

Java, programmation avancée

**Eléments avancés du langage
Programmation concurrente
Communications distantes
Administration et sécurité**

Leuville Objects
3 rue de la Porte de Buc
F-78000 Versailles
FRANCE

tel : + 33 (0)1 39 50 2000
fax: + 33 (0)1 39 50 2015

www.leuville.com
contact@leuville.com

© Leuville Objects, 2000-2014
29 rue Georges Clémenceau
F-91310 Leuville sur Orge
FRANCE

<http://www.leuville.com>

Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Les marques citées sont des marques commerciales déposées par leurs propriétaires respectifs.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A RÉPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

Table des matières

Aperçu des nouveautés introduites dans Java 5.....	1
Java 5	1-3
Les types génériques	1-5
La nouvelle boucle for	1-9
Autoboxing et Unboxing	1-13
Les énumérations	1-17
Les imports statiques	1-21
Les varargs	1-25
Les annotations	1-29
Java 5 : les nouveautés des packages java.lang et java.util.....	33
StringBuilder	2-35
Lancement de processus avec ProcessBuilder	2-39
Le formatage de texte	2-43
La classe Console.....	2-47
La classe Scanner.....	2-51
API de concurrence.....	2-57
La boucle for each	61
La boucle for each.....	3-63
L'interface Iterable	3-69
Les énumérations en Java.....	81
Rôle des énumérations	4-83
Structure d'une énumération.....	4-85
Utilisation d'une énumération	4-87
Utilisation de méthodes	4-89
La redéfinition et la covariance des méthodes en Java.....	91
Redéfinition	5-93
Covariance du type de retour	5-95
Utilisation d'une redéfinition.....	5-97
Les classes imbriquées et anonymes en Java	99
Classe imbriquée.....	6-101
La classe imbriquée renforce l'encapsulation.....	6-103
Classe imbriquée et visibilité.....	6-105
Classe anonyme	6-109
La généricité en Java.....	113
Présentation de la généricité	7-115
Utilisation de la généricité	7-117
Définition d'une classe générique simple	7-119
Un exemple complet	7-121
Méthodes génériques	7-123



Généricité avancée en Java.....	125
Restrictions et limites.....	8-127
Règle d'héritage pour les types génériques	8-131
Les paramètres de type joker	8-133
Mécanisme d'effacement de type	8-137
Interopérabilité entre types génériques et non génériques.....	8-141
Les collections en Java.....	143
Qu'est-ce qu'une collection	9-145
Les interfaces de Collection.....	9-147
Les différentes classes de collection	9-149
L'interface Iterator	9-153
L'interface List	9-155
L'interface Set.....	9-157
L'interface Queue	9-159
L'interface Map	9-161
Critères de choix d'une implémentation	9-165
Les structures de hashage.....	167
Utilité du hashCode	10-169
La méthode equals	10-179
Redéfinir la méthode equals	10-181
Cohérence equals / hashCode	10-187
Collections triées.....	189
Collections triées.....	11-191
Ordre naturel	11-199
L'interface Comparator.....	11-205
Tri de listes quelconques.....	11-209
Nouveautés sur les collections avec Java 5 et Java 6.....	211
L'interface Queue	12-213
L'interface ConcurrentMap	12-219
L'interface NavigableMap.....	12-225
Les annotations en Java.....	227
Qu'est-ce qu'une annotation ?	13-229
Annotations standard	13-231
Les méta-annotations	13-239
Utilisation avancée des annotations.....	245
Création d'annotations	14-247
Réflexivité des annotations	14-253
L'outil Annotation Processing Tool (apt).....	14-257
Aperçu des nouveautés introduites dans Java 6.....	259
Java 6	15-261
Nouvelles annotations	15-263



Classpath.....	15-271
Remote Method Invocation, les bases.....	273
Présentation.....	16-275
Architecture RMI.....	16-277
Etapas de réalisation d'une application RMI.....	16-279
Déroulement sur un exemple.....	16-281
Modélisation du problème.....	16-283
Définition des interfaces des services.....	16-285
Typage des valeurs de retour et paramètres.....	16-287
Contrôle de la serialization.....	16-289
Interface Gestionnaire.....	16-291
Implémentation des interfaces Remote.....	16-293
Implémentation de l'interface Gestionnaire.....	16-297
Réalisation du serveur de Banque.....	16-299
Le serveur de noms.....	16-301
Réalisation de l'application cliente.....	16-303
Client de type applet.....	16-305
Compilation des différents fichiers.....	16-307
Types d'installations RMI et procédures de lancement.....	16-309
Echanges entre processus et rmiregistry.....	16-311
Client RMI de type applet.....	16-313
Client RMI de type application autonome non téléchargée.....	16-315
Remote Method Invocation: aspects avancés.....	317
Architecture RMI.....	17-319
Types d'installations RMI et procédures de lancement.....	17-321
Echanges entre processus et rmiregistry.....	17-323
Client RMI de type application autonome téléchargeable.....	17-325
Sockets spécifiques.....	17-329
Réalisation de callbacks RMI.....	17-331
Politique d'activation des objets serveurs.....	17-335
Activation: adaptation de la classe GestionnaireImpl.....	17-337
Classe de paramétrage de l'activation (setup).....	17-339
Lancement côté serveur.....	17-345
JMS (Java Messaging Service).....	347
Service de messagerie.....	18-349
Principe et acteurs.....	18-351
Domaines de messagerie.....	18-353
Description de l'API.....	18-357
Connexions JMS.....	18-359
Sessions JMS.....	18-361
Messages JMS.....	18-363
Selecteurs de messages.....	18-367
Exemples de programmation avec JMS.....	18-369
Création d'applications JMS robustes.....	18-375



JMS 1.1	18-377
Programmation réseau à base de Sockets.....	379
Socket réseau	19-381
Sockets en mode connecté	19-383
ServerSocket	19-385
Socket.....	19-387
Sockets en mode non connecté	19-389
Sockets en mode non connecté multicast	19-413
Entrées-sorties de type NIO	427
Caractéristiques.....	20-429
Exemple: fichier.....	20-431
Buffer	20-433
Buffer.flip().....	20-435
Buffer.clear().....	20-437
Opérations de lecture	20-439
Opérations d'écriture	20-441
Allocation d'un buffer	20-443
Mapper un fichier en mémoire.....	20-445
Gestion des verrous.....	20-447
Utiliser plusieurs buffers simultanément	20-449
Encodage / décodage octet-unicode.....	20-451
Exemple octets-unicode	20-453
Entrées-sorties de type NIO appliquées au réseau.....	455
E/S réseau asynchrones.....	21-457
ServerSocketChannel.....	21-459
SocketChannel	21-461
Threads et MultiThreading.....	465
Programmation parallèle.....	22-467
Thread	22-471
Utilité du multithreading.....	22-473
Classe Thread.....	22-477
Solution 1	22-481
Solution 2	22-487
Choix d'implémentation d'un thread.....	22-493
Les états d'un thread	22-497
Priorités	22-499
Partage de la ressource CPU	22-501
Démons	22-503
Groupes de threads.....	22-505
Executor	22-507
Pool de threads.....	22-511
Synchronisation de threads	515



