which is a: byte a = 1;
iconst_1  // Push int constant 1 again.
istore_2  // Pop into local variable 2, which is b: byte b = 1;
iload_1   // Push a (a is already stored as an int in local variable 1).
iload_2   // Push b (b is already stored as an int in local variable 2).
iadd      // Perform addition. Top of stack is now (a + b), an int.
int2byte  // Convert int result to byte (result still occupies 32 bits).
istore_3  // Pop into local variable 3, which is byte c: byte c = (byte) (a + b);
iload_3   // Push the value of c so it can be returned.
ireturn   // Proudly return the result of the addition: return c;
```

Decompilation à l'aide de l'outil javap

```
public class Decompilation {
  int test() {
      byte a = 1;
      byte b = 1;
      byte c = (byte) (a + b);
      return c;
  }

  public static void main(String[] args) {
      Decompilation d = new Decompilation();
      int res = d.test();
      System.out.println("Out: " + res);
  }
}
```

La décompilation peut se faire à l'aide de l'outil **javap**:

```
javap -c -private Decompilation
```

- -public: Shows only public classes and members.
- -protected: Shows only protected and public classes and members.
- -package: Shows only package, protected, and public classes and members.
- -private: Shows all classes and members.

Exemple de décompilation

Par exemple, le code précédent décompilé par:

```
javap -c -public  Decompilation > Decompilation.txt
```

donne:

```
Compiled from "Decompilation.java"
class Decompilation extends java.lang.Object{
public static void main();
Code:
 0:  new #2; //class Decompilation
 3:  dup
 4:  invokespecial #3; //Method "<init>":()V
 7:  astore_0
 8:  aload_0
 9:  invokevirtual #4; //Method test:()I
 12: istore_1
 13: getstatic #5; //Field java/lang/System.out:Ljava/io/PrintStream;
 16: new #6; //class java/lang/StringBuilder
 19: dup
 20: invokespecial #7; //Method java/lang/StringBuilder."<init>":()V
 23: ldc #8; //String Out:
 25: invokevirtual #9; //Method java/lang/StringBuilder.append:...
 28: iload_1
 29: invokevirtual #10; //Method java/lang/StringBuilder.append:...
 32: invokevirtual #11; //Method java/lang/StringBuilder.toString:...
 35: invokevirtual #12; //Method java/io/PrintStream.println:...
 38: return
}
```

Decompilation avec jd-gui

Il existe des outils permettant de décompiler le *bytecode* et de revenir jusqu'au code source, par exemple JD (http://jd.benow.ca/). On obtient, pour la classe **Decompilation** précédente le code:

```
import java.io.PrintStream;

 public class Decompilation
```

```
{
  int test()
  {
    int i = 1;
    int j = 1;
    int k = (byte)(i + j);
    return k;
  }

  public static void main(String[] paramArrayOfString) {
    Decompilation localDecompilation = new Decompilation();
    int i = localDecompilation.test();
    System.out.println("Out: " + i);
  }
}
```

2.3 Chargement dynamique de code

L'utilisation de *bytecode* intermédiaire impose de résoudre les dépendances entre classes lors de l'exécution. Cela n'empêche pas le compilateur de réaliser des vérifications entre classes, par exemple la présence ou non d'une fonction appellée sur un objet de type B depuis un objet de type A.

C'est dans le **CLASSPATH** que la machine virtuelle cherche les classes mentionnées après les directives **import**:

```
import p.Decompilation;
  public class Chargement {
      public static void main() {
          Decompilation d = new Decompilation(); }}
```

A la compilation, on obtient:

```
javac Chargement.java
Chargement.java:1: package p does not exist
import p.Decompilation;
        ^
1 error
```

ce qui montre que le compilateur cherche **Decompilation** dans le sous répertoire **p** du **CLASSPATH**. Si celui-ci est situé dans **unautreendroit**, il faut mettre à jour le **CLASSPATH**:

```
export CLASSPATH=./unautreendroit:$CLASSPATH
```

Le CLASSPATH et les jar

Le **CLASSPATH** donne la liste des emplacements ou la machine virtuelle est autorisée à charger des classes. S'il s'agit d'un nom de répertoire, il désigne la racine de l'arborescence correspondante aux *packages*. Si le **CLASSPATH** contient des fichiers *jar*, les classes sont cherchées et chargées directement depuis l'intérieur de l'archive, la racine de l'arborescence correspondant à la racine de l'archive.

L'exemple suivant permet de charger le fichier **./unautreendroit/p/Decompilation.class**, ou le fichier **p/Decompilation.class** à l'intérieur de archive.jar.

```
export CLASSPATH=./unautreendroit:./archive.jar:$CLASSPATH
```

La création d'un *jar* se fait à l'aide de la commande *jar*:

```
> cd unautreendroit
unautreendroit> jar cvf archive.jar */*.class
manifest ajouté
ajout : p/Decompilation.class (39% compressés)
```

Comme pour la commande *tar*, on peut visualiser un *jar*:

```
jar tf archive.jar
META-INF/MANIFEST.MF
p/Decompilation.class
```

Les jar

La spécification des fichiers *jar* [JS] décrit l'utilisation du Manifest qui permet d'ajouter des informations pour l'utilisation du *jar*. Ce Manifest contient:

- Des informations générales (version, date et auteur, CLASSPATH des ressources requises).
- La classe contenant le **main** si ce jar contient une application qui est lancée via l'exécution de **java -jar x.jar**.
- Des informations pour les applets embarquées dans le *jar*.
- Des informations de signature.

```
Manifest-Version: 2.0
Created-By: 1.0 (JFL)
Main-Class: p.Decompilation

Name: p/Decompilation.class
Digest_Algorithms: MD5
MD5-Digest: base64(ae322ab9de701f1e79bc2040b26349e9)
```

On peut alors construire et exécuter un *jar* comme suit:

```
jar cfm executable.jar Manifest.txt p/Decompilation.class
java -jar executable.jar
Out: 2
```

Le classloader

Le chargement dynamique de classe repose sur l'utilisation d'un chargeur de classe (*classloader*). A partir du nom de la classe, il localise (notamment en parcourant le **CLASSPATH**) ou génère les données qui définissent la classe. A partir des données récupérées, il permet de créer un objet de type **Class**, type que l'on retrouve quand on fait de l'introspection.

La méthode importante du chargeur de classe, qui est invoqué lorsqu'on réaliser une instanciation est:

```
// L'appel permettant de créer l'objet Class à partir de son nom
Class r = loadClass(String className, boolean resolveIt);
```

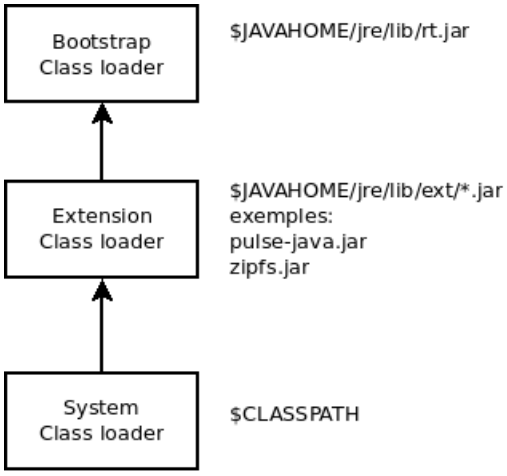
Le booléen permet de spécifier si un lien permanent est créé entre la classe chargée et son nom. Les étapes de l'implémentation de **loadClass** sont:

- vérifications (nom, classe déjà chargée, classe system)
- chargement des données
- définition de la classe (conversion de bytes en Class)
- résolution de la classe (lier la classe au nom)
- on retourne l'objet classe

Hiérarchie de délégation

Les différents chargeurs de classe sont hiérarchisés selon un modèle de délégation. La méthode *loadClass* opère alors de la sorte:

- Si la classe est déjà chargée, elle est retournée
- Sinon, le chargeur délègue le chargement à son parent
- Si le parent ne trouve pas la classe, le chargeur:
  - appelle **findClass** pour la trouver
  - charge la classe le cas échéant



Changer le chargeur de classe

Si l'on créé un chargeur de classe personalisé, il est possible de l'utiliser explicitement ou bien en le précisant au lancement de la machine virtuelle:

```
package cl;
public class CustomCL extends ClassLoader {
      public Class<?> loadClass(String name) throws ClassNotFoundException {
              System.out.println("Custom class loader. Chargement de: " + name);
```

```
            System.out.println("Chargement via mon père...");
            return super.loadClass(name);
    }}
```

```
java -Djava.system.class.loader=cl.CustomCL cl.Main
```

Ce qui donne pour un programme Main:

```
System.out.println("Démarrage...");
System.out.println("Mon classLoader est: " + Main.class.getClassLoader());
Integer i = new Integer(4);
A a = new A();
System.out.println("Fin.");
```

```
Custom class loader. Chargement de: cl.Main
Chargement via mon père...
Démarrage...
Mon classLoader est: sun.misc.Launcher$AppClassLoader@3432a325
Fin.
```

Personnaliser son chargeur de classe: surcharge

```
    public Class<?> loadClass(String name) throws ClassNotFoundException {
            System.out.println("Custom class loader. Chargement de: " + name);
             if (name.startsWith("cl2.")) {
                return getClass(name); // Chargement spécialisé
            }
            System.out.println("Chargement via mon père...");
            return super.loadClass(name);
    }

    private Class<?> getClass(String name)
            throws ClassNotFoundException {
            String file = name.replace('.', File.separatorChar) + ".class";
            byte[] b = null;
            try {
                b = loadClassData(file);
                Class<?> c = defineClass(name, b, 0, b.length);
                resolveClass(c);
                return c;
            } catch (IOException e) { e.printStackTrace(); return null; }
        }

    private byte[] loadClassData(String name) throws IOException {
    InputStream stream = getClass().getClassLoader() .getResourceAsStream(name);
    int size = stream.available();
    byte buff[] = new byte[size];
    DataInputStream in = new DataInputStream(stream);
    in.readFully(buff); in.close();
```

```
    return buff;
}
```

Personnaliser son chargeur de classe: résultat

CL Le branchement dans la méthode **loadClass** permet d'appeler une implémentation spécifique de **getClass**. L'appel à la méthode **defineClass** charge, depuis le fichier, le tableau contenant le code de cette classe particulière. Comme **defineClass** est une méthode *final*, on passe la main à l'implémentation de la JVM pour réaliser cela et l'objet **Class** résultant possède alors un pointeur vers le chargeur de classe qui l'a chargé. Ainsi, tous les objets internes à cette classe pourront être eux aussi chargés par ce chargeur de classe personnalisé. Avec ce chargeur, le Main précédent donne:

```
Custom class loader. Chargement de: cl2.Main
Custom class loader. Chargement de: java.lang.Object
Chargement via mon père...
Custom class loader. Chargement de: java.lang.String
Chargement via mon père...
Custom class loader. Chargement de: java.lang.System
Chargement via mon père...
Custom class loader. Chargement de: java.io.PrintStream
Chargement via mon père...
Démarrage...
Custom class loader. Chargement de: java.lang.StringBuilder
Chargement via mon père...
Custom class loader. Chargement de: java.lang.Class
Chargement via mon père...
Mon classLoader est: cl2.CustomCL@565f0e7d
Custom class loader. Chargement de: java.lang.Integer
Chargement via mon père...
Custom class loader. Chargement de: cl2.A
Fin.
```

2.4 Les conventions pour la JVM

Quelques variables sont importantes pour démarrer un programme java. Ces variables sont aussi utiles pour des scripts shell permettant de lancer des appcliations Java.

- **PATH**: variable pour la recherche des exécutables
- **CLASSPATH**: chargement dynamique des classes
- **JAVA_HOME**: le répertoire racine de la JDK

Sous Windows:

```
set JAVA_HOME=c:\jdk1.6
set JRE_HOME=c:\jdk1.6
set CLASSPATH=.;c:\tomcat\lib\servlet.jar
```

Sous Linux:

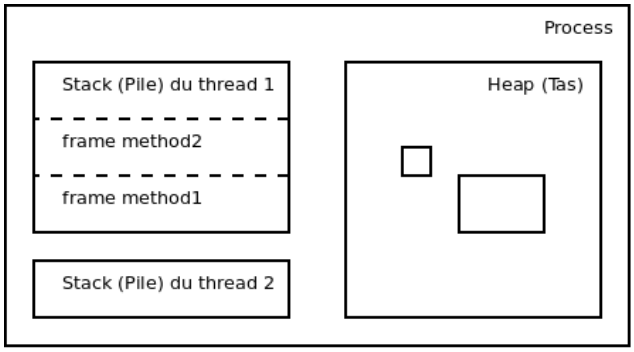
```
export JAVA_HOME=/usr/lib/jvm/java-6-sun
ls /usr/bin/java
/usr/bin/java -> /etc/alternatives/java
ls /etc/alternatives/java
/etc/alternatives/java -> /usr/lib/jvm/java-6-sun/jre/bin/java
```

2.5 Données runtime

JVMS Un programme Java peut être multithreadé. Chaque thread géré par la JVM possède un *program counter register* contenant l'adresse de l'instruction acutellement exécutée.

Chaque *thread* Java possède une pile (*stack*). Une pile stocke des *frames*. Cela permet de gérer les variables locales, résultats partiels, appels de fonction. La mémoire de cette pile n'a pas besoin d'être contigüe.

Au démarrage de la JVM, un tas (*heap*) est créé. Ce tas contient les objets et tableaux instanciés. Ces objets ne sont jamais désalloués (cf *garbage collector*). Ce tas est lui aussi de taille fixe ou dynamique, et non nécessairement contigu.



Changer les tailles par défaut

La spécification permet à une JVM d'avoir une pile de taille fixe (**StackOverflowError**) ou dynamique (**OutOfMemoryError**). On peut spécifier la taille de la pile au lancement de la JVM:

```
java  -Xss1024k maClasse
// peut lever une exeception java.lang.StackOverflowError .
```

On peut aussi préciser la taille du tas:

```
java  -Xmx512M maClasse
```

La JVM contient aussi une zone de méthode (*method area*) qui stocke les structures de classe, attributs, codes de méthodes et constructeurs. Cette zone peut faire partie du tas. La JVM contient un pool de données constantes (*runtime constant pool*) qui est une table par classe contenant des constantes connues à la compilation ou des attributs déterminés au *runtime*. Enfin, la JVM peut contenir des méthodes natives, par exemple écrites en C.

Frames

Une *frame* est utilisée pour stocker des données locales, des résultats partiels, des valeurs de retour pour les méthodes, et les pointeurs vers les objets du tas. Une *frame* n'existe que le temps d'existence d'une méthode et est empilée sur la pile, ainsi que le thread ayant créé cette *frame*. Chaque *frame* possède:

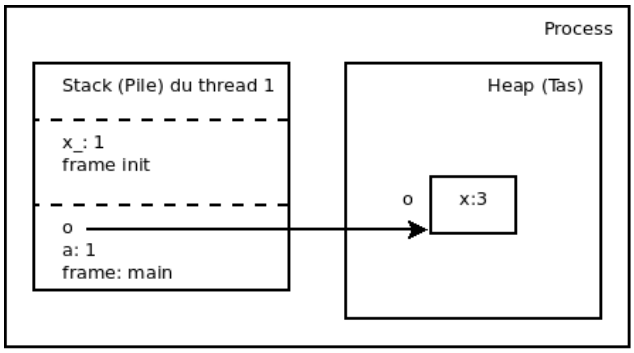
- son propre tableau de variables locales
- sa propre pile d'opérandes (cf plus loin)
- une référence sur le *runtime constant pool* de la classe de la méthode courante

Par conséquent, pour chaque *thread*, une seule *frame* est active à la fois à un moment donné. La *frame* courante cesse de l'être si une méthode est appelée ou si la méthode termine.

Exemple

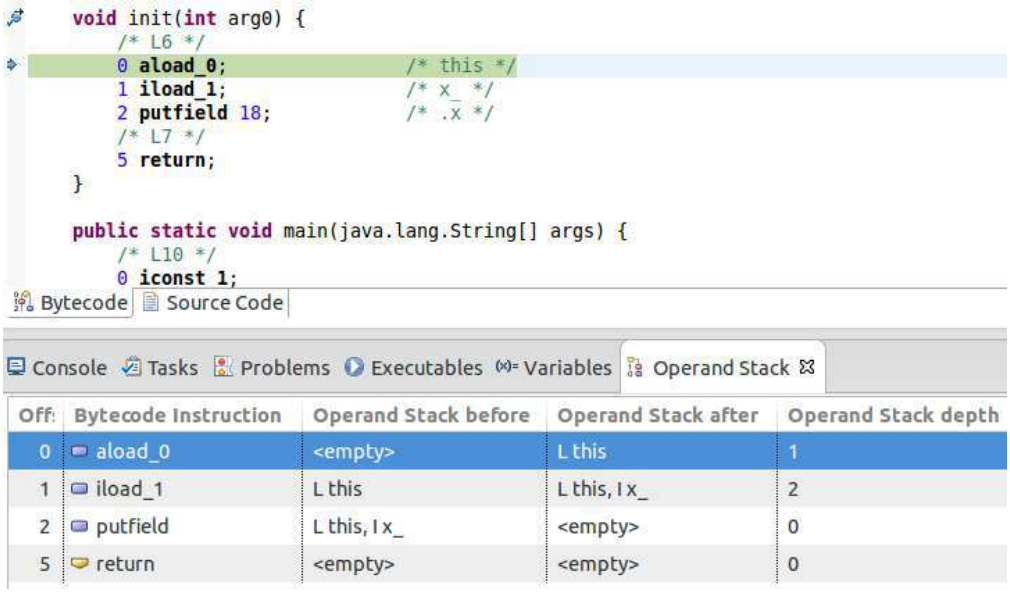
```
public class MyObj {
	public int x;
	void init(int x_) {
		x = x_;
	}

	public static void main(String[] args) {
		int a = 1;
		MyObj o = new MyObj();
		o.init(a);
	}
}
```



Operand stack

La pile d'opérandes d'une fonction contient les variables manipulées par le bytecode. Dans l'exemple suivant, l'opcode **aload_0** pousse l'objet de la variable 0, ici **this**, sur la pile d'opérandes. Puis **iload_1** charge la valeur de la variable locale 1. Puis, **putfield** consomme ces deux valeurs empilées pour modifier le champ numéro 18 de l'objet considéré.



2.6 Garbage collector

GB Dans Java, la gestion du nettoyage de la mémoire est délégué à la JVM au travers du ramasse-miette (*garbage collector*). Le but principal est d'éviter au développeur la gestion de la désallocation des ressources inutilisées, d'éviter des bugs de double désallocation, voire même de perdre du temps à désallouer.

Tous les objets sont stockés dans le tas (*heap*) après des appels à l'opérateur **new**. Les objets sont collectés par le ramasse-miette lorsque:

- le programme ne référence plus l'objet du tas
- aucun objet référencé ne contient de référence vers l'objet du tas

Le ramasse-miette chercher aussi à combattre la fragmentation du tas. La taille du tas, même dynamique, coûte en espace et en temps.

L'implémentation du ramasse-miette est libre. La spécification de la JVM dit juste que le tas doit être *garbage collected*. La difficulté d'implémentation du ramasse-miette réside dans le fait qu'il faut garder une trace des objets utilisés ou non, puis les détruire. Cela coûte d'avantage en temps CPU qu'une désallocation manuelle.

A la destruction, la JVM appelle la méthode **finalize**, dernière méthode étant appellée:

```
protected finalize() throws Throwable { ... }
```

Atteignabilité

La manière la plus simple de déterminer les objets à collecter est de calculer l'atteignabilité d'un objet. Un objet est dit attaignable ssi il existe un chemin de références depuis les racines (les références du programme) jusqu'à cet objet.

Tous les objets sont situés dans le tas. Les variables locales sont, quant à elle, sur la pile. Chaque variable locale est soit un type primitif, soit une référence d'objet. Ainsi, les racines sont donc l'union de de toutes les références des piles. Les racines contiennent aussi les objets constants comme les **String**

dans le *runtime constant pool*. Par exemple, le *runtime constant pool* peut pointer sur des **String** de la pile contenant le nom de la classe ou des méthodes.

Deux grandes familles de ramasse-miettes peuvent être distingués:

- ramasse-miettes par décompte de références (*reference counting*)
- ramasse-miettes par traces (*tracing*)

Le ramasse-miette par décompte de référence collecte les objets lorsque le compteur tombe à zéro. Le problème d'un tel rammasse-miette est l'*overhead* d'incrémentation/décrémentation mais surtout la non détection de cycles d'objets. Le ramasse miette par traces parcourt le graphe des objets et pose une marque sur chacun d'eux. Puis les objets non marqués sont supprimés du tas. On appelle ce processus *mark and sweep*.

Quel que soit le type de ramasse-miette, la présence de méthodes de finalisation impacte la phase *sweep*. Les objets non marqués sans méthode *finalize* peuvent être collectés à moins qu'ils ne soient référencés par des objtes non marqués ayant une méthode *finalize*.

Fragmentation

Pour combattre la fragmentation, deux stratégies similaires permettent de réduire la fragmentation de la pile après la désallocation d'objets:

- Défragmentation par compactage (*compacting*)
- Défragmentation par copie (*copying*)

La défragmentation par compactage déplace les objets marqués d'un côté de la pile, afin d'obtenir une large plage contigüe de l'autre: on compacte les objets d'un côté. En insérant un niveau d'indirection dans les adresses du tas d'objets, on évite le rafraichissement de toutes les références de la pile.

La défragmentation par copie déplace tous les objets dans une nouvelle zone de la pile. L'intégralité des objets étant déplacés, ils sont assurément contigus. Pour réaliser cette opération, une traversée totale des objets depuis les racines est obligatoire. Dans le cas d'un ramasse-miette de ce type, on peut combiner la recherche d'objets à collecter et la défragmentation: l'ancienne zone mémoire est donc par conséquent libre ou contenant des objets à désallouer. Ce processus élimine la phase de marquage et de collecte (*mark and sweep*).

Une implémentation classique de ce procédé est appellé le ramasse-miette *stop and copy*. La pile est divisée en deux régions et les allocations se font dans l'une d'elle jusqu'à épuisement. Le programme est alors arrêté et les objets copiés dans l'autre partie à partir de la traversée du graphe d'objets. Le programme est relancé et les allocations se font dans la nouvelle région.

Ramasse-miette générationnel

La durée de vie des objets impacte l'efficacité du ramasse-miette. De nombreux objets (plus de 98% dans des mesures expérimentales [GCH]) ont une durée de vie très courte: *most objects die young*. Le ramasse-miette par copie est dans ce cas très efficace car les objets meurrent car ils ne sont pas visités lors de la copie. Si l'objet survit après le premier passage du ramasse-miette, il y a de grandes chances qu'il devienne permanent. Le ramasse-miette par copie est peu performant sur ce type d'objets. Inversement, le ramasse-miette par marquage-compactage est performant sur les objets à durée de vie longue puisqu'ils sont compactés d'un côté de la pile et ne sont ensuite plus recopiés. Les ramasse-miettes par marquage sont pour leur part plus longs à exécuter puisqu'ils doivent examiner de nombreux objets dans la pile.

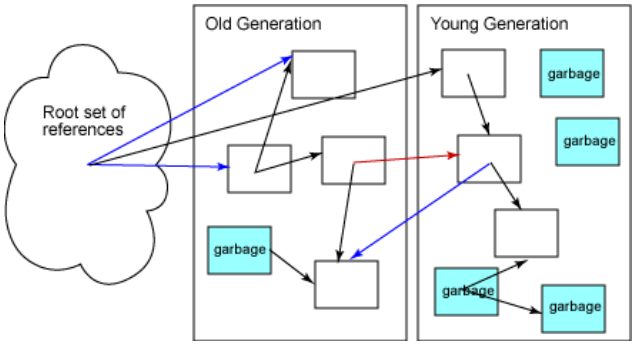
Un ramasse-miette générationnel se base sur le principe de ségrégation des générations: certains objets sont dits jeunes et d'autres sont promues à un niveau de génération supérieur. Pour simplifier on peut considérer deux génération: *young generation* et *old generation*.

Lorsqu'une allocation échoue, un trigger *minor collection* est déclenché, qui engendre la collection de la génération la plus jeune. Cette collection peut être très rapide et récupérer un espace mémoire conséquent. Si la récupération est suffisante, le programme peut continuer, sinon une autre génération est attaquée.

Références inter-générationnelles

La collection par générations successives peut engendrer la collection d'un objet jeune alors qu'un objet vieux pointe encore sur cet objet. En effet, le tas est divisé en plusieurs générations et tous les types de ramasse-miette par copie ou marquage doivent parcourir les objets depuis les racines. L'algorithme de parcourt se limite donc à une génération du tas, évitant de parcourir l'intégralité du tas. Il peut alors considérer un objet comme à collecter car non marqué, alors qu'un objet de la veille génération pointe encore dessus.

Le ramasse-miette doit donc construire la liste des références inter-générationnelles, par exemple lors de la promotion d'un objet jeune dans la vieille génération (méthode la plus efficace). Lors de l'analyse de la *minor collection*, ces références inter-générationnelles sont alors considérées comme des références racines, résolvant le problème évoqué ci-avant.



3 Les processus légers: thread

Contributeurs

Michel Cosnard, Jean-Francois Lalande, Fabrice Peix

3.1 Historique

La notion de processus est apparue lorsque les ordinateurs sont devenus assez puissants pour permettre l'exécution de plusieurs programmes de façon concurrente. En effet lorsque plusieurs programmes s'exécutent simultanément, il est important d'assurer que la défaillance d'un seul programme n'entraîne pas la défaillance de tout le système. Dans cette optique, la notion de processus a permis l'isolement de chaque programme, empêchant ainsi un processus de modifier les données d'un autre processus.

Dans un même temps, les modèles de programmation évoluaient et très vite le désir d'exprimer de la concurrence au sein d'un même programme s'est fait ressentir. La première réponse à cette demande a été l'utilisation de plusieurs processus associés à différents moyens de communication (mémoire partagée, sémaphores, pipes, et puis socket). Bien que rapidement mise en øeuvre, cette solution présentait plusieurs inconvénients majeurs. Le premier est que l'ensemble des moyens de communication demande un passage en mode noyau (ce qui est coûteux en temps). Le second inconvénient est que la création d'un processus est relativement longue. Les autres problèmes étaient le nombre limité de processus au sein des systèmes d'exploitation et une utilisation mémoire non négligeable. Ainsi, bien que la notion de processus permette effectivement de réaliser de la programmation concurrente elle est très restrictive dans ce qu'elle permet de faire.

Afin de pallier à l'ensemble de ces problèmes, il fallait définir une nouvelle entité. La solution proposée a été d'associer plusieurs exécutions concurrentes à un même processus. Pour cela on a donc défini les informations minimales nécessaires (état des registres, pile,) permettant d'avoir plusieurs fils d'exécution au sein d'un même processus, les threads (ou processus légers) étaient nés.

3.2 Les processus lourds et légers

Les processus (ou processus lourds) sont en général plus proches du système d'exploitation. Ils s'agit par exemple d'un processus unix. Ce type de processus obtient un espace mémoire dédié, une pile, un nouveau jeu de variable, toutes ces données restant inaccessibles d'un processus à l'autre.

L'utilisation du **fork()** de POSIX est un exemple de processus lourd. Le seul cas de partage entre ces processus est le code et les accès aux descripteurs de fichiers. Par contre, chaque processus obtient une pile, l'état des registres du CPU et doit redéfinir le masquage pour l'interception des signaux. Cependant, utiliser des processus lourds communiquants est difficile à mettre en oeuvre, surtout à cause de l'accès à la mémoire partagée. Pour partager les données, on préfère utiliser les processus légers.

Un processus léger est un flot d'exécution partageant l'intégralité de son espace d'adressage avec d'autres processus légers. Il est en général géré à un plus haut niveau, par rapport au système d'exploitation. Les différents processus légers peuvent s'exécuter alors dans un même processus lourd, mais de manière concurrente.

Création de processus légers en Java

En Java, ces processus légers sont les appelés *thread*. En pratique, on peut utiliser la classe Thread et Runnable, par exemple:

```
public class MyThread extends Thread {
public void run() { ... }

public class MyRunnable implements Runnable
public void run() { ... }
```

```
MyThread t = new MyThread();
t.start();

Thread t = new Thread(new MyRunnable);
t.start();
```

Un thread est actif lorsqu'il est en cours d'exécution de la méthode **run()**. Avant cet état, il est existant (ce qui correspond au new...). Il peut être éventuellement bloqué ou interrompu (**interrupt()**). Lorsque le thread atteint l'accolade fermante du **run()**, il a atteint sa fin d'exécution et peut être collecté par le garbage collector, à condition que l'utilisateur redonne la référance correspondante.

En cours d'exécution, il peut volontairement laisser la main à d'autres processus en utilisant les méthodes *sleep()* ou **pause()**.

Création de processus lourds en java

Il existe aussi la possibilité de créer des processus lourds, depuis la machine virtuelle. Cela revient à un **fork()** suivit d'un **execXX** en C. Les classes à utiliser sont les classes Runtime et Process, par exemple:

```
Runtime.getruntime().exec("cmd");
```

3.3 Les schedulers

Plusieurs processus lourds peuvent s'exécuter en concurrence sur un ou plusieurs processeurs. Le système d'exploitation donne la main alternativement à chaque processus lourd. On parle de *scheduler* pour exprimer le fait qu'on doit ordonnancer la prise du fil d'exécution par chaque processus lourd, alternativement.

A l'intérieur d'un même processus lourd, le scheduler peut avoir des comportements différents pour la politique de prise du fil d'exécution des processus légers. Le choix de cette politique peut poser les problèmes de concurrence (dead lock, données corrompues, ...) puisque le scheduler peut par exemple décider arbitrairement d'arrêter l'exécution d'un thread pour donner la main à un autre.

Les systèmes de gestion de threads se subdivisent en deux catégories :

- les schedulers coopératifs
- les schedulers préemptifs

Scheduler coopératif

Dans ces systèmes, les threads s'exécutent jusqu'à ce qu'ils décident de relâcher explicitement le processeur (instruction yield) pour laisser un autre thread s'exécuter. En d'autres termes, l'ordonnancement des threads doit être réalisé par le programmeur.

Avantages: Le principal avantage de ce modèle est la simplicité de la gestion des données partagées. En effet, puisque les threads rendent explicitement la main, le problème des données partagées pouvant se trouver dans un état incohérent ne se pose pas.

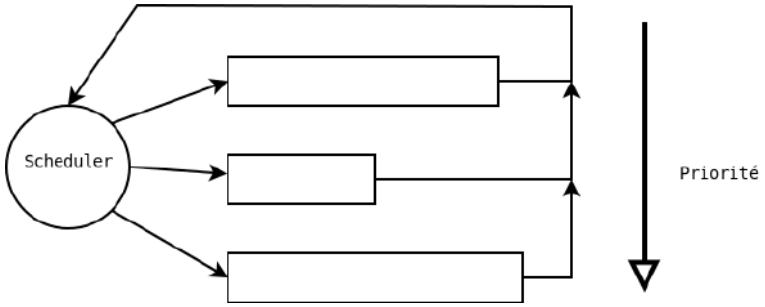
Inconvénients: Le principal désavantage de telles implémentations est l'énorme difficulté voire l'impossibilité de le faire fonctionner sur plusieurs processeurs. On notera que la gestion de l'ordonnancement n'apporte pas que des avantages et pose notamment le problème de choisir intelligemment les endroits où réaliser un yield. Enfin, de telles implémentations demandent impérativement l'utilisation d'entrées/sorties non bloquantes (pas toujours disponibles) pour éviter que l'ensemble du processus soit bloqué.

Le mode coopératif exige de rendre explicitement le fil d'exécution du scheduler. Le scheduler est donc passif: il attend qu'un processus rende la main pour ensuite choisir le prochain processus.

FCFS

L'algorithme FCFS (*First-Come-First-Served*) (premier arrivé, premier servi) est le plus simple des algorithmes d'ordonnancement. Le premier processus qui entre dans la file est le premier à bénéficier du processeur. Ce système est par contre très peu rentable. Les processus longs sont favorisés par rapport aux processus courts. Le type de gestion est non-préemptif, car rien n'est prévu par le système d'exploitation pour déloger le processus du cpu, ou favoriser un processus dans la file.

Sur la figure ci-dessous, le scheduler choisit tout d'abord par priorité décroissante le processus à exécuter. Puis, à priorité égal, il donne la main séquentiellement à chaque processus.



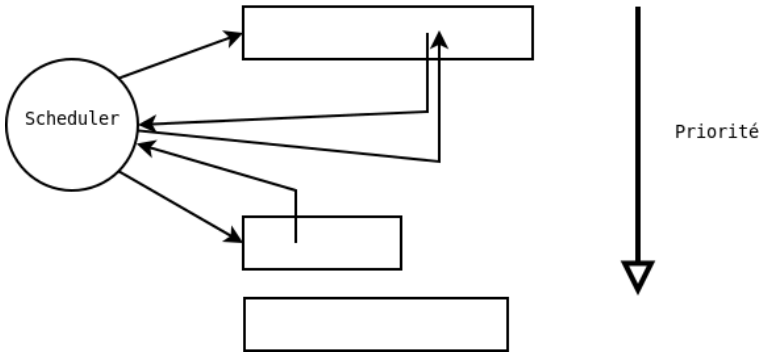
Scheduler préemptif

Dans ces systèmes, il existe un thread particulier ou bien une partie du système d'exploitation (que l'on nomme scheduler ou ordonnanceur) qui est chargé de décider quel thread doit s'exécuter. Son rôle est de répartir au mieux (en fonction des priorités par exemple) les ressources du système. De ce fait le programmeur n'a pas à s'occuper de l'ordonnancement.

Avantages: Les avantages de ce modèle sont multiples. La charge de définir un ordonnancement étant assurée par une entité externe la rend dynamique. Ceci a pour conséquence de la rendre beaucoup plus souple. De plus, l'utilisation d'entrées/sorties bloquantes ne pose plus aucun problème.

Inconvénients: Le principal désavantage de ce modèle est la difficulté que pose la gestion des données partagées. En effet, puisque l'ordonnancement est externalisé aucune supposition ne peut être faite sur l'ordre d'exécution des threads et sur les interruptions du scheduler qui peuvent survenir à tout moment.

Pour un scheduler préemptif, le scheduler peut prendre la main pendant l'exécution d'un thread et donner le fil d'exécution à un thread en cours d'exécution. Sur la figure ci-dessous on montre le changement du fil d'exécution en plein milieu des processus.



Exemple d'algorithme préemptif: round-robin

L'algorithme *round-robin* est spécialement adapté aux systèmes en temps partagé. On définit un quantum de temps (time quantum) d'utilisation de l'unité centrale. La file d'attente des processus éligibles est vue comme une queue circulaire (fifo circulaire). Tout nouveau processus est placé à la fin de la liste.

De deux choses l'une, soit le processus actif rend le fil d'exécution avant la fin de sa tranche de temps (pour cause d'entrée/sortie ou volontairement) soit il est préempté, et dans les deux cas placé en fin de liste. Un processus obtiendra le processeur au bout de $(n-1)T$ secondes au plus n nombre de processus et T longueur du quantum de temps): la famine est donc assurément évitée.

Remarquons que si le quantum de temps est trop grand, round-robin devient équivalent à FCFS. De l'autre coté si le quantum de temps est très court, nous avons théoriquement un processeur n fois moins rapide pour chaque processus (n nombre de processus). Malheureusement si le quantum de temps est court, le nombre de changements de contexte dûs à la préemption grandit, d'où une diminution du taux utile, d'où un processeur virtuel très lent. Une règle empirique est d'utiliser un quantum de temps tel que 80% des processus interrompent naturellement leur utilisation de l'unité centrale avant l'expiration du quantum de temps.

Choix d'ordonnancement dans Java

Le choix de a été de ne pas faire de choix. En effet, contraint par l'objectif d'être multi-plateformes le langage Java a décidé de ne pas faire de spécifications trop contraignantes sur l'implémentation et le fonctionnement des threads. Java ne définit que quelques règles sur la nature des threads et la façon de les ordonnancer.

Toutefois, La spécification Java des threads permet une implémentation préemptive, ce qui permet d'effectuer un mapping des threads sur les threads du système d'exploitation hôte. Dans ce cas le scheduler peut-être par exemple du type round-robin.

La conséquence de ceci est que si l'on désire être réellement portable au sens de la spécification Java, il faut que les programmes puissent fonctionner (c'est à dire avoir un fonctionnement identique) quelles que soient les spécificités de l'implémentation. Ceci demande donc de faire des appels à la méthode **yield**, dans le cas d'une implémentation coopérative, sans pour autant profiter des avantages de cette implémentation (une meilleure maîtrise des données partagées).

Toutefois, la majorité des machines virtuelle que l'on trouve aujourd'hui sur les ordinateurs de bureau utilisent des threads préemptifs, à la différence de ce qui existe dans les Cards et dans certains systèmes embarqués. On peut donc raisonnablement partir du principe que, pour la programmation d'applications destinées au desktop, on considère que l'implémentation est préemptive.

Priorité et interruptions

Il est possible de changer la priorité d'un thread afin qu'il ait une priorité particulière pour accéder au processeur. Le thread de plus forte priorité accède plus souvent au processeur. Par défaut, un thread a la priorité de son père. Pour changer la priorité d'un thread on utilise la méthode suivante: **setPriority (int prio)**.

Seule la possibilité d'interrompre un processus a été gardé dans Java 1.5. Les méthodes permettant de stopper ou de suspendre un thread sont dépréciées, pour éviter de laisser des objets en cours de modification dans un état "non-cohérent". La méthode *interrupt()* positionne un statut d'interruption et peut lever des exceptions si le thread était en pause ou en cours d'utilisation d'entrées/sorties.

Fil d'exécution et atomicité

Les instructions des différentes threads peuvent être exécutées à tour de rôle, dans un ordre qui dépend du choix du scheduler. Le scheduler donne le fil d'exécution du processeur à un thread pendant un certain temps, puis reprend la main, pour réévaluer le prochain thread qui prendra de nouveau le fil d'exécution. Dans le cas où plusieurs processeurs sont disponibles, le scheduler s'occupe de répartir *n* threads sur *m* processeurs.

On parle d'atomicité d'une instruction lorsque le scheduler ne peut pas prendre la main au cours de l'exécution de cette instruction. En Java, l'atomicité n'est garantie que sur les affectations (sauf pour les *double* et *long*). Sinon, le scheduler peut interrompre l'exécution d'une instruction et lui redonner la main à posteriori pour terminer son accomplissement. Les instructions atomiques classiques sont par exemple l'ouverture d'un bloc synchronizé, la prise d'un jeton d'un sémaphore, ou l'acquisition d'un lock.

Une erreur classique en programmation est de croire que le **if** est atomique ce qui peut conduire à des bugs liés à la concurrences de threads:

```
boolean array_access = false;
if (!array_access)
{
   array_access = true;
   ... // do the job
}
array_access = false;
```

Rendez-vous et pauses

Un thread peut attendre qu'un autre thread se termine. On utilise la méthode **join()** pour attendre la terminaison d'un autre fil d'exécution concurrent avant de continuer son propre fil d'exécution.

```
MyThread m1 = new MyThread();
MyThread m2 = new MyThread();
m1.start();
m2.start();
m1.join();
m2.join();
```

En cas d'attente, le thread ne consomme pas (ou peu) de temps CPU. On peut soi-même demander explicitement l'endormissement d'un thread pendant un temps donné en utilisant **Thread.sleep(int temps)**, ce qui endort le thread courant (donc soi-même puisque c'est notre "moi-même" qui exécute cet appel). On peut aussi rendre explicitement la main au scheduler avec **yield()**, ce qui peut-être particulièrement utile lorsqu'on programme par exemple des interfaces graphiques.