Synchronisation	23-517
Rendez-vous.....	23-525
Interface Lock	23-537
Exemple d'utilisation d'un Lock	23-539
Les sémaphores	23-541
Exemple d'utilisation d'un sémaphore	23-543
Les collections synchronisées	23-545
Les files d'attentes	23-549
Mécanismes avancés des threads Java	553
Threads et variables	24-555
Introduction à JMX.....	559
Java Management Extensions	25-561
Architecture	25-567
Le niveau instrumentation	25-569
Agents JMX	25-571
Adapteurs et connecteurs	25-575
Monitoring d'une JVM avec JMX	25-577
L'essentiel de JMX.....	585
Management Beans	26-587
Standard MBeans	26-589
Enregistrement d'un MBean	26-595
Envoi de notifications	26-601
Dynamic MBean	26-607
Model MBean	26-623
Connecteurs JMX.....	635
Lancement d'un connecteur RMI	27-637
Utiliser JConsole via le connecteur RMI	27-643
Accès au serveur JMX via le connecteur RMI	27-647
Sécurité et permissions en Java.....	651
Le questionnaire de sécurité en Java.....	28-653
La sécurité de la plate-forme Java 2	28-655
Les fichiers de règles de sécurité	28-661
L'outil PolicyTool.....	28-673
JAAS : Java Authentication and Authorization System.....	675
Objectifs de JAAS	29-677
Possibilités de JAAS	29-679
Terminologie JAAS	29-681
Vue d'ensemble	29-683
A quoi ressemble une application qui utilise JAAS	29-685
LoginContext	29-695
Les modules d'identification ou LoginModule	29-697
Les méthodes doAs() et doAsPrivileged().....	29-701



Subject et Principal	29-703
La gestion des permissions avec JAAS	29-705
AccessControlContext	29-709
La classe Policy	29-711
Utiliser un fichier de règles spécifique	29-713
Structure d'un fichier de règles	29-715
Attribuer des permissions à des utilisateurs identifiés	29-719
Les différents types de Principal	29-721
Les callbacks JAAS	29-723
javax.security.auth.callback	29-725
Un CallbackHandler	29-729
Ecrire un module d'identification	29-731
Le ClassLoader Java.....	739
Le chargeur de classes (classloader)	30-741
Mécanisme de chargement des classes	30-743
L'absence de classe comme atout	30-747
L'utilisation de chargeurs de classe comme espace de nom	30-751
Création d'un classloader	30-753
Références	757
Concepts Objet	31-759
UML	31-761
Langages	31-763
Architectures réparties	31-765
Objets métier et composants	31-767
Web Services	31-769
SOA	31-771
Bibliographie	31-773
Objet, UML, design patterns	31-775
CORBA	31-779
Langages	31-781
Objets métier et composants	31-787
Web Services	31-789
SOA	31-791
Embarqué	31-793
Annexes	795
L'API d'introspection.....	797
Utilité de l'introspection	32-799
Les inconvénients de l'introspection	32-801
Manipulation des classes	32-803
Manipulation des champs	32-823
Manipulation des méthodes	32-829
Manipulation des constructeurs	32-835
La classe Array	32-841

Java, Programmation avancée

Aperçu des nouveautés introduites dans Java 5

Version 2.2

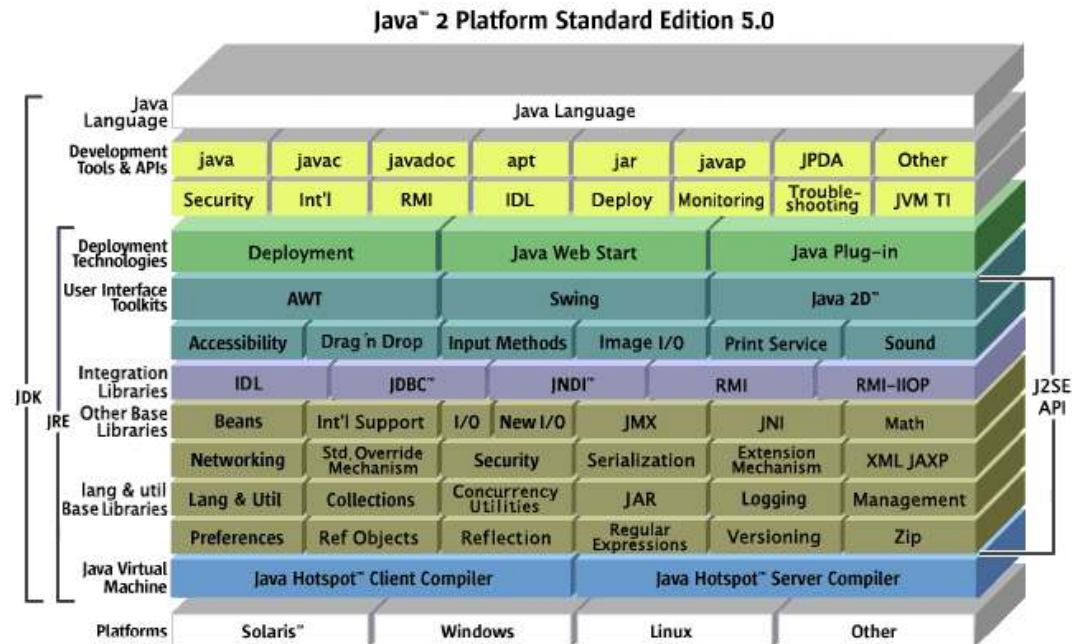
- Généricité
- Boucle for each
- Autoboxing, Unboxing
- Les énumérations
- Les imports statiques
- Les varargs
- Les annotations

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Java 5

- Version du langage Java sortie en 2004
- Version majeure du langage Java car introduit de nouveaux concepts et de nouvelles syntaxes dans le langage.



- Liste de nouveautés : <http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html>

Java 5

Notes

Les types génériques

Principe

- Permet de paramétrer certains types Java avec d'autres types.
- Evite de manipuler des instances de Object.
- Evite des casts inutiles.
- Rajoute des contrôles de types lors de la compilation.

Usages classiques

- Typer une collection.
- API de reflection.
- ...

Les types génériques

Notes

Les types génériques

Exemple

Sans généricité

```
List valeurs = new ArrayList();
valeurs.add("foo");
valeurs.add("bar");
valeurs.add(new Integer(10)); // Autorisé

for (Iterator it = valeurs.iterator(); it.hasNext();)
{
    Object val = it.next();
    if (val instanceof String) {
        System.out.println(val);
    }
}
```

Avec généricité

```
List<String> valeurs = new ArrayList<String>();
valeurs.add("foo");
valeurs.add("bar");
//valeurs.add(new Integer(10)); <-- Interdit

for (Iterator<String> it = valeurs.iterator();
it.hasNext();) {
    String val = it.next();
    System.out.println(val);
}
```


Les types génériques

Notes

La nouvelle boucle for

Principe

- Unifier le processus d'itération à travers diverses structures de données.

Usages classiques

- Parcourir une collection (liste ou map).
- Parcourir un tableau.
- Parcourir toute structure de données maison.

La nouvelle boucle for

Notes

La nouvelle boucle for

Exemple

```
List<String> valeurs = new ArrayList<String>();  
valeurs.add("foo");  
valeurs.add("bar");  
  
for (String val : valeurs) {  
    System.out.println(val);  
}
```

La nouvelle boucle for

Notes

Autoboxing et Unboxing

Principe

- Autoboxing : correspondance automatique d'un type primitif vers son wrapper.
- Unboxing : correspondance automatique d'un wrapper vers son type primitif.

Usages classiques

- Ajouter des valeurs primitives dans des structures contenant des Objects.
- Récupérer sous forme primitive des valeurs récupérées à partir de structures contenant des Objects.

Autoboxing et Unboxing

Notes

Autoboxing et Unboxing

Exemple

Sans autoboxing / unboxing

```
Integer [] tableau = new Integer[5];

for (int i = 0; i < 5; i++) {
    tableau[i] = new Integer(i);
}

for (Integer i : tableau) {
    int carre = i.intValue() * i.intValue();
    System.out.println(carre);
}
```

Avec autoboxing / unboxing

```
Integer [] tableau = new Integer[5];

for (int i = 0; i < 5; i++) {
    tableau[i] = i;
}

for (int i : tableau) {
    int carre = i * i;
    System.out.println(carre);
}
```


Autoboxing et Unboxing

Notes

Les énumérations

Principe

- Créer une pseudo classe dont les instances seraient limitées à ce qui a été conçu lors du développement.
- Sans passer par des variables statiques.

Usage classique

- Enumérer des valeurs possibles en pouvant associer des attributs et des comportements à ses valeurs.
- Aller au delà de l'utilisation de valeur numérique pour déclarer des constantes.

Les énumérations

Notes

Les énumérations

Exemple

```
public enum JourSemaine {  
  
    LUNDI(8),  
    MARDI(8),  
    MERCREDI(5),  
    JEUDI(8),  
    VENDREDI(7),  
    SAMEDI(0),  
    DIMANCHE(0);  
  
    private int nbHeuresTravaillees;  
  
    private JourSemaine(int nbHeuresTravaillees) {  
        this.nbHeuresTravaillees = nbHeuresTravaillees;  
    }  
  
    public int getNbHeuresTravaillees() {  
        return nbHeuresTravaillees;  
    }  
}  
  
int nbHeuresTravailleesSemaine = 0;  
for (JourSemaine jour : JourSemaine.values()) {  
    nbHeuresTravailleesSemaine += jour.getNbHeuresTravaillees();  
}  
  
System.out.println(nbHeuresTravailleesSemaine);
```

Les énumérations

Notes

Les imports statiques

Principe

- Utiliser une commande `import` pour utiliser directement les membres statiques d'une classe / interface.

Usage classique

- Import de méthodes de classes utilitaires (fonctions), telles que les méthodes de `java.lang.Math`.
- Utilisation de constantes déclarées dans une interface.

Précautions d'emploi

- A utiliser avec précaution, le code étant rendu moins lisible.

Les imports statiques

Notes

Les imports statiques

Exemple

Sans import statique

```
double angleDegres = 90;

double angleRadians = Math.toRadians(angleDegres);
double cosinus = Math.cos(angleRadians);
double sinus = Math.cos(angleRadians);
double tangente = Math.tan(angleRadians);

System.out.printf("%f° = %f rad, cos = %f, sin = %f, tan = %f"
, angleDegres, angleRadians, cosinus, sinus, tangente);
```

Avec import statique

```
import static java.lang.Math.*
...
double angleDegres = 90;

double angleRadians = toRadians(angleDegres);
double cosinus = cos(angleRadians);
double sinus = cos(angleRadians);
double tangente = tan(angleRadians);

System.out.printf("%f° = %f rad, cos = %f, sin = %f, tan = %f"
, angleDegres, angleRadians, cosinus, sinus, tangente);
```


Les imports statiques

Notes

Les varargs

Principe

- Méthodes pouvant prendre un nombre variable d'arguments.

Usage classique

- La méthode `printf` de `PrintWriter`.
- les méthodes de manipulation réflexive.

Limitations

- Un seul paramètre de type varargs par méthode.
- Forcément le dernier paramètre.

Les varargs

Notes

Les varargs

Exemple

Sans varargs

```
public int somme(int[] valeurs) {  
    int somme = 0;  
    for (int valeur : valeurs) {  
        somme += valeur;  
    }  
  
    return somme;  
}  
...  
int somme = o.somme(new int[] {1, 2, 3, 4, 5});  
System.out.println(somme);
```

Avec varargs

```
public int somme(int... valeurs) {  
    int somme = 0;  
    for (int valeur : valeurs) {  
        somme += valeur;  
    }  
  
    return somme;  
}  
...  
int somme = o.somme(1, 2, 3, 4, 5);  
System.out.println(somme);
```

Les varargs

Notes

Les annotations

Principe

- Associer des métadonnées aux classes, méthodes, attributs, paramètres, ...
- Pouvoir traiter ses données à divers moments du processus de développement (compilation, déploiement, exécution, ...)

Usages classiques

- En remplacement de fichiers de configuration XML (mapping O/R, descripteur de déploiement, ...).
- AOP (Programmation Orientée Aspect).