```
// Affichage graphique
...
// Fin affichage graphique
Thread.yield();
```

3.4 Gestion de tâches

Certaines applications nécessitent de traiter des ensembles de tâches sur différents threads, souvent à des fins de performances. Il peut aussi s'agir de découper un calcul en plusieurs sous calculs, notamment si l'on projette ensuite de réaliser ce calcul de manière distribuée. Un certain nombre de classes de haut niveau sont dédiées à la gestion de ces tâches et épargnent ce travail de gestion à l'aide des primitives de base de la concurrence.

L'interface **Future** permet d'attendre ou de manipuler une tâche encore en cours d'exécution et dont on attend le résultat. Les principales méthodes utilisées sont **get(), cancel(), isCancelled(), isDone()**. La méthode **get()** retourne le résultat de la tâche et reste bloquante si le résultat n'est pas encore calculé. Les autres méthodes permettent de contrôler la tâche et éventuellement d'annuler son exécution.

Pool de threads

Pour éviter de devoir créer un thread à chaque fois qu'une nouvelle tâche doit être exécutée, par exemple dans le cas classique d'un serveur web. On utilise dans ce cas un pool de thread, qui n'est rien d'autre qu'une collection de threads qui se nourrit de tâches disponibles dans une queue. L'interface **Executor** permet de spécifier les méthodes classiques d'une tâche et l'on peut ensuite décider de différentes polique d'exécutions des tâches en agissant sur des objets implémentant **Executor**. Les politiques proposées dans Java 1.5 sont les suivantes:

Excecutors.newCachedThreadPool(): un pool de thread de taille non limité, réutilisant les threads déjà créés et en créant de nouveau au besoin. Après 60 secondes d'inactivité, le thread est détruit.

Excecutors.newFixedThreadPool(int n): un pool de thread de taille n. Si un thread se termine prématurément, il est automatiquement remplacé.

Excecutors.newSingleThreadExecutor(): crée un seul thread ce qui garantit la séquentialité des tâches mises dans la queue de taille non bornée.

Si l'on utilise des pools de threads de taille bornée, on peut se demander quel est la taille optimale de l'ensemble. La loi d'Amdahl donne une bonne idée du dimensionnement optimal pour une utilisation maximale d'un système multiprocesseur. Si *WT* est le temps moyen d'attente d'une tâche et *ST* son temps moyen d'exécution, avec *N* processeurs la loi d'Amdahl propose un ensemble de threads de taille *N/(1+WT/ST)* Ceci ne tient évidemment pas compte des aléas temporel dus par exemple aux entrées/sorties.

Rendez-vous de threads

Lorsque l'on souhaite synchroniser plusieurs threads entre eux, c'est à dire faire en sorte que les threads s'attendent les uns les autres (par exemple après un calcul distribué), on peut utiliser la classe **CyclicBarrier** qui bloquent les threads à une barrière (l'instruction **CyclicBarrier.await()** et les débloquent quand tous les threads y sont rendus. Cette barrière est cyclique car elle peut-être réutilisée pour une prochaine utilisation (elle est réinitialisée).

La classe **CountdownLatch** est très similaire à **CyclicBarrier** mais elle est plus adaptée à la coordination d'un groupe de threads ayant à traiter un problème divisé en sous problèmes. Chaque thread décrémente un compteur en appellant **countDown()** après avoir traité une des tâches, puis est bloquée à l'appel de **CyclicBarrier.await()**. Le débloquage des threads ne se fait que lorsque le compteur atteint 0.

Enfin, la classe *Exchanger* permet de réaliser des échanges de données entre deux threads coopératifs. Il s'agit d'une barrière cyclique de 2 éléments avec la possibilité supplémentaire de s'échanger des données lorsqu'ils atteignent la barrière.

4 L'exclusion mutuelle des threads

Contributeurs

Michel Cosnard, Jean-Francois Lalande, Fabrice Peix

4.1 Les problèmes liés à la concurrence

L'exécution concurrente de threads ayant accès au même espace mémoire peut provoquer des écriture/lecture entrelacées rendant les données incohérentes ou faussant le cours "voulu" de l'exécution. On tente alors de protéger les données des accès concurrents en excluant les threads mutuellement. Techniquement, il s'agit de poser des **locks** ou des moniteurs.

L'exemple classique consiste à considérer deux threads et un objet stockant des valeurs dans un tableau (par exemple à l'aide de la classe **Vector**). Un des threads réalise des écritures dans cet objet pendant que l'autre réalise des parcourts. Suivant la manière dont le scheduler donne la main aux threads et suivant la fréquence à laquelle chaque thread accède à l'objet, il est possible que le thread réaliser une opération d'écriture *pendant* que l'autre thread est en train de réaliser un parcourt. Généralement, cela génère la levée de l'exception **ConcurrentModificationException**.

4.2 Locks et moniteurs

Les systèmes préemptifs demandent l'utilisation de lock afin d'éviter les problèmes posés par un accès concurrent à une même ressource partagée. On peut distinguer deux méthodes équivalentes, mais ayant des sémantiques différentes, permettant de sérialiser les accès à une ressource. La première déjà évoquée est le lock qui permet (en l'associant à une donnée) de s'assurer qu'il n'y aura pas d'accès concurrent à cette donnée partagée. La procédure est assez simple :

- Acquisition du lock
- Utilisation des données partagées
- Relâchement du lock

Dans cette approche on restreint l'accès à la donnée, il faut donc prendre le lock avant chaque utilisation de cette donnée. Dans Java 1.5, on dispose de la classe *Lock* pour cela.

La deuxième approche (moniteur) ne consiste plus à restreindre l'accès à une donnée mais au code qui modifie cette donnée. C'est la solution retenue par pour réaliser la protection des données partagées. Dans cette approche, on sérialise l'accès à une portion de code.

Un moniteur se pose à l'aide du mot clef *synchronized*: on dit aussi que le bloc est synchronisé. Si c'est la méthode tout entière sur laquelle on pose un moniteur sur *this*, la méthode est dite "synchronisée".

Enfin, un objet peut être totalement synchronizé: toutes ses méthodes le sont.

Exemple de blocs synchronisés

L'exemple ci-dessous montre comment réaliser la gestion d'un tableau dynamique à accès concurrents à l'aide de blocs (ou méthodes) synchronisés.

```
public class TableauDynamique {
  private Object[] tableau; // le conteneur
  private int nb; // la place utilisée

  public TableauDynamique (int taille) {
    tableau = new Object[taille];
    nb = 0; }
  public synchronized int size() {
    return nb; }
  public synchronized Object elementAt(int i) throws NoSuchElementException {
    if (i < 0 || i = nb)
      throw new NoSuchElementException();
    else
      return tableau[i];
  }
  public void append(Object x) {
  Object[] tmp = tableau;
  synchronized(this) { // allouer un tableau plus grand si nécessaire
    if (nb == tableau.length) {
      tableau = new Object[3*(nb + 1)/2];
    for (int i = 0; i < nb; ++i)
      tableau[i] = tmp[i];
    }
    tableau[nb] = x;
    nb++;
  }
}}
```

Méthodes synchronisées

Une méthode synchronisée peut appeler une autre méthode synchronisée sur le même objet sans être suspendue. En effet, le thread courant possède le moniteur sur l'instance de la classe dont on exécute la méthode: il peut donc appeller une autre méhode du même objet, ce moniteur étant déjà acquis.

Les méthodes statiques peuvent aussi être synchronisées mais la synchronisation se fait sur l'objet de type **Class**. Si une méthode non statique veut synchroniser son accès avec une méthode statique elle doit se synchroniser relativement au même objet. Elle doit donc contenir une construction de la forme : **synchronized(this.getClass()){...}**.

Le choix de l'objet sur lequel le moniteur est posé est primordial. Choisir *this* comme objet pour l'appel à *synchronized(this)* sans réfléchir et vérifier que l'exclusion est bien effective est une grossière erreur. En général, l'objet à passer à l'appel à synchroniser est l'objet à protéger et qui sera commun à tous les threads. Il est aussi possible de créer un objet ad-hoc spécialement concu pour la synchronisation. Dans un certain sens, on revient alors à l'utilisation des classes implémentants *Lock*. Typiquement, un objet de type **Ressource** peut être utilisé par un thread pour exclure un autre threads:

```
// Quelque part dans le thread principal
Ressource r = new Ressource();
```

```
// Dans chaque portion de code exécutée dans des threads:
synchronized(r) {
  // ici, j'exclu les threads souhaitant poser un moniteur sur r
  ... }
```

Locks

L'interface **Lock** fournit les primitives nécessaire pour manipuler différentes classes de locks. Dans Java 1.5, ces classes ont un comportement spécifique qu'il faut adapter au type de modèle traité. Nous donnons par la suite deux exemples de locks particuliers.

```
Lock l = ...;
l.lock();
try {
    // access the resource protected by this lock
} finally {
    l.unlock();
}
```

ReentrantLock

Ce lock est dit réentrant dans le sens ou un thread possédant déjà ce lock et le demandant à nouveau ne se bloque pas lui-même. Ce type de blocage ne peut pas survenir avec un moniteur puisque un moniteur déjà posé sur une ressource est considéré comme acquis si un autre appel survient essaynt un moniteur sur cette même ressource.

ReadWriteLock

Ce lock fournit une implémentation permettant d'avoir une politique d'accès à une ressources avec plusieurs lecteurs et un unique écrivain. Les accès en lecture sont alors très efficace car réellement concurrents.

4.3 Problème de vivacité ou liveness

A cause de l'exclusion mutuelle qui tente d'assurer que rien de faux n'arrive, il est courant que rien n'arrive du tout: c'est le problème de vivacité ou liveness.

Les problèmes de liveness peuvent être de quatre types :

Famine (contention) Bien que la thread soit dans une état ou elle puisse s'exécuter, une autre thread (plus prioritaire par exemple) l'empêche toujours de s'exécuter.

Endormissement (dormancy) Un thread est suspendu mais n'est jamais réveillée.

Interblocage (deadlock) Plusieurs threads s'attendent de façon circulaire avant de continuer leur exécution.

Terminaison prématurée Un thread (à cause d'un ordonnancement particulier) reçoit un stop() prématurément.

Le problème de liveness le plus courant est le problème de **deadlock**. Un thread1 attend une "chose" que doit libérer un thread2 qui attend lui-même une chose que doit libérer le thread1. Une stratégie simple (malheureusement pas toujours applicable) pour éviter les deadlocks consiste à numéroter les "choses" à attendre et à toujours les prendre dans le même ordre.

Exemple de deadlock

Le système de lock peut provoquer des interblocages ou deadlocks. L'exemple suivant utilise de objets de type **Cell**, cell1 et cell2. Si le thread 1 exécute swapValue sur cell1, il prends le moniteur sur cell1. De même le thread 2 peut prendre le moniteur sur cell2 où cell2=other dans le code cell1. Dans ce cas, à l'instruction **other.getValue()**, cell1 attend d'avoir le moniteur sur cell2 et inversement...

```
class Cell {
  private long value_;

  synchronized long getValue() { return value_;}
  synchronized void setValue(long v) {value_ = v;}

  synchronized void swapValue(Cell other) {
    long t = getValue();
    long v = other.getValue();
    setValue(v);
    other.setValue(t);
  }
 }
```

Solution dite d'ordre

Une solution classique dite "d'ordre" permet de résoudre ce problème. Elle utilise la notion de précédence des locks: il s'agit de toujours prendre le lock sur l'objet ayant le plus grand hashcode, avant de passer au suivant.

```
void swapValue(Cell other) {
 if (other == this) return; // alias check

 Cell fst = this; // order via hash codes
 Cell snd = other;
 if (fst.hashCode() > snd.hashCode()) {
   fst = other;
   snd = this;
 }
 synchronized(fst) {
   synchronized (snd) {
     long t = fst.value;
     fst.value = snd.value;
     snd.value = t;
   }
 }
}
```

Résoudre la famine ou l'endormissement

Une autre approche qui peut souvent porter ces fruits pour éliminer les problèmes de famine ou d'endormissement est la stratégie "diviser pour régner". Plutôt que d'avoir un unique moniteur, celui-ci est subdivisé en plusieurs sous-moniteurs contrôlant chacun l'accès à une ressource particulière. Cela permet de minimiser le temps d'attente des threads et augmente l'efficacité d'exécution du programme.

Malheureusement, la multiplication des moniteurs favorise l'apparition de deadlocks. Si un thread doit poser plusieurs moniteurs sur plusieurs ressources, la possibilité d'un deadlock apparait. Une stratégie de résolution consiste à ne pas attendre un moniteur s'il est déjà pris: si un moniteur est occupé, on relâche alors tous les moniteurs déjà pris et on diffère l'action à réaliser.

Enfin, la meilleure stratégie pour éviter tout problème de liveness est de supprimer au maximum les exclusions mutuelles, quand cela est possible. Avant de supprimer l'exclusion mutuelle d'une partie de code il est très important de penser aux deux règles suivantes :

- Ne jamais avoir d'idée préconçue sur la progression relative de deux threads. En particulier toujours vérifier si le code est valide dans le cas ou l'autre thread n'a même pas démarré ou si il est déjà terminée.
- Supposer que le scheduler peut libérer le processeur du thread courant à n'importe quel point du code.

Les sémaphores

Les sémaphores permettent d'étendre la notion de *lock* en introduisant la manipulation de jetons (ou de permissions). Un sémaphore permet d'acquérir et de relâcher un jeton. Si un thread ne peut acquérir de jetons, il reste bloqué sur l'acquisition jusqu'à ce qu'un autre thread relâche un jeton. D'autre primitives permettent notamment de connaître le nombre de jeton disponibles, d'essayer d'acquérir un jeton si possible mais de manière non bloquante.

Il n'y a aucune contrainte sur l'identité des objets et des threads qui acquièrent ou relâchent un jeton d'un sémaphore. C'est donc un système très flexible mais générateur de bugs, puisqu'on peut très facilement oublier de relâcher un jeton et bloquer d'autres threads. Pour éviter ce genres de problèmes, on peut déjà essayer d'acquérir et de relâcher des jetons au sein d'une même classe pour éviter la dispersion du code de gestion du sémaphore dans de multiples objets.

Notons enfin qu'un sémaphore ayant un seul jeton (ou permis) est appelé un sémaphore binaire et peut servir de lock d'exclusion mutuelle. La différence entre un sémaphore binaire et un *lock* réside dans le fait qu'un sémaphore binaire peut être relâché par un thread différent, ce que ne permet pas toujours de faire un *lock* (**Lock** n'est qu'une interface). Ceci peut-être utile dans certains cas pour éviter des *deadlock*.