Les annotations

Notes

Les annotations

Exemple

Annotations JAXRS et Common combinées

```
@Path("/taches")
public class TacheAccesseur {

    private EntityManager entityManager;

    public void init() {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu1");
        entityManager = emf.createEntityManager();
    }

    @SuppressWarnings("unchecked")
    @GET
    public Tache[] listerTaches() {
        Query query = entityManager.createQuery("from Tache t order by t.dateLimite");
        List<Tache> liste = query.getResultList();
        return liste.toArray(new Tache[liste.size()]);
    }
}
```


Les annotations

Notes

- `StringBuilder`
- Lancement de processus avec `ProcessBuilder`
- Formatage de textes
- `Scanner`
- API de concurrence

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

StringBuilder

présentation

- Rappel : la manipulation de chaînes de caractères en Java est consommatrice de mémoire (String immuable).
- Il est préférable d'utiliser des classes dédiées comme `StringBuffer` ou `StringBuilder`.
- `StringBuilder` est une alternative à `StringBuffer`
 - Non synchronisé donc plus efficace en environnement monothreadé.
 - Dispose des mêmes méthodes.
 - Conforme au pattern builder.

StringBuilder

Notes

StringBuilder

Exemple

```
public static void main(String[] args) {  
    StringBuilder sb = new StringBuilder();  
    sb.append("Hello")  
        .append("World !")  
        .insert(5, " ")  
        .reverse()  
        .reverse();  
  
    System.out.println(sb);  
}
```

StringBuilder

Notes

Lancement de processus avec `ProcessBuilder`

Principe

- Alternative à la méthode `Runtime.exec()`
- Avec un meilleur contrôle des variables d'environnement.
- Avec la possibilité de lancer plusieurs instances de processus à partir d'une même configuration.

Utilisation

- Créer une instance de `ProcessBuilder`
- La configurer
- La lancer avec la méthode `start()`.

Lancement de processus avec ProcessBuilder

Notes

Lancement de processus avec `ProcessBuilder`

Exemple

```
ProcessBuilder pb = new ProcessBuilder("jconsole.exe");
try {
    Process p = pb.start();
    int exitValue = p.waitFor();
    System.out.println(exitValue);
} catch (IOException e) {
    e.printStackTrace();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

Lancement de processus avec ProcessBuilder

Notes


Le formatage de texte

La classe `java.util.Formatter`

- Interpréteur de chaînes de type `printf` du langage C.
- Permet de créer des modèles de chaînes de caractères et d'y injecter des valeurs de variables formatées.

Exemple

```
StringBuilder sb = new StringBuilder();  
Formatter formatter = new Formatter(sb, Locale.FRANCE);  
  
formatter.format(  
    "Bonjour %s, nous sommes le %2$tA %2$td %2$tB %2$tY",  
    "Richard", new Date());  
  
System.out.println(sb);
```



```
Bonjour Richard, nous sommes le mercredi 10 mars 2010
```

Le formatage de texte

Notes

Le formatage de texte

Syntaxe

- Syntaxe de base : `%[argument_index$]conversion`
 - `%s` : Chaîne de caractères
 - `%2$b` : Second paramètre, booléen
- Types numériques : `%[argument_index$][flags][width][.precision]conversion`
 - `%+10.4f` : numérique sur 10 caractères, avec signe, 4 décimales
- Types date / heure : `%[argument_index$][flags][width]conversion`
 - `%1$tM %1$tE, %1$tY` : Affichage du mois, du jour du mois et de l'année

Le formatage de texte

Notes

La classe `Console`

Permet d'interagir avec la console système

- Arrivée avec java 6
- Alternative à l'utilisation des entrées/sorties standards
- Utile lorsque l'on souhaite utiliser des mots de passe
 - Offre un support des mots de passe sécurisé : méthode `readPassword`
 - Récupération de la console en utilisant `System.console()`

```
Console console = System.console();

if (console == null) {
    System.out.println("Problème d'accès à la console");
} else {
    String text = console.readLine("Lecture des informations : ");
    System.out.println(text);
}
```

- Lecture des données saisies avec la méthode `readLine`

La classe Console

Notes

La classe `Console`

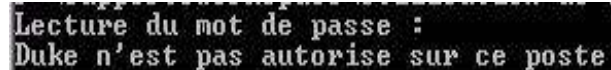
Exemple d'utilisation de la méthode `readPassword` :

```
Console console = System.console();
String text = null;
char[] password = console.readPassword("Lecture du mot de passe : ");
text = new String(password);

if("Duke".equals(text)) {
    console.format("Duke n'est pas autorise sur ce poste %n");
} else {
    console.format("Votre mot de passe est " + text + "%n");
}

console.flush();
```

- Le mot de passe n'apparaît pas à la saisie



```
Lecture du mot de passe :
Duke n'est pas autorise sur ce poste
```

La classe Console

Notes

La classe Scanner

La classe Scanner permet d'effectuer des traitements sur les flux d'entrée

- Fait partie de Java depuis la version 5 du langage
- Utilisation de toutes sources d'entrée
 - `String`
 - `System.in`
 - Les classes qui implémentent `Readable`
 - ...
- Utilisation des espaces comme séparateur (espace, tabulation et fin de ligne ...)
- Possibilité d'utiliser des expressions régulières

```
Scanner scanner = new Scanner("Utilisation d'un scanner");

while(scanner.hasNext()) {
    System.out.println(scanner.next());
}
```

- Offre de nombreuses méthodes pour traiter les différents types de données

La classe Scanner

La classe Scanner offre de nombreuses possibilités souvent ignorées. Associé aux expressions régulières, il est possible de mettre en place des application de gestion de flux très rapidement.

Notes

La classe Scanner

Présentation de certaines méthodes de la classe Scanner

méthode	description
delimiter()	Retourne l'expression régulière associée au Scanner
findInLine(Pattern)	Cherche la prochaine occurrence de l'expression indiquée
hasNext()	Retourne true s'il y a encore des données à exploiter
hasNextXXX()	XXX correspond à un type spécifique (Boolean, Byte ...)
next()	Retourne la prochaine valeur
nextXXX()	XXX correspond à un type spécifique (Boolean, Byte ...)
reset()	Remise à zéro du Scanner
skip(Pattern)	L'entrée est ignorée si elle respecte l'expression régulière

Table 1: Certaines méthodes de la classe Scanner

Cette liste n'est pas exhaustive

La classe Scanner

Notes

La classe `Scanner`

Exemple d'utilisation de la classe `Scanner` avec des expressions régulières

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
    System.out.println(result.group(i));
s.close();
```


La classe Scanner

Notes

API de concurrence

Classes utilitaires

- Nouvelles classes d'aide au développement d'applications multithreadées.
- Packages :
 - `java.util.concurrent`
 - `java.util.concurrent.atomic`
 - `java.util.concurrent.locks`
- Fournit des implémentations de pools de thread, de collections thread-safe, de sémaphores, ...