```
Semaphore s = new Semaphore(3, true);
s.acquire() // prend un jeton
s.release() // relache un jeton
```

Exemple d'utilisation de sémaphores

Dans l'exemple suivant 100 jetons sont disponibles. La classe **Pool** fournit des objets à l'utilisateur mais n'est pas censé en fournir "trop". Il serait par exemple interdit de donner le 101ème objet contenu dans le *pool* d'objet. En programmation non-concurrente, un simple test suffirait. En programmation concurrente, l'utilisation d'un sémaphore devient obligatoire !

```
class Pool {
   private static final MAX_AVAILABLE = 100;
   private final Semaphore available =
                 new Semaphore(MAX_AVAILABLE, true);

   public Object getItem() throws InterruptedException {
     available.acquire();
     return getNextAvailableItem();
```

```
    }

    public void putItem(Object x) {
      if (markAsUnused(x))
        available.release();
    }
}
```

4.4 Synchronisation

Pour les applications concurrentes il est souvent très important de pouvoir synchroniser des threads entre eux. Ceci particulièrement utile pour s'assurer qu'un autre thread est bien dans un certain état (terminé en particulier).

Java propose un mécanisme d'attente/notification. Les primitives suivantes sont des méthodes de la classe java.lang.Object qui sont utilisées pour la synchronisation. Elles doivent être appelées sur un objet associé à un moniteur détenu au moment de l'appel par le thread courant:

public void wait() throws InterruptedException suspend l'activité de la thread courante, "libère" le moniteur et attend une notification sur le même objet. Quand la thread a reçu une notification elle peut continuer son exécution dès qu'elle a "repris" le moniteur du bloc synchronisé courant.

public final native void wait(long timeout) throws InterruptedException est équivalente à la précédente mais l'attente est bornée par un paramètre de timeout en millisecondes. Avant de pouvoir reprendre son exécution après un timeout il est toujours nécessaire de reprendre le moniteur.

public void notify() envoie une notification à une thread en attente **wait()** sur le même objet.

public void notifyAll() envoie une notification à toutes les threads en attente **wait()** sur le même objet.

Exemple de synchronization

L'exemple suivant montre comment temporiser l'incrémentation, lorsque celle-ci n'est pas possible. Si l'utilisateur appelle **inc()** et que le compteur est déjà au maximum MAX, l'objet appelle tout d'abord **attendIncrementable()** qui ne rend la main que si l'on peut incrémenter. Si cela n'est pas possible, la méthode appelle **wait()** pour mettre "en pause" le thread courant, libérer l'objet et attendre d'être notifié d'un changement: en l'occurence, on attend une décrémentation.

```
public interface Compteur {
public static final long MIN = 0; // minimum
public static final long MAX = 5; // maximum

public long value(); // valeur entre MIN et MAX
public void inc(); // incremente si value() < MAX
public void dec(); // decremente si value() MIN }

public class CompteurConcurrent implements Compteur {
 protected long count = MIN;
 public synchronized long value() {
 return count; }

 public synchronized void inc() {
 attendIncrementable();
 setCount(count + 1); }
```

```
 public synchronized void dec() {
 attendDecrementable();
 setCount(count - 1); }
 protected synchronized void setCount(long newValue) {
 count = newValue;
 notifyAll(); }
 protected synchronized void attendIncrementable() {
 while (count >= MAX)
 try { wait(); } catch(InterruptedException ex) {}; }
 protected synchronized void attendDecrementable() {
 while (count <= MIN)
 try { wait(); } catch(InterruptedException ex) {};
 }
}
```

Les Timers

Les Timers permettent de planifier des tâches régulières. Il s'agit de réveiller des objets implémentant **TimerTask** à intervalles réguliers. On utilise la classe **Timer** pour spécifier les intervalles de valeurs. Toutes les tâches planifiées s'exécutent alors dans le même processus léger.

On peut annuler un timer avec la méthode **cancel()** de **Timer** ou annuler une tâche planifiée en utilisant cette fois celle de **TimerTask**.

```
import java.util.*;
public class DateTask extends TimerTask {
  String msg;
  public DateTask(String msg) { this.msg = msg; }
  public void run() {
    System.out.println(Thread.currentThread().getName() +
                       " " + msg + ": " + new Date());
  }
}

public class TimerExample {
  public static void main(String[] args) {
    Timer timer = new Timer();
    DateTask task0 = new DateTask("task0");
    DateTask task1 = new DateTask("task1");
    DateTask task2 = new DateTask("task2");
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.HOUR_OF_DAY,17);
    cal.set(Calendar.MINUTE,14);
    cal.set(Calendar.SECOND,0);
    Date givenDate = cal.getTime();
    //task0: dès maintenant, toutes les 5 secondes
    timer.schedule(task0, 0, 5000);
    //task1: départ dans 2 secondes, toutes les 3 secondes
    timer.schedule(task1, 2000, 3000);
    //task2: une seule fois à la date fixée
    timer.schedule(task2, givenDate);
    System.out.println(Thread.currentThread().getName()
```

```
                                        + " terminé!");
  }
```

4.5 Collections thread-safe

Les accès concurrents à des collections provoquent souvent la levée de l'exceptions **ConcurrentModificationException**. L'idée principale des collections thread-safe est d'implémenter des classes robustes face à des accès concurrents mais faiblement consistantes au niveau de la cohérence des données. Une facon simple d'implémenter une collection thread-safe est de garantir la synchronisation des opérations, à la main, ou à l'aide de wrappers:

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c)
```

Il ne faut cependant pas oublier de protéger les itérateurs pendant toute la durée de l'itération:

```
Collection c = Collections.synchronizedCollection(myCollection);
   ...
synchronized(c) {
    Iterator i = c.iterator(); // Must be in the synchronized block
    while (i.hasNext())
       foo(i.next());
}
```

Exemple de création d'un wrapper thread-safe

Pour illustrer comment un code peut être rendu thread-safe, on considère la servlet suivante:

```
public class UnsafeGuestbookServlet extends HttpServlet {
    private Set visitorSet = new HashSet();
    protected void doGet(HttpServletRequest httpServletRequest,
                   HttpServletResponse httpServletResponse)
                   throws ServletException, IOException {
        String visitorName = httpServletRequest.getParameter("NAME");
        if (visitorName != null)
            visitorSet.add(visitorName);
    }
}
```

Cette servlet peut être modifiée en remplacant l'attribut privé par:

```
private Set visitorSet = Collections.synchronizedSet(new HashSet());
```

Ces wrappers introduisent cependant des difficultés au niveau des performances d'utilisation des collections. Si l'accès est fréquent ou si l'espace mémoire utilisé est conséquent, l'utilisation de copies temporaires fait augmenter le nombre de traitements et l'espace mémoire occupé.

Wrappers thread-safe et faiblement consistants

L'explication précédente se base sur l'idée que le plus simple est d'imposer l'exclusion mutuelle sur l'accès à une collection; il suffit pour cela d'ajouter un **synchronized** sur les méthodes de classes pour garantir un comportement thread-safe. Ceci étant, il faut considérer les deux problèmes suivants:

- De nombreuses opérations sont composées (exemple: création d'un itérateur, puis appel à **next()** pour le parcourt) et requierent donc des mécanismes de synchronization plus complexes.
- Certains opérations, comme **get()** peuvent supporter la concurrence, si l'on autorise la lecture multiple.

Pour ces raisons, Java 1.5 introduit un certains nombres de wrappers qui sont *thread safe* mais faiblement consistants.

Le premier exemple que l'on peut prendre, commun à toutes les collections, est l'itérateur faiblement consistant (*weakly consitent*). Lors de l'appel à **next()**, si un nouvel élément à été ajouté entre temps, il sera ou ne sera pas retourné par **next()**. De même, si un élément est enlevé entre deux appels à **next()**, il ne sera pas retourné pour les prochains appels (mais il a pu avoir été déjà retourné). Ceci étant, dans tous les cas, l'appel à **next()** ne levera pas l'exceptions **ConcurrentModificationException**.

Listes, Vecteurs, HashMap, Queues

Les mêmes idées d'accès robustes sont implémentées dans les structures de données classiques, au niveau des primitives d'accès. La classe **CopyOnWriteArrayList** permet de créer un *wrapper* d'accès à un liste (**ArrayList**) ou à un vecteur (**Vector**). Ce système de wrapper évite de proposer de nouvelles classes, remplaçant **ArrayList** et **Vector** et met en place le comportement suivant: lorsqu'un accès d'écriture modifie la structure, une copie de la liste ou du vecteur est créée et les itérations de parcours en cours se font sur cette copie. Lorsque toutes les itérations en cours sont terminées, la copie est alors détruite.

De même, un système de wrapper permet de transformer une *HashMap* en **ConcurrentHashMap**. Les opérations multiples sont autorisées, en lecture et en écriture et les itérateurs retournés sont *weakly consistent*. Ce wrapper est diamétralement opposé à **Collections.synchronizedMap** qui est une implémentation ou chaque méthode est synchronizée garantissant l'exclusion mutuelle. Pour les ensembles, un système de wrapper permet de transformer un **HashSet** en set à accès concurrent.

Pour les queues, la classe **ConcurrentLinkedQueue** implémente une version *thread safe* d'une queue FIFO. Java 1.5 introduit des queues bloquantes, ce qui permet par exemple de faire attendre un producteur, lorsqu'un consommateur est trop lent par rapport au producteur. Ces classes implémentent l'interface **BlockingQueue**.

5 Entrées / Sorties

Contributeurs

Jean-François Lalande, François Guerry

Référence importante: JTIO

5.1 IO Simples

Un fichier peut-être lu/écrit en utilisant différentes classes de Java. Cela permet d'accéder aux données du plus bas au plus haut niveau, suivant que le développeur nécessite de comprendre les données.

Au niveau le plus bas, la lecture peut se faire au niveau de l'octet. Dans ce cas, le fichier est vu comme un flux d'octets. Une lecture s'arrête quand l'opération **read()** renvoie -1.

```
FileInputStream in = null;
FileOutputStream out = null;
try { in = new FileInputStream("xanadu.txt");
      out = new FileOutputStream("outagain.txt");
      int c;
      while ((c = in.read()) != -1) {
          out.write(c);
      } catch ...
```

De manière très similaire, un fichier peut-être vu comme un flux de caractère. La différence réside dans la gestion de l'internationalisation: le caractère, stocké en Unicode, est projeté dans le jeu de caractère local.

```
FileReader inputStream = new FileReader("xanadu.txt");
FileWriter outputStream = new FileWriter("characteroutput.txt");
// code identique au précédent pour lecture/écriture
```

Entrées/sorties ligne et bufferisée

Juste après le découpage caractère par caractère vient naturellement l'accès par ligne au contenu d'un fichier. A partir des objets de type **FileReader** qui fournit le flux de caractères, la classe **BufferedReader** agrège le flux en le découpant à chaque *carriage-return* ("r"), ou *line-feed* ("n"). La taille du buffer peut être précisée à la construction.

```
BufferedReader inputStream = null;
PrintWriter outputStream = null;
```

```
try { inputStream = new BufferedReader(new FileReader("xanadu.txt"));
      outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));

      String l;
      while ((l = inputStream.readLine()) != null) {
          outputStream.println(l);
    }
```

Cette implémentation est plus efficace en terme de performance grâce à l'utilisation du buffer. Le nombre d'appels à l'API native de lecture est réduit puisque la lecture s'opère depuis une zone mémoire, le buffer, qui n'est raffraichit que lorsque ce buffer est vide. Lors de l'écriture, le buffer est *flushé* automatiquement et peut être forcé par un appel à **flush()**.

Dans le cas bufferisé ou non, les classes de lecture héritent de **Reader** et implèmentent **Readable**. La contrainte minimum de l'interface est très faible: il faut implémenter **int read(CharBuffer cb)**.

Entrées / Sorties formattées

Les entrées sorties formattées permettent de découper le flux en mots projettés directement dans le type adéquat. Dans un premier temps, le formattage peut-être basé sur la classe **String** avec comme séparateur l'espace. La classe **Scanner** fournit l'implémentation de ce principe en travaillant par exemple sur un flux bufferisé qui se parcourt alors comme un itérateur:

```
Scanner s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
while (s.hasNext()) {
    System.out.println(s.next());
}
```

Si le type du contenu est connu, des primitives permettent de projeter directement le mot dans la classe voulue, sauf usi le prochain mot ne peut être interprété dans le type voulu:

```
s.useLocale(Locale.FRANCE);
while (s.hasNext()) {
    if (s.hasNextDouble()) {
            sum += s.nextDouble();
        } else { s.next(); }}
```

D'autres méthodes de **Scanner** permettent de manipuler le flux formattée:

- **String findInLine(String pattern)**: trouve le pattern spécifié
- **public Scanner skip(Pattern pattern)**: positionne le scanner sur le pattern
- **Scanner reset()**, **Scanner useDelimiter(Pattern pattern)**, ...

Interactions avec les entrées et sorties standards

Les 3 entrées/sorties standards des systèmes POSIX sont accessible depuis la machine virtuelle au travers de la classe **System**:

- **System.in**: entrée standard (implémente **OutputStream**)
- **System.out**: sortie standard (implémente **OutputStream**)

- **System.err**: sortie d'erreur standard (impléente **InputStream**)

On peut y accéder en utilisant les classes **OutputStreamWriter** et **InputStreamReader**.

Une alternative intéressante pour intéragir avec la console est l'utilisation de la classe **Console**. Elle fournit notamment une implémentation plus sûre pour la lecture de mot de passe.

```
Console c = System.console();
if (c == null) {
         System.err.println("No console.");
         System.exit(1); }
char [] p = c.readPassword("Enter your password: ");
```

L'affichage des caractères est désactivé dans la console. De plus, l'implémentation de la récupération du mot de passe comme un tableau permet de désallouer plus rapidemment que s'il s'agissait d'une **String**. Si le programme n'est pas attaché à une console ou qu'il n'est pas en intéractivité avec une console, l'objet renvoyé est un pointeur **null**.

Flux de données: vers la sérialisation

Les types simples supportent une projection directe de leur représentation dans un fichier. Les classes à utiliser sont **DataInputStream** et **DataOutputStream**. La classe String est elle aussi concernée car son implémentation est spéciale. L'exemple suivant montre comment écrire et relire des types simples:

```
static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = { "Java T-shirt", "Java Mug" };

DataOutputStream out = new DataOutputStream(
                new BufferedOutputStream(new FileOutputStream(dataFile)));
for (int i = 0; i < prices.length; i ++) {
  out.writeDouble(prices[i]);
  out.writeInt(units[i]);
  out.writeUTF(descs[i]);
}

DataInputStream  in = new DataInputStream(
                new BufferedInputStream(new FileInputStream(dataFile)));

double price; int unit; String desc; double total = 0.0;

try { while (true) {
          price = in.readDouble();
          unit = in.readInt();
          desc = in.readUTF();
          System.out.format("You ordered %d units of %s at $%.2f%n",
                  unit, desc, price);
          total += unit * price;
      }
} catch (EOFException e) { }
```

5.2 Sérialisation

Les objets peuvent être écrits directement dans des fichiers de donnée: on appelle cela le processus de sérialisation. Un objet est sérialisable s'il implémente l'interface **Serializable**. Aucune méthode n'est à implémenter: il s'agit d'une sorte de flag signalant que l'objet *peut* être sérialisé. Dans l'exemple ci-dessous, une instance de **Maison** peut être sérialisée:

```
 /**
 * Une classe agrégé qui est attribut de PersonneIO.
 */
package io;

import java.io.Serializable;

/**
 * Important: doit lui aussi être Serializable sous peine de:
 * java.io.NotSerializableException: io.Maison
 * lors de la serialisation de PersonneIO.
 */
public class Maison implements Serializable {
  public String adresse;
  public Maison(String adresse) {
    this.adresse = adresse;
}}
```

Sérialisation: dépendances

L'intérêt de la sérialisation est d'automatiser l'écriture des dépendances de classes. L'écriture d'une classe agrégant plusieurs objets provoque l'écriture des objets agrégés. Dans l'exemple suivant, l'instance de **Maison** est écrit automatiquement dans le fichier lorsque l'instance de **PersonneIO** est sérialisée.

```
/** Classe qui va être sérialisée. */
package io;
import java.io.Serializable;

public class PersonneIO implements Serializable {

  private int num_compte_bancaire = 8878;
  public String nom = "none";
  public Maison m;
  public PersonneIO() {
    m = new Maison("Mehun");
  }
  protected void setNom(String n) {
    nom = n;
  }
  public void setAdresse(String a) {
    this.m.adresse = a;
  }
}
```

Sérialisation: écriture

Lors du processus d'écriture, les objets sont sérialisés à l'aide de la classe **ObjectOutputStream**. Un identifiant unique de sérialisation est calculé à la compilation: chaque instance sérialisée poss_de cet identifiant. Cela empêche le chargement d'un objet sérialisé ne correspondant plus à une nouvelle version de la classe correspondante.

```
package io;
import java.io.FileNotFoundException; import java.io.FileOutputStream;
import java.io.IOException; import java.io.ObjectOutputStream;
import java.io.Serializable;
public class PersonneIOMainWrite implements Serializable {
  public static void main(String[] args) {
    PersonneIO p = new PersonneIO();
    p.setNom("JFL");
    PersonneIO p2 = new PersonneIO();
    p2.setAdresse("?");
    FileOutputStream fos;
    try {   fos = new FileOutputStream("file.out");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(p);
      oos.writeObject(p2);
      oos.close(); }
      catch (FileNotFoundException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace();
      }}}
```

Sérialisation: lecture

A la relecture des objets sérialisés, il est impératif de connaitre l'ordre de sérialisation (on peut toutefois s'en sortir en faisant de l'introspection). L'exception particulière qu peut être levée est **ClassNotFoundException** dans le cas ou la JVM ne trouve pas la définition de la classe à charger.

```
package io;
import java.io.FileInputStream; import java.io.FileNotFoundException;
import java.io.IOException; import java.io.ObjectInputStream;
import java.io.Serializable;
public class PersonneIOMainRead implements Serializable {
  public static void main(String[] args) {
try { FileInputStream is = new FileInputStream("file.out");
      ObjectInputStream in = new ObjectInputStream(is);
      PersonneIO p = (PersonneIO)in.readObject();
      PersonneIO p2 = (PersonneIO)in.readObject();
      System.out.println(p.nom + " habite " + p.m.adresse);
      System.out.println(p2.nom + " habite " + p2.m.adresse);
      in.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }}}
```

5.3 Beans

L'écriture sous la forme de Bean s'appuie sur l'introspection dans Java. A partir des méthodes **getX()**, les attributs (privés/protected/public) sont lus et écrits en XML.

```
package io;
import java.io.BufferedOutputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintStream;

public class Ecrire {
      public static void main(String[] args) {
      FileOutputStream file = null;
      try { file = new FileOutputStream("output.txt");
      } catch (FileNotFoundException e) {
              System.err.println("Erreur de sortie sur output.txt");
              e.printStackTrace(); }
      BufferedOutputStream out = new BufferedOutputStream(file);
      PrintStream print = new PrintStream(out);
      print.println("Une ligne !");
      print.println("Une autre !");
      print.close();
}}
```

Relecture d'un Bean

A la relecture, les méthodes **setX()** sont appelés afin de rétablir les attributs de l'objets qui sont relus dans le XML.

```
/** Ecriture d'un objet JFrame sous forme d'un bean.
 *  (Exemple de "Java en concentré", D. Flanagan) */
package io;
import java.beans.XMLDecoder;
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

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

    try {
      BufferedInputStream in = new BufferedInputStream(
                               new FileInputStream("bean.xml"));
      XMLDecoder decoder = new XMLDecoder(in);
      Object b = decoder.readObject();
      javax.swing.JFrame frame = (javax.swing.JFrame)b;
      decoder.close();
    } catch (FileNotFoundException e) { e.printStackTrace(); }
  }}
```

5.4 SAX

Il existe de nombreuses librairies de manipulation de fichiers XML. Les deux grandes familles de parseurs sont les parseurs évenementiels et les parseurs à parcours d'arbres.

Les parseurs évenementiels déclenchent des *callbacks* lorsque les balises sont rencontrées. Le code de ces *callbacks* sont dans un *handler*, c'est à dire une classe à part qui surcharge le *handler* par défaut. Le parseur le plus connu est SAX qui signifie *Simple API for XML*.

Les parseurs à parcours d'arbres sont basés sur un parseur SAX. Ils instancient une représentation objet du document, ce qui est plus long en temps et plus couteux en mémoire. Néanmoins, le parcours d'un tel arbre se révèle très pratique à l'utilisation.

En Java, un parseur SAX s'instancie de la sorte:

```
public class ParserXML {
  public static void main(String[] args)
    throws ParserConfigurationException, SAXException, IOException {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    SAXParser parser = factory.newSAXParser();

    DefaultHandler handler = new XMLHandler(); // mon handler
    parser.parse(new File("io/exemple.xml"),handler);
    }}
```

Handler pour SAX

Un *handler* recoit les évenements liés au *parsing* du document. Il ne s'agit pas réellement d'évenements mais tout simplement du fait que les méthodes du *handler* son appellées par le parser lorsqu'un évenement du type "je rencontre une balise" se produit.

Plusieurs méthodes de **DefaultHandler** peuvent être surchargées, notamment à la rencontre d'une balise, fermeture d'une balise ou lors de la détection d'une erreur. Au moment de la récupération d'une balise, ses attributs et leurs valuers peuvent être récupérés dans un objet de type **Attributes**.

```
package io;
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class XMLHandler extends DefaultHandler {

    public void startDocument() throws SAXException {
      System.out.println("Ca démarre !");
    }
    public void startElement(String uri, String localName,
                             String qName, Attributes attributes)
                             throws SAXException {
      System.out.println("qname = " + qname);
        System.out.println("Attribut: " + arg3.getValue(0));
}}
```

5.5 Java NIO 2

Deux paquets ont été spécialement développés pour intéragir avec le *filesystem*: **java.nio.file** et **java.nio.file.attribute**. Ce sont deux sous packages du package **java.nio**, nio signifiant *New Input/Output*. La classe la plus importante dans ces *packages* est la classe **Path** qui fournit des services de manipulation d'arborescence de répertoires et de fichiers.

Ils sont disponibles à partir de Java 1.7. La difficulté de l'implémentation de ces classes réside dans la portabilité de la machine virtuelle: il faut être capable de gérer des systèmes de fichiers différents sur des *operating systems* différents.

Path

La classe **Path** permet de manipuler un fichier ou une arborescence sous la forme d'un objet. La classe **Paths** fournit une méthode statique pour construire des objets **Path** à partir d'une **String**:

```
Path p1 = Paths.get("/tmp/foo");
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get("file:///Users/joe/FileTest.java");
```

C'est en fait un raccourci vers l'objet **FileSystems**:

```
Path p4 = FileSystems.getDefault().getPath("/home/jf/java.rst");
```

Le constructeur de **Path** même s'il existe, est assez inutile puisqu'il ne raccroche pas l'objet instancié au système de fichier.

L'objet instancié contient ensuite une représentation d'un chemin, de sa racine jusqu'au fichier ou au répertoire finaux. Cette représentation permet alors d'apeller les méthodes suivantes qui retournent toutes des objets de type **Path**:

- La méthode **Path getName(n)** permet de récupérer le n-ème élément du chemin, e.g. "jf" pour p4 avec n=1.
- La méthode **Path getParent()** renvoie le répertoire parent du chemin, e.g. /home pour p4..
- La méthode **Path getRoot()** renvoie la racine du chemin.

Dépendances avec l'OS

Certaines méthodes peuvent dépendre du système d'exploitation et de l'encodage de certaines informations pour le système de fichier. Par exemple, la méthode **isHidden()** se base sur la présence d'un . au début du nom de fichier pour les systèmes GNU/Linux alors que Microsoft Windows stocke cette information dans un fichier.

Dans l'exemple suivant, JTIO montre les résultats obtenus sur **Path** "sally/bar" sous Solaris et Windows:

Method Invoked	Returns in the Solaris OS	Returns in Microsoft Windows
toString	sally/bar	sally\bar
getName	bar	bar
getName(0)	sally	sally
getNameCount	2	2
subpath(0,1)	sally	sally
getParent	sally	sally

getRoot	null	null
isHidden	false	false

Normalisation, conversion, concatenation

Un chemin peut contenir des parties comportant de "." ou des "..", notamment si on les construit dynamiquement par concaténation. On peut donc par exemple se retrouver avec un chemin instancié sous la forme "/home/jf/.", ce qui n'est pas très élégant. La méthode **normalize()** permet de simplifier la représentation du chemin, sans s'occuper de l'existence réelle du fichier dénommé.

Une autre méthode permet de récupérer un chemin absolu: **toAbsolutePath()**. Cette méthode est particulièrement utile lorsque la méthode **getRoot()** renvoie un pointeur null (le chemin est relatif). La méthode **toRealPath(b)** combine les effets précédents: il est simplifié, puis si le chemin est relatif, il est converti, et si b est *true* les liens symboliques sont résolus.

Deux chemins peuvent être concaténés à l'aide de la méthode **resolve()**:

```
Path p1 = Paths.get("/home/joe/foo");          // Solaris
System.out.format("%s%n", p1.resolve("bar"));  // Result is /home/joe/foo/bar
```

A l'inverse, un chemin relatif peut être construit entre 2 chemins:

```
Path p1 = Paths.get("home");
Path p3 = Paths.get("home/sally/bar");
Path p1_to_p3 = p1.relativize(p3);  // Result is sally/bar
Path p3_to_p1 = p3.relativize(p1);  // Result is ../..
```

Parcourt, comparaison

Un chemin se parcourt en utilisant l'interface **Iterable**. L'itérateur parcourt le chemin à partir de la racine. De plus, un chemin implémente l'interface **Comparable** ce qui permet de trier des ensembles de chemins. Souvent, on ne compare par l'égalité mais si un chemin est un sous chemin d'un autre. Deux méthodes sont proposées: **startsWith(beginning)** et **endsWith(ending)**.

```
Path path = ...;
for (Path name: path) {
  System.out.println(name);
}

Path path = ...;
Path otherPath = ...;
Path beginning = Paths.get("/home");
Path ending = Paths.get("foo");

if (path.equals(otherPath)) {
//equality logic here
} else if (path.startsWith(beginning)) {
//path begins with "/home"
} else if (path.endsWith(ending)) {
//path ends with "foo"
}
```

Vérification

La primitive **checkAccess(AccessMode... modes)** permet de vérifier si un certains nombres de droits sont présent sur le fichier dénommé par l'objet de type **Path**:

```
Path file = ...;
try {
  file.checkAccess(READ, EXECUTE);
  ...
} catch (IOException x) {
//Logic for error condition...
return;
}
```

On peut aussi vérifier si deux **Path** dénomment le même fichier sur le système de fichier (incluant la prise en compte des liens symboliques le cas échéant):

```
Path p1 = ...;
Path p2 = ...;
try {
if (p1.isSameFile(p2)) {
//Logic when the paths locate the same file
}} catch (IOException x) {
//Logic for error condition...
}
```

Opérations sur les fichiers

Un fichier se déplace à l'aide de **moveTo(Path target, CopyOption... options)**, la copie est réalisée par un méthode similaire: **copyTo(Path target, CopyOption... options)**, la suppression avec **delete()**:

```
Path path = ...;
Path newPath = ...;
path.moveTo(newPath, REPLACE_EXISTING);
Path newPath2 = ...;
newPath.copyTo(newPath2, REPLACE_EXISTING, COPY_ATTRIBUTES);
newPath2.delete();
```