API de concurrence

Notes

API de concurrence

Threads

- Modification de la gestion de la priorisation des Thread.
- Apparition d'une numération `Thread.State`, valeur de retour de la méthode `getState()` de `Thread`.
- Nouvelle Thread Dump API permettant d'accéder à la pile d'exécution de n'importe quel Thread.
- Méthode `sleep` permettant d'interrompre un thread pour une durée inférieure à une milliseconde.

API de concurrence

Notes

Java, Programmation avancée

La boucle for each

Version 2.2

- Syntaxe de la boucle for each
- L'interface Iterable

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

La boucle for each

Présentation

- Nouvelle boucle apparue dans la version 1.5 du langage Java.
- Permet d'itérer à travers différents types de structures de données sans avoir besoin de connaître la façon dont la structure stocke ses valeurs.
- Evolution de l'API Iterator / Enumerator présente depuis la version 1.2 du langage.
- Fonctionne avec toute structure dite "itérable".

La boucle for each

Notes

La boucle for each

Syntaxe

```
for (Type variable_locale : structure_iterable) {  
    variable_locale.  
}
```

Exemples

```
String[] values = {"Un", "Deux", "Trois"};  
  
for (String value : values) {  
    System.out.println(value);  
}
```

```
List<String> values =  
    new ArrayList<String>();  
  
...  
  
for (String value : values) {  
    System.out.println(value);  
}
```

```
Map<Integer, String> map =  
    new HashMap<Integer, String>();  
  
...  
  
for (int key : map.keySet()) {  
    System.out.println(map.get(key));  
}
```

La boucle for each

Notes

La boucle for each

Types compatibles

- Toute implémentation de l'interface `java.util.Iterable`.
- Les tableaux, quelque soit leur type.

La boucle for each

Notes

L'interface Iterable

java.lang

Interface Iterable<T>

All Known Subinterfaces:

[BeanContext](#), [BeanContextServices](#), [BlockingDeque<E>](#), [BlockingQueue<E>](#), [Collection<E>](#), [Deque<E>](#), [List<E>](#), [NavigableSet<E>](#), [Queue<E>](#), [Set<E>](#), [SortedSet<E>](#)

All Known Implementing Classes:

[AbstractCollection](#), [AbstractList](#), [AbstractQueue](#), [AbstractSequentialList](#), [AbstractSet](#), [ArrayBlockingQueue](#), [ArrayDeque](#), [ArrayList](#), [AttributeList](#), [BatchUpdateException](#), [BeanContextServicesSupport](#), [BeanContextSupport](#), [ConcurrentLinkedQueue](#), [ConcurrentSkipListSet](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#), [DataTruncation](#), [DelayQueue](#), [EnumSet](#), [HashSet](#), [JobStateReasons](#), [LinkedBlockingDeque](#), [LinkedBlockingQueue](#), [LinkedHashSet](#), [LinkedList](#), [PriorityBlockingQueue](#), [PriorityQueue](#), [RoleList](#), [RoleUnresolvedList](#), [RowSetWarning](#), [SQLException](#), [ServiceLoader](#), [SQLClientInfoException](#), [SQLDataException](#), [SQLException](#), [SQLFeatureNotSupportedException](#), [SQLIntegrityConstraintViolationException](#), [SQLInvalidAuthorizationSpecException](#), [SQLNonTransientConnectionException](#), [SQLNonTransientException](#), [SQLRecoverableException](#), [SQLSyntaxErrorException](#), [SQLTimeoutException](#), [SQLTransactionRollbackException](#), [SQLTransientConnectionException](#), [SQLTransientException](#), [SQLWarning](#), [Stack](#), [SyncFactoryException](#), [SynchronousQueue](#), [SyncProviderException](#), [TreeSet](#), [Vector](#)

```
public interface Iterable<T>
```

Implementing this interface allows an object to be the target of the "foreach" statement.

Since:

1.5

Method Summary

Iterator<T>	iterator ()	Returns an iterator over a set of elements of type T.
-----------------------------------	-----------------------------	---

L'interface Iterable

Notes

L'interface Iterable

Implémenter l'interface Iterable

- Créer une classe implémentant l'interface générique.
- Implémenter la seule méthode : `public Iterator<T> iterator()`

Exemple : matrice

```
public class Matrice {  
  
    private int[][] values;  
    private int dim;  
  
    public Matrice(int dim) {  
        this.dim = dim;  
        values = new int[dim][dim];  
    }  
  
    public int get(int i, int j) {  
        return values[i][j];  
    }  
  
    public void set(int i, int j, int value) {  
        values[i][j] = value;  
    }  
  
}
```


L'interface Iterable

Notes

L'interface Iterable

Exemple : implémentation de l'interface Iterable par la classe Matrice

- Fournit une façon standard de parcourir la matrice.

```
public class Matrice implements Iterable<Integer> {  
    ...  
    public Iterator<Integer> iterator() {  
        return new Iterator<Integer>() {  
  
            private int i, j;  
  
            public boolean hasNext() {  
                return (i < dim && j < dim);  
            }  
  
            public Integer next() {  
                int val = values[i][j];  
  
                j++;  
                if (j == dim) {  
                    i++;  
                    j = 0;  
                }  
  
                return val;  
            }  
  
            public void remove() {}  
        };  
    }  
}
```

L'interface Iterable

Notes

L'interface Iterable

Exemple : création et parcours d'une matrice

```
public static void main(String[] args) {  
    Matrice m = new Matrice(3);  
    for (int i = 0, j = 0; (i < 3); j = ((j + 1) < 3 ? j+1 : 0), i = (j==0) ? i+1 : i) {  
        System.out.println(i + " , " + j);  
        m.set(i, j, i * j);  
    }  
  
    for (int val : m) {  
        System.out.println(val);  
    }  
}
```

L'interface Iterable

Notes

L'interface Iterable

Exemple : ajout d'un parcours de chaque ligne de la matrice

- Ajout d'une méthode à la classe `Matrice`.

```
...
public Iterable<Integer> getLigne(final int i) {
    return new Iterable<Integer>() {
        public Iterator<Integer> iterator() {
            return new Iterator<Integer>() {

                private int j;

                public boolean hasNext() {
                    return j < dim;
                }

                public Integer next() {
                    return values[i][j++];
                }

                public void remove() {}
            };
        }
    };
}
...
```

L'interface Iterable

Notes

L'interface Iterable

Exemple : parcours de la matrice ligne par ligne

```
public static void main(String[] args) {
    Matrice m = new Matrice(3);
    for (int i = 0, j = 0; (i < 3); j = ((j + 1) < 3 ? j+1 : 0), i = (j==0) ? i+1 : i) {
        System.out.println(i + " , " + j);
        m.set(i, j, i * j);
    }

    for (int i = 0; i < 3; i++) {
        for (int val : m.getLigne(i)) {
            System.out.print(val + " ");
        }
        System.out.println();
    }
}
```