Une nouvelle primitive permet de créer un fichier: **createFile()**. Pour les entrées/sorties, on utilise les classes vues précédemment à partir de l'**InputStream** récupéré sur le **Path**:

```
Path file = ...;
try {
file.createFile();
in = file.newInputStream();
BufferedReader reader = new BufferedReader(
                        new InputStreamReader(in));
OutputStream out = new BufferedOutputStream(
                    file.newOutputStream(CREATE, APPEND));
```

Attributs d'un fichier

Les attributs d'un fichier peuvent être lus à travers la classe **Attributes** du package **java.nio.file.attribute**. Un appel statique à la méthode **readBasicFileAttributes** ou les méthodes spécifiques **readDosFileAttributes** et **readPosixFileAttributes**. Voici un exemple issu de TOR:

```
import java.nio.file.attribute.*;
public class WindowsAttributePrinter {
public static void main(String args) throws IOException {
  for (String name : args) {
    Path p = Path.get(name);
    DosFileAttributes attrs =
              Attributes.readDosFileAttributes(path, false);
    if (attrs.isArchive()) {
      System.out.println(name + " is backed up.");
    }
    if (attrs.isReadOnly()) {
      System.out.println(name + " is read-only.");
    }
    if (attrs.isHidden()) {
      System.out.println(name + " is hidden.");
    }
    if (attrs.isSystem()) {
      System.out.println(name + " is a system file.");
  }}
```

La classe FileVisitor

L'interface **FileVisitor<T>** permet de parcourir une arborescence de répertoire ainsi que tous les fichiers contenus dans cette arborescence. Pensé comme un parseur SAX, le programmeur doit surcharger les méthodes suivantes qui sont déclenchées au cours de la visite:

- **preVisitDirectory(T)**: appellé avant chaque visite d'un répertoire
- **preVisitDirectoryFailed(T, IOException)**: invoqué en cas de visite impossible
- **postVisitDirectory(T,IOException)**: appellé après chaque visite d'un répertoire
- **visitFile** et **visitFileFailed***: appellé après chaque visite de fichier

```
import static java.nio.file.FileVisitResult.*;
public static class PrintFiles extends SimpleFileVisitor<Path> {
  //Print information about each type of file.
  public FileVisitResult visitFile(Path file, BasicFileAttributes attr) {
      if (attr.isSymbolicLink()) {
          System.out.format("Symbolic link: %s ", file);
      } else if (attr.isRegularFile()) {
          System.out.format("Regular file: %s ", file);
      }
      return CONTINUE;
  }
  public FileVisitResult preVisitDirectoryFailed(Path d, IOException e) {
      System.err.println(e);
      return CONTINUE;
  }}
```

Démarrage de la visite

Le lancement du parcourt de l'arborescence s'effectue à l'appel de **Files.walkFileTree**. Deux signatures différentes sont disponibles:

```
walkFileTree(Path start, FileVisitor<? super Path> visitor)
walkFileTree(Path start, Set<FileVisitOption> options, int maxDepth,
             FileVisitor<? super Path> visitor)
```

Dans les deux cas, le chemin de départ est donné ainsi que l'objet dérivant de **FileVisitor**. Dans la deuxième signature des options parmi **FOLLOW_LINKS** et **DETECT_CYCLES** et la profondeur maximum peuvent être précisés. La valeur retournée est particulièrement importante: elle est choisie parmi **CONTINUE**, **TERMINATE**, **SKIP_SUBTREE** , **SKIP_SIBLINGS** (abandon du répertoire courant et de ses frères).

```
Path startingDir = ...;
PrintFiles pf = new PrintFiles();
Files.walkFileTree(startingDir, pf);
EnumSet<FileVisitOption> opts = EnumSet.of(FOLLOW_LINKS);
Finder finder = new Finder(pattern);
Files.walkFileTree(startingDir, opts, Integer.MAX_VALUE, finder);
```

Le comportement du **FileVisitor** n'est pas spécifié. Dans la machine virtuelle de Sun, le parcourt est en profondeur. Il est donc particulièrement important d'effectuer la bonne action dans la bonne méthode, par exemple si l'on encode une suppression récursive de répertoires.

Surveiller un répertoire

Surveiller un répertoire permet de réagir à la modification du système de fichier. Un service de surveillance **WatchService** doit être créé et associé à un **Path**:

```
WatchService watcher = FileSystems.getDefault().newWatchService();
Path dir = ...;
try {
  WatchKey key = dir.register(watcher, ENTRY_CREATE, ENTRY_DELETE,
                              ENTRY_MODIFY);
} catch (IOException x) {
  System.err.println(x);
}
```

A partir de cette clef, les évenements sont récupérés à l'aide de la méthode **pollEvents()**. Ils peuvent ensuite être filtrés par types **ENTRY_CREATE**, **ENTRY_DELETE**, **ENTRY_MODIFY**, **OVERFLOW**.

```
for (WatchEvent<?> event: key.pollEvents()) {
  WatchEvent.Kind<?> kind = event.kind();
  if (kind == ENTRY_CREATE) {
    WatchEvent<Path> ev = (WatchEvent<Path>)event;
    Path filename = ev.context();
  }
}
```

6 Introspection

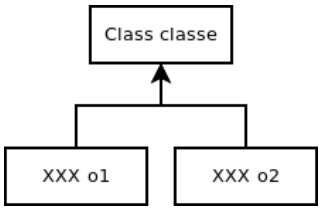
Contributeurs

Jean-Francois Lalande

L'introspection (ou aussi traduit reflexion pour *reflective*) est une capacité du langage Java à permettre l'accès à l'information sur les classes chargées, leurs attributs, méthodes ou constructeurs. Cette fonctionnalité du langage est très utile pour un certain nombre de programme qui analysent dynamiquement des classes du programme, comme par exemple les débuggueurs, les inspecteurs d'objets, les services de sérialisation du type **Serializable** ou **Bean**, les IDEs.

6.1 La classe Class

Chaque objet *o* instancié possède une référence vers un autre objet *def* de type Class. Il s'agit d'un objet contenant un certain nombre d'informations à propos de la classe de l'objet *o*. Evidemment, cet objet *def* est statique puisque toutes les instances de classe comme *o* partagent la même définition de classe. Visuellement, on peut représenter cela ainsi:



La récupération de l'objet **Class** se fait directement sur l'objet ou en appelant la méthode statique de la classe concernée:

```
Class classe = o.getClass();
Class classe = Class.forName("java.lang.String");
```

La classe **Class** est en fait un type paramétré: **Class Class<T>**. Lors de l'introspection d'une classe, le programmeur ne sait pas forcément quel est le type de l'objet *o* (sinon à quoi sert de fait de l'introspection ?). L'écriture précédente signifie donc en pratique:

```
Class<?> classe = o.getClass();
```

6.2 Inspecter une classe

Un certain nombre de méthodes sont disponibles sur la classe **Class** afin d'inspecter les différentes méthodes, attributs, constructeurs, localisation, classes mères, classes agrégées de la classe.

D'autres classes du package **java.lang** permettent de récupérer les informations de la classe **Class**:

- **Constructor<T>**: la classe représentant un constructeur
- **Field**: la classe représentatant un attribut de classe
- **Method**: la classe représentant une méthode de classe
- **Type**: la classe représentant les interfaces implémentées par une classe

- **Package**: la classe représentant le package d'une classe
- **Annotation**: la classe représentant les annotation de la classe

Le but de l'inspection (ou introspection) est de répondre à des questions du type:

- Est-ce que cette classe est publique ?
- Quels sont les constructeurs disponibles pour cette classe ?
- Cette classe possède-t-elle un attribut nommé **truc** ?
- Y a-t-il une méthode **X(String t)** dans cette classe ?

Le but secondaire de l'inspection est de réaliser des actions dynamiquement, en fonction des informations collectées précédemment:

- Construction d'un nouvel objet
- Appel de la méthode **X(String t)**
- Affichage de l'attribut **truc**

Modificateurs de classe

Les méthodes suivantes renseignent sur la définition de la classe [JDOC]:

isAnonymousClass() Retourne vrai si et seulement si la classe est une classe anonyme.

isArray() Détermine si l'objet représente un tableau.

isAssignableFrom(Class<?> cls) Détermine si la classe ou l'interface représenté par ces objet est une super classe ou super interface de la classe cls passée en paramètre.

isEnum() Retourne vrai si la classe est une énumération.

isInstance(Object obj) Détermine si l'objet spécifié en paramètre est compatible avec la classe interrogée pour un possible assignement.

isInterface() Détermine si l'objet de type Classe est une interface.

isPrimitive() Détermine si l'objet de type Classe est un type primitif.

Interfaces, classe mère et attributs

Interfaces et classe mère

La recherche d'interfaces et d'une classe mère s'opère au travers des méthodes:

```
Class<? super T>     getSuperclass()
Class<?>[]   getInterfaces()
```

La classe mère peut être une classe, une interface, un type primitif ou void. Si la classe sur laquelle la méthode est appellée est Object ou une interface, un type primitif ou void, alors null est renvoyé. Les interfaces implémentées peuvent être multiple et sont renvoyées dans l'ordre de leur déclaration par le mot clef *implements*. Si aucune interface n'est implémentée, le tableau est de taille 0.

Attributs

Les attributs de classes peuvent être récupérés à l'aide de **Field[] getFields()**. Cet attribut peut être modifié, par exemple à l'aide de **setInt**, **setFloat**, ... ou tout simplement la méthode **set(Object obj, Object value)** qui modifie l'attribut représenté par l'objet de type **Field** sur l'objet obj avec l'objet value.

```
Classe c = vehicule.getField("moteur");
Moteur m = new Moteur();
c.set(vehicule, m);
```

Les constructeurs

L'ensemble des constructeurs sont renvoyés par la méthode suivante:

```
public Constructor<?>[] getConstructors() throws SecurityException
```

L'objet de type **Constructor** ainsi récupéré peut-être à son tour inspecté. Par exemple, ses paramètres d'appel peuvent être récupérés:

```
Class<?>[]   getParameterTypes()
```

Un constructeur peut aussi être invoqué, afin de construire une nouvelle instance de l'objet que décrit la classe **Class**. On utilise pour cela la méthode **T newInstance(Object... initargs)** qui demande à la classe **Class** une nouvelle instance de classe, avec, comme paramètres (à nombre variables) les paramètres passés à la méthode.

La difficulté de l'utilisation de **newInstance** réside dans la découverte des paramètres du constructeur. En effet, il n'y a pas toujours de constructeurs sans paramètres permetant d'appeler **newInstance()**.

```
String str = new String("test");
Class c = str.getClass();
String str2 = c.newInstance();
```

Les méthodes

Les méthodes peuvent être récupérées dans un tableau d'objets **Method**. Il est aussi possible de chercher une méthode particulière à partir de son nom et de la liste des classes de ses paramètres (le nom n'est en effet pas suffisant):

```
Method[]     getDeclaredMethods()
Method       getDeclaredMethod(String name, Class<?>... parameterTypes)
```

Une méthode peut ensuite être inspectée par exemple pour connaitre son type de retour à l'aide de **Class<?> getReturnType()**, ses paramètres à l'aide de **Class<?>[] getParameterTypes()**, les exceptions qu'elle est susceptible de lever à l'aide de **Class<?>[] getExceptionTypes()**.

Une méthode peut ensuite être invoquée sur l'objet passé en premier paramètre à l'aide des paramètres (à nombre variable) dans les paramètres suivants:

```
Object invoke(Object obj, Object... args)
```

7 Divers

Contributeurs

Jean-Francois Lalande, Frédéric Tronel

7.1 Variance et héritage

VW La notion de variance intervient lorsqu'on s'intéresse au sous typage de données complexe, typiquement des classes. Le langage de programmation doit décider de règles de typage en cas d'héritage et pour les classes génériques. On distingue:

- la covariance: préservation du type du plus spécifique au plus générique
- la contravariance: du type le plus générique au plus spécifique
- la bivariance: si les deux sont possibles
- l'invariance: si aucun ne l'est

Pour éviter le problème de variance lors de l'héritage, une solution simple consiste à garder le typage originel, par exemple pour une signature de méthode:

```
public class Livre {
      Texte f(Texte x) {
              System.out.println("Je retourne un Texte d'un Livre.");
              return x;
      }}
```

```
public class LivreRouge extends Livre {
      // Invariant:
      @Override
      Texte f(Texte x) {
              System.out.println("Je retourne un Texte Rouge.");
              return x;
}}
```

Héritage et cast

Lorsqu'on appelle une méthode surchargée, Java va toujours choisir la plus spécialisée, même si l'on *cast* un objet dans son super type. L'exemple suivant illustre cela:

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

```
                LivreRouge lr = new LivreRouge();
                Texte texte = new Texte();
                Texte t = lr.f(texte);
                Livre l2 = (Livre)lr;
                Texte t2 = l2.f(texte);
        }}
```

Donne, à l'exécution:

```
Je retourne un Texte Rouge.
Je retourne un Texte Rouge.
```

Personne ne peut appeler la méthode de la super classe, sauf depuis l'intérieur de la classe, à l'aide de **super()**. Pour permettre de l'appeler tout de même, on peut bricoler une méthode supplémentaire, mais cela va à l'encontre des notions d'encapsulation et d'héritage:

```
Texte originalF(Texte x) {
                return super.f(x);
        }
```

Type de retour covariant

Le type de retour d'une fonction est covariant: on peut redéfinir une méthode en remplaçant le type de retour par un type plus spécialisé que dans la classe parente. On peut donc par exemple spécifier une méthode **TexteVert f(Texte x)**:

```
public class LivreVert extends Livre {
        // Type de retour covariant
        @Override
        TexteVert f(Texte x)
        {
                System.out.println("Je retourne un Texte Vert.");
                return new TexteVert();
        }}
```

Qui donne, a l'exécution de:

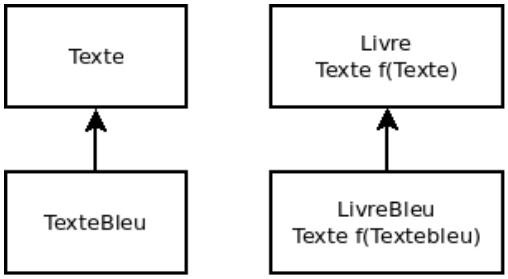
```
LivreVert lv = new LivreVert();
Texte texte = new Texte();
Texte t = lv.f(texte);
Livre l2 = (Livre)lv;
Texte t2 = l2.f(texte);
```

```
Je retourne un Texte Vert.
Je retourne un Texte Vert.
```

On remarquera qu'ainsi, l'appel à la fonction *f* sur l2 permet de récupérer un objet de type **Texte**, sans vraiment savoir quel est le type réel. La covariance du type de retour est donc "sure", d'un point de vue typage.