L'interface Iterable

Notes

Java, Programmation avancée

Les énumérations en Java

Version 2.2

- Rôle des énumérations
- Structure d'une énumération
- Utilisation de méthodes

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Rôle des énumérations

- Permet de définir un ensemble de constantes dans un objet unique
- Peut être utilisée comme variable dans une classe
- Est un type de classe particulier
- Doit être défini dans son propre fichier

Rôle des énumérations

Notes

Structure d'une énumération

Utilisation du mot clé enum

- Se représente sous la forme

```
public enum JOURS{  
    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE  
};
```

Peut contenir un constructeur

```
public enum Capitale {  
    PARIS ("FRANCE", "EUROPE"),  
    WASHINGTON ("USA", "AMERIQUE");  
  
    private final String pays;  
    private final String continent;  
  
    Capitale(String pays, String continent){  
        this.pays = pays;  
        this.continent = continent;  
    }  
};
```


Structure d'une énumération

Notes

Le constructeur est forcément de visibilité "friendly" ou private.

Utilisation d'une énumération

```
public class EnumJoursTest {  
    public static void main(String[] args) {  
        JOURS unJour = JOURS.MARDI;  
        System.out.println(unJour);  
        System.out.println("//-----");  
        if(JOURS.MARDI == JOURS.valueOf("MARDI"))  
            System.out.println("unJour == MARDI");  
        System.out.println("//-----");  
        StringBuffer txt = new StringBuffer();  
        for(JOURS j : JOURS.values()){  
            switch (j) {  
                case LUNDI: txt.append("Bonjour "); break;  
                case MARDI: txt.append("Comment va "); break;  
                case MERCREDI: txt.append("Très bien "); break;  
                case JEUDI: txt.append("Je viens de la part de "); break;  
                case VENDREDI: txt.append("Dire "); break;  
                case SAMEDI: txt.append("qu'il se prépare "); break;  
                case DIMANCHE: txt.append("Pour le voyage de "); break;  
            }  
            txt.append(j).append(". \n");  
        }  
        System.out.println(txt);  
    }  
}
```



```
MARDI  
//-----  
unJour == MARDI  
//-----  
Bonjour LUNDI.  
Comment va MARDI.  
Très bien MERCREDI.  
Je viens de la part de JEUDI.  
Dire VENDREDI.  
qu'il se prépare SAMEDI.  
Pour le voyage de DIMANCHE.
```


Utilisation d'une énumération

Notes

Utilisation de méthodes

Permet d'accéder aux données membre de l'énumération en respectant les conventions d'accès Java

Peut être spécialisée pour un des éléments de l'énumération (surcharge)

```
public enum Capitale {  
    PARIS ("FRANCE", "EUROPE"),  
    WASHINGTON ("USA", "AMERIQUE") {  
        public String getContinent() {  
            return "AMERIQUE DU NORD";  
        }  
    };  
  
    private final String pays;  
    private final String continent;  
  
    Capitale(String pays, String continent) {  
        this.pays = pays;  
        this.continent = continent;  
    }  
  
    public String getPays() {  
        return this.pays;  
    }  
  
    public String getContinent() {  
        return this.continent;  
    }  
};
```

Utilisation de méthodes

Notes

La redéfinition et la covariance des méthodes en Java

- Rôle de la redéfinition
- Covariance du type de retour
- Utilisation d'une redéfinition

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Rédéfinition

Rôle

- La redéfinition d'une méthode d'une classe consiste à fournir au niveau d'une sous-classe une nouvelle implémentation de cette méthode.
- Cette implémentation masque alors complètement celle de la superclasse.
- La signature de la méthode de la sous-classe doit être exactement identique à celle de la super-classe.
- L'emploi de `super` dans le code de la méthode de la sous-classe permet de réutiliser le code écrit dans la super-classe, qui n'est plus appellable directement.
- `super` a un sens uniquement à l'intérieur d'une méthode.

Rédéfinition

Notes

Covariance du type de retour

Possible depuis Java 5

- La version Java 5.0 apporte la possibilité de modifier le type de retour d'une méthode redéfinie.
- Cependant il faut que le nouveau type soit hérité du type défini à l'origine de la méthode pour garder cette même cohésion de la redéfinition.
- Il faut toutefois faire attention aux changements de signature qui peuvent affecter les classes filles.
- Avec ce genre de modifications, il peut être difficile de remarquer une redéfinition. C'est pourquoi il est judicieux de l'indiquer avec l'annotation `@Override`.

Covariance du type de retour

Notes

Utilisation d'une redéfinition

```
public class ClasseMere {  
  
    public ClasseMere() {  
    }  
  
    public Number getNumber(int nb) {  
        return nb;  
    }  
  
}
```

```
public class ClasseFille extends ClasseMere{  
  
    public ClasseFille() {  
    }  
  
    @Override  
    public Long getNumber(int nb) {  
        Number val = super.getNumber(nb);  
        return (long) (1.0 + val.longValue());  
    }  
  
}
```

Utilisation d'une redéfinition

Notes

La classe Long hérite Number

Java, Programmation avancée

Les classes imbriquées et anonymes en Java

Version 2.2

- Comprendre les classes imbriquées et anonymes

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Classe imbriquée

Caractéristiques

- Définie à l'intérieur d'une autre classe
 - soit au niveau des attributs ou des méthodes,
 - soit à l'intérieur d'un bloc: classe locale
- Peut accéder à son contexte englobant
 - variables et méthodes d'instances si elle n'est pas statique
 - variables et méthodes statiques si elle est statique
 - variables locales constantes si elle est locale

```
public class A {  
    int x;  
    private class B {  
        int y;  
        public void m()  
        {  
            System.out.println(y + " " + A.this.x);  
        }  
    };  
    public void methode() {  
        final int i = 12;  
        class Locale {  
            public void m() {  
                System.out.println(i+" "+A.this.x);  
            }  
        }  
    }  
}
```

Classe imbriquée

Caractéristiques

La différence essentielle entre des classes locales et les fonctions membres est la possibilité de faire référence aux variables locales dans la portée de leur définition. Une restriction cruciale repose sur le fait que ces variables locales et paramètres doivent être final.

Une classe locale peut utiliser des variables locales car le compilateur donne automatiquement à la classe un champ privé pour maintenir une copie de chaque variable locale. Le constructeur ajoute aussi des arguments cachés à tous les constructeurs de classes locales pour initialiser automatiquement ces champs private à une valeur particulière.