Paramètres et héritage

Lorsqu'on construit un modèle objet s'appuyant sur l'héritage, il serait assez naturel de faire un raisonnement similaire pour les paramètres de méthodes qui sont redéfinies. Ainsi, par exemple, si **TexteBleu** hérite de **Texte** on peut penser que **LivreBleu** pourrait avoir une méthode utilisant un paramètre covariant, i.e.:

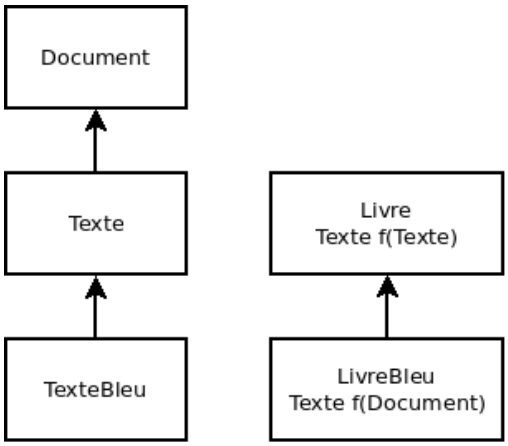


Si Java acceptait une telle redéfinition, le typage ne serait pas sûr car la méthode spécialisée **f** ne saurait que faire d'un appel avec un objet **TexteRouge** héritant de **Texte**:

```
LivreBleu lb = new LivreBleu();
TexteRouge texteRouge = new TexteRouge();
Texte t = lb.f(texteRouge);
```

Paramètre contravariant

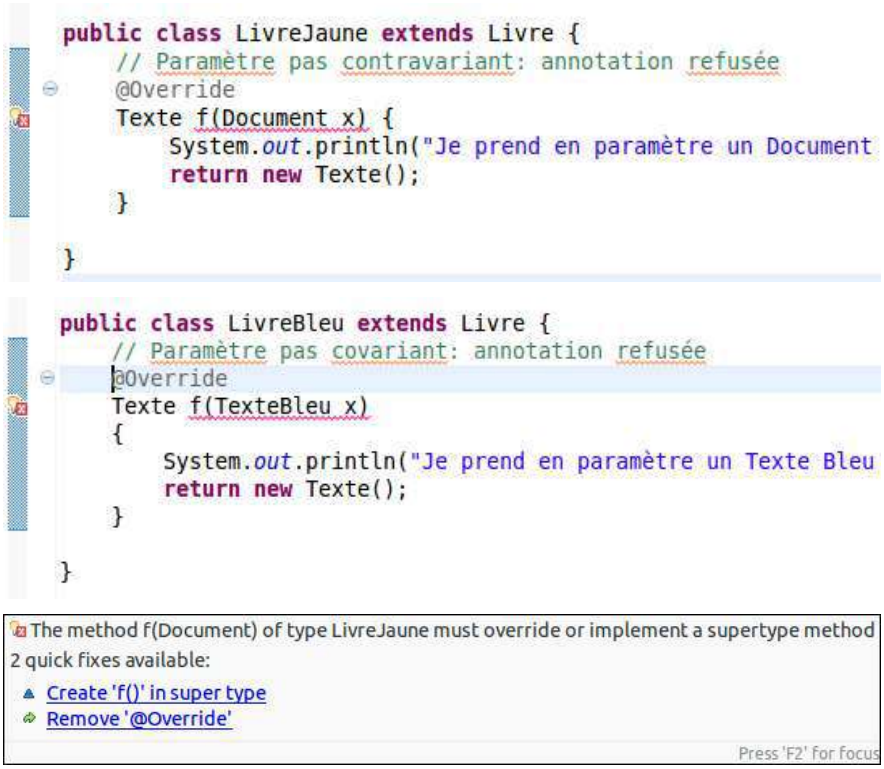
Contrairement à l'intuition qui pousserait à vouloir considérer un paramètre comme covariant, il est plus sûr d'astreindre un paramètre à la contravariance. Ainsi, une méthode redéfinie doit recevoir en paramètre un type plus général que sa classe parente. Si l'on suppose que **Texte** hérite de **Document**, on pourrait donc vouloir faire:



Configuration qui n'est en général pas vraiment ce qui est recherché par un développeur. De plus, l'ajout de la contravariance au langage pose de sérieux problèmes quand on le combine à l'*overloading* [SOC]. Du coup, le langage Java n'utilise ni la covariance ni la contravariance pour un paramètre: il doit être invariant pour surcharger la méthode.

Paramètre contravariant et @Override

Et effectivement, l'utilisation de l'annotation **@Override** combinée à un paramètre contravariant, et à fortiori covariant, ne plait pas du tout au compilateur:



Overloading (surcharge)

Sans avoir utilisé l'annotation **@Override** on peut croire, à tort, avoir réussi à redéfinir une méthode alors qu'en réalité, on l'a surchargée. Suivant le paramètre d'appel, la machine virtuelle choisit l'une ou l'autre des méthodes. Par exemple:

```
public class LivreJaune extends Livre {
// Paramètre pas contravariant: annotation refusée
//@Override:
Texte f(Document x)
{
        System.out.println("Je prend en paramètre un Document et retourne un Texte.");
        return new Texte();
}}
```

surcharge la méthode **f** du **LivreJaune**. Le Main suivant le démontre:

```
LivreJaune lj = new LivreJaune();
Texte texte = new Texte();
Texte t = lj.f(texte);
Document d = new Document();
Texte t2 = lj.f(d);
```

```
Je retourne un Texte d'un Livre.
Je prend en paramètre un Document et retourne un Texte.
```

Le même comportement a lieu pour un paramètre covariant. Le choix de java est l'invariance pour les paramètres d'appel au profit de l'utilisation libre de l'*overloading*.

Variance avec un générique

La gestion des génériques permet de contourner le problème de la variance de paramètres. Bien que non totalement équivalent, l'exemple suivant montre comment spécialiser une méthode d'une sous classe générique.

```
public class LivreToken<T extends Texte> {
Texte f(T x) {
        System.out.println("Je retourne un Texte d'un Livre.");
        return x;
}}
public class LivreMauve extends LivreToken<TexteMauve> {
Texte f(TexteMauve x) {
        System.out.println("Je retourne un Texte Mauve d'un Livre.");
        return x;
}}
```

On peut alors faire:

```
LivreMauve lm = new LivreMauve();
TexteMauve texteMauve = new TexteMauve();
Texte t2 = lm.f(texteMauve);
```

mais pas:

```
LivreMauve lm = new LivreMauve();
Texte texte = new Texte();
Texte t = lm.f(texte);
// MainMauve.java: error: no suitable method found for lm.f(Texte)
```

7.2 Comparable

Java utilise l'interface **Comparable** pour permettre de comparer des objets et grâce à cette comparaison, d'appliquer des algorithmes de tri, notamment dans les collections triées. L'interace **Comparable** ne possède qu'une seule méthode:

```
public interface Comparable<T> {
  int compareTo(T o)
}
```

Mais de nombreuses classes (**Integer**, **Character**, **Calendar**, **URI**, etc.) l'implémentent. La méthode **compareTo** doit donner l'implémentation de la comparaison (anticommutative, transitive, symmétrique pour les exceptions) de deux objets instances d'une même classe:

```
public class LivreComparable implements Comparable<LivreComparable> {
  @Override
  String auteur = "";
  String titre = "";
  public int compareTo(LivreComparable o) {
    System.out.println("Je me compare à un Livre Comparable");
    if (! auteur.equals(o.auteur))
      return auteur.compareTo(o.auteur);
    return titre.compareTo(o.titre);
}}
```

Tris et Comparable

Il devient alors très facile de trier une collection:

```
LivreComparable l2 = new LivreComparable();
l2.auteur = "Stroustrup B."; l2.titre = "Le langage C++";
LivreComparable l3 = new LivreComparable();
l3.auteur = "Puybaret E."; l3.titre = "Les Cahiers du Programmeur Java";
LivreComparable l4 = new LivreComparable();
l4.auteur = "Lalande J.-F."; l4.titre = "POO avancée";
ArrayList<LivreComparable> a = new ArrayList<LivreComparable>();
a.add(l2); a.add(l3); a.add(l4);
System.out.println(a);
Collections.sort(a);
System.out.println(a);
```

```
[Stroustrup B. - Le ..., Puybaret E. - Les..., Lalande J.-F. - POO...]
Je me compare à un Livre Comparable
Je me compare à un Livre Comparable
Je me compare à un Livre Comparable
[Lalande J.-F. - POO..., Puybaret E. - Les..., Stroustrup B. - Le ...]
```

Héritage et Comparable

Lorsqu'une classe hérite d'une classe implémentant l'interface **Comparable**, les paramètres de **compareTo** restent invariants. Il faut donc pouvoir se comparer au type du parent. Bien souvent, l'appel à **super** suffit (en fait, on ne réimplémente même pas la méthode puisqu'on l'a par héritage):

```
public class LivreRougeComparable extends LivreComparable {
      @Override
      public int compareTo(LivreComparable o) {
              return super.compareTo(o);
      }}
```

On peut tout de même s'amuser à spécialiser la méthode, par exemple:

```
@Override
public int compareTo(LivreComparable o) {
        if (o instanceof LivreRougeComparable)
```

```
                return super.compareTo(o);
        else
                return 1;
}
```

7.3 Autoboxing - Unboxing

L'*autoboxing* désigne la capacité du compilateur à ajouter au code source ce qui manque pour manipuler un type simple sous forme d'une référence. Cette capacité n'existe que depuis la version 5 de Java:

```
Integer x = new Integer(5);
Integer y = 4;
// ou bien même:
Integer z = x + y - 3 * x;
```

L'autoboxing est possible à d'autres endroits qu'une initialisation d'une référence, par exemple lors d'appels à une méthode d'ajout d'une collection:

```
Vector<Integer> v = new Vector<Integer>();
v.add(y);
v.add(8);
```

Inversement, l'*unboxing* permet de repasser au type primitif:

```
Integer x = new Integer(5);
int u = x;
```

Attention à ne pas *autoboxer* dans un **Object**, car le compilateur refusera d'*unboxer*:

```
Object o = 4;
System.out.println("o=" + o); // ok, imprime 4.
int o2 = o; // error: incompatible types, required: int, found: Object
```

7.4 JNI

Java offre la possibilité de faire des appels natifs. Du côté du code java, on charge la librairie de manière statique et on déclare une méthode native:

```
public class TestNatif {
      static { System.loadLibrary("jfllib"); }

      private native void test(String name);

      public static void main(String[] args) {
              TestNatif t = new TestNatif();
              t.test("JFL");
      }}
```

On génère le .h correspondant:

```
javah TestNatif
```

Ce qui produit le fichier d'entête TestNatif.h contenant:

```
JNIEXPORT void JNICALL Java_TestNatif_test
(JNIEnv *, jobject, jstring);
```

Du côté du C

On doit ensuite implémenter le code C de l'appel natif dans TestNatif.c:

```
#include <jni.h>
#include <stdio.h>
#include <TestNatif.h>

JNIEXPORT void JNICALL Java_TestNatif_test (JNIEnv * env , jobject o, jstring s)
{
  const char *name= (*env)->GetStringUTFChars(env,s,0);
  printf("String %s\n", name);
}
```

Puis compiler la librairie:

```
gcc -shared -o libjfllib.so TestNatif.c  -I/usr/lib/jvm/java-7-oracle/include -I/usr/lib
```

Et on peut enfin exécuter le code java:

```
java TestNatif
String JFL
```

8 Code sample License

9 Bibliographie

BB(1, 2)	Bytecode basics, A first look at the bytecodes of the Java virtual machine, Bill Venners, JavaWorld.com, 09/01/96.
JVM	Java Virtual Machine, Wikipedia, the free encyclopedia.
JS	JAR File Specification, Sun Microsystems, Inc., 1999.
JVMS	The JavaTM Virtual Machine Specification, Second Edition, Tim Lindholm, Frank Yellin, Sun Microsystems, Inc., 1999.
GB	Java's garbage-collected heap, An introduction to the garbage-collected heap of the Java virtual machine, Bill Venners, JavaWorld.com, 1996.
GCH	Java theory and practice: Garabage collection in the Hotspot JVM, Brian Goetz, 25 Nov 2003.
JTIO	Lesson: Basic I/O, The Java Tutorials, Oracle Corporation.
TOR	The Open Road: java.nio.file, Elliotte Rusty Harold, Java.net, 3 Juillet 2008.
JDOC	Java™ Platform, Standard Edition 6, API Specification, Sun Microsystems.
CL	Java ClassLoader - Write your own ClassLoader, Miron Sadziak, 15/05/10.
VW	Covariance and contravariance (computer science), 24/5/2014.
SOC	Why is there no parameter contra-variance for overriding?, 24/5/2014.