La seule façon de travailler correctement avec des variables locales est de ne manipuler que des variables locales déclarées en final ce qui interdit tout changement. Avec cette garantie, une classe locale peut être assurée que les copies internes des variables sont en mode "sync" avec les variables locales réelles de cette classe.

Les classes locales ne peuvent contenir de champs, méthodes ou classes déclarées en static. Les membres static doivent être déclarées au niveau le plus haut. Aussi parce que les interfaces imbriquées sont implicitement static, les classes locales ne peuvent être imbriquées dans une définition d'interface.

Une autre restriction des classes locales réside dans le fait qu'elles ne peuvent être déclarées en public, protected, private ou static. Ces modificateurs sont tous utilisés pour les membres d'une classe et ne sont pas permis pour les déclarations de variables locales. Pour la même raison cela n'est pas permis avec les déclarations de classes locales.

La classe imbriquée renforce l'encapsulation

Exemple typique: structure de données et itérateur

```
public class LinkedList
{
    public interface Linkable { ... }

    // la tete de liste
    Linkable head;

    public void addToHead(Linkable node) { ... }
    public Linkable removeHead() { ... }

    // cette methode cree et retourne la liste chainee
    public Enumeration enumerate()
    {
        // definition d'une classe locale
        class Enumerator implements Enumeration {
            Linkable current;
            public Enumerator() { this.current = LinkedList.this.head; }
            public Object nextElement() {
                if (current == null)
                    throw new NoSuchElementException("LinkedList");
                Object value = current;
                current = current.getNext();
                return value;
            }
        }
        // creation et retour d'une instance de la classe Enumerator
        return new Enumerator();
    }
}
```


La classe imbriquée renforce l'encapsulation

Exemple typique: structure de données et itérateur

Une structure de données est fréquemment accompagnée d'autres classes qui en facilitent l'utilisation telles que les itérateurs.

Le mécanisme des classes imbriquées permet de définir les classes et interfaces d'itérateurs à l'intérieur de la structure de données. Les avantages d'une telle construction sont intéressants:

- rattachement des itérateurs à leur structure de définition,
- simplification de la programmation des itérateurs.

Classe imbriquée et visibilité

Champs et variables accessibles depuis une classe locale

```
class A { protected char a = 'a'; }
class B { protected char b = 'b'; }

public class C extends A {
    private char c = 'c';
    public static char d = 'd';
    public void createLocalObject(final char e) {
        final char f = 'f';
        int i = 0; // i n'est pas final donc non utilisable par une classe locale
        class Local extends B {
            char g = 'g';
            public void printVars() {
                // tous ces champs et ces variables sont accessibles
                // par cette classe
                System.out.println(g); // this.g : champ de cette classe
                System.out.println(f); // f : variable local final
                System.out.println(e); // e : argument local final
                System.out.println(C.d); // C.this.d : champ de la classe englobante
                System.out.println(C.this.c); // C.this.c : champ de la classe englobante
                System.out.println(b); // b : hérité par cette classe
                System.out.println(C.this.a); // a : hérité par la classe englobante
            }
        }
        local l = this.new Local();
        l.printVars()
    }
}
```

Classe imbriquée et visibilité

Notes

Classe imbriquée et visibilité

Champs et variables accessibles depuis une classe locale

Le main se présente de la façon suivante:

```
public static void main(String[] args) {  
    // cree une instance de la classe englobante et invoque la methode  
    // qui definit et cree la classe local  
    C c = new C();  
    c.createLocalObject('e');// passe une valeur au param. final e  
}
```

Une classe locale peut utiliser:

- des champs définis dans la classe locale elle-même
- des champs hérités par la classe locale
- des paramètres final dans la portée de la définition de la classe locale,
- des champs de la classe englobante,
- des champs hérités par la classe englobante.

Classe imbriquée et visibilité

Notes

Classe anonyme

Caractéristiques

- Classe sans nom qui ne peut pas être réutilisée
- Confond les opérations de définition de classe et d'instanciation
- Possède les mêmes facultés qu'une classe locale
- Le nom de classe qui suit l'opérateur new peut être:
 - une classe: la classe anonyme en est alors une sous-classe
 - une interface: la classe anonyme est alors une sous-classe de Object

```
public interface Description {  
    public String decrisTOI() ;  
}  
  
Itineraire iti = new Itineraire();  
Capitale paris = new Capitale("Paris");  
iti.ajouter(paris);  
iti.ajouter(new Description() {  
    public String decrisTOI() {  
        return "La Tour Eiffel";  
    }  
});
```

Classe anonyme

Caractéristiques

Une classe anonyme est une extension du concept de classe locale. Au lieu de déclarer une classe locale avec un traitement Java et ensuite l'instancier avec un autre traitement Java, une classe anonyme combine les deux activités dans une seule expression Java. C'est une extension de l'utilisation de l'opérateur new.

Bien entendu, une classe anonyme ne porte pas de nom et parce qu'elle est définie dans la même expression qui l'instancie, elle ne peut être instanciée qu'une seule fois.

A l'exception de ces différences, une classe anonyme est grandement similaire à une classe locale dans son comportement.

Lorsque l'on écrit une classe adaptateur, le problème de savoir si on nomme la classe ou pas ne dépend que d'un point de vue de clarté. Cela entraîne un style de programmation plutôt qu'un style de fonctionnement.

Sur l'exemple de la page précédente si le nom suivant l'opérateur new est un nom de classe, la classe anonyme est une sous classe de la classe nommée, mais si le nom est un nom d'interface (comme dans l'exemple) la classe anonyme est simplement une implémentation de l'interface. Dans ce cas, la classe anonyme est toujours une sous-classe de Object.

Il n'y a aucune possibilité de spécifier une clause extends ou implements

Classe anonyme

Classes anonymes contre classes locales

```
import java.util.*;
public class LinkedList
{
    public interface Linkable { ... }

    // la tete de liste
    Linkable head;

    public void addToHead(Linkable node) { ... }
    public Linkable removeHead() { ... }

    // cette methode cree et retourne la liste chainee
    public Enumeration enumerate()
    {
        // instancie et retourne cette implémentation
        return new Enumeration() {
            Linkable current = head; // obligatoire car il n'y a pas
                                   // de constructeur pour une classe anonyme
            public boolean hasMoreElements() { return current != null; }
            public Object nextElement() {
                if (current == null)
                    throw new NoSuchElementException("LinkedList");
                Object value = current;
                current = current.getNext();
                return value;
            }
        }; // le ; est obligatoire
    }
}
```


Classe anonyme

Classes anonymes contre classes locales

Parce que cette classe n'a pas de nom, il n'est pas possible de définir de constructeur dans le corps de cette classe. Ainsi, tous les arguments spécifiés entre les parenthèses suivant le nom de la super-classe dans une définition de classe anonyme sont implicitement passés aux constructeurs de la super-classe. Les classes anonymes sont couramment utilisées comme des sous-classes simples dont les potentiels constructeurs ne prendraient aucun argument.

Les classes anonymes ne portant pas de nom, la compilation de l'exemple de la page précédente entraîne la création de deux fichiers: un pour la classe englobante et un autre pour la classe anonyme. Le nom qui est donné au fichier correspondant à la classe anonyme porte le nom de la classe englobante suivi du rang de la classe anonyme. Pour notre exemple, cela donne: `Liste.class` et `Lister$1.class`.

Dans votre code, lorsque vous avez le choix entre utiliser une classe anonyme et employer une classe locale, la décision découle souvent de choix de style de programmation. En général, vous devrez considérer l'usage de classe anonyme à la place de classe locale dans les cas suivants:

- la classe a un corps très réduit,
- seule une instance de la classe est nécessaire,
- la classe est utilisé juste après avoir été définie,
- nommer la classe ne fournira aucune aide supplémentaire,
- aucun constructeur n'est utile pour la définition de l'instance utile dans la suite du code.

Java, Programmation avancée

La généricité en Java

Version 2.2

- Présentation de la généricité
- Utilisation de la généricité
- Définition d'une classe générique simple
- Définition d'une méthode générique

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Présentation de la généricité

Utilité

- Ecrire des classes et des opérations qui puissent s'appliquer à des familles de types
 - exemple: structures de données génériques
- Bénéficier de contrôles de types plus stricts

Avant la généricité (avant Java 5)

```
ArrayList liste = new ArrayList();  
liste.add("Un texte");  
liste.add(new Integer(3));
```

- Possibilité d'ajouter des objets sans vérification de type
- Peut poser des problèmes lors de la récupération des informations
 - Vérification de type avant d'affecter la valeur dans une variable
- Transtypage fréquent

Présentation de la généricité

Notes :

Utilisation de la généricité

Avec la généricité

```
ArrayList<String> liste = new ArrayList<String>();  
liste.add("Un texte");  
liste.add(new Integer(3)); // Impossible, pas de type String
```

- Typage plus strict: impossible d'ajouter un élément de type différent de celui attendu
- Plus de transtypage
 - Ce code sans généricité

```
ArrayList liste = new ArrayList ();  
liste.add(new A());  
//...  
A unA = (A) liste.get(0);
```

- devient avec la généricité

```
ArrayList<A> liste = new ArrayList<A> ();  
liste.add(new A());  
//...  
A unA = liste.get(0);
```

Utilisation de la généricité

Notes :

La généricité permet un usage des structures de données plus respectueux des types et une écriture plus élégante.

Définition d'une classe générique simple

Légère modification de la construction de la classe

```
public class MaPaire<T>
```

En C++ on écrit **template <class T> class MaListe**

- T qui est inclut entre < > est une variable de type.
- T sera remplacé par le type spécifié lors de l'utilisation de la classe

```
MaPaire<String> liste = new MaPaire<String>("un", "deux");
```

T est remplacé par String

Définition d'une classe générique simple

Notes :

Un exemple complet

```
public class MaPaire<T> {  
  
    private T premier;  
    private T second;  
  
    public MaPaire(){  
        premier = null;  
        second = null;  
    }  
    public MaPaire(T premier, T second){  
        this.premier = premier;  
        this.second = second;  
    }  
  
    public T getPremier(){  
        return premier;  
    }  
  
    public void setPremier(T premier){  
        this.premier = premier;  
    }  
  
    public T getSecond(){  
        return second;  
    }  
  
    public void setSecond(T second){  
        this.second = second;  
    }  
}  
  
MaPaire<String> liste = new MaPaire<String>("un", "deux");  
System.out.println("Valeur de Premier = " + liste.getPremier() + " et valeur de Second = "  
    + liste.getSecond());
```

Un exemple complet

Notes :

Méthodes génériques

Définition de la méthode 'static'

```
public static <T> T max(T val1, T val2) // Presque juste
```

Pour pouvoir comparer des valeurs, doit étendre java.lang.Comparable

```
public static <T extends Comparable> T max(T val1, T val2)
```

Exemple

```
public static <T extends Comparable> T max(T val1, T val2){  
    return (val1.compareTo(val2) > 0 ? val1 : val2);  
}  
  
System.out.println("La valeur la plus grande entre 10 et 5 est : " + MethodeGenerique.max(10, 5
```

Résultat :

La valeur la plus grande entre 10 et 5 est : 10

Méthodes génériques

Notes :

Java, Programmation avancée

Généricité avancée en Java

Version 2.2

- Règles d'utilisation
- Règles d'héritage
- Types joker
- Mécanisme d'effacement et ses conséquences
- Interopérabilité entre types génériques et non génériques

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Restrictions et limites

Type primitifs invalides comme paramètre de type

```
MaPaire<int> liste = new MaPaire<int>(1, 5); // Erreur, type primitif
MaPaire<Integer> liste = new MaPaire<Integer>(1, 5); // OK
```

Une demande de type sur un objet générique ne retourne que du type brut

```
if(liste instanceof MaPaire)
    équivaut à
if(liste instanceof MaPaire<Integer>)

if(liste instanceof MaPaire<T>) // T n'est pas testé
```

- Une comparaison pourra retourner vrai

```
MaPaire<String> paireString = new MaPaire<String>();
MaPaire<MaPaire<Integer>> pairePaire = new MaPaire<MaPaire<Integer>>();

if(paireString.getClass() == pairePaire.getClass())
    System.out.println("Attention, retourne vrai !");

Retourne
    Attention, retourne vrai !
```


Restrictions et limites

Type primitifs invalides comme paramètre de type

Utilisation de l'autoboxing sur les données de type Wrapper (version objet du type primitif).

Restrictions et limites

Il n'est pas possible de déclencher ou d'intercepter une exception

```
public class ExceptionErreur <T> extends Throwable // Un Generic ne peut étendre Throwable
```

Il est possible de spécifier une exception en se basant sur les variables de type

```
public static <T extends Throwable> void doException(T t) throws T
{
    try{
        // du code
    } catch(Throwable th){
        t.initCause(th);
        throw t;
    }
}
```

Il n'est pas possible de déclarer des tableaux de types avec paramètres

```
MaPaire<String>[] table = new MaPaire<String>(10); // Erreur
```

Pour collecter des objets, l'utilisation d'un ArrayList convient parfaitement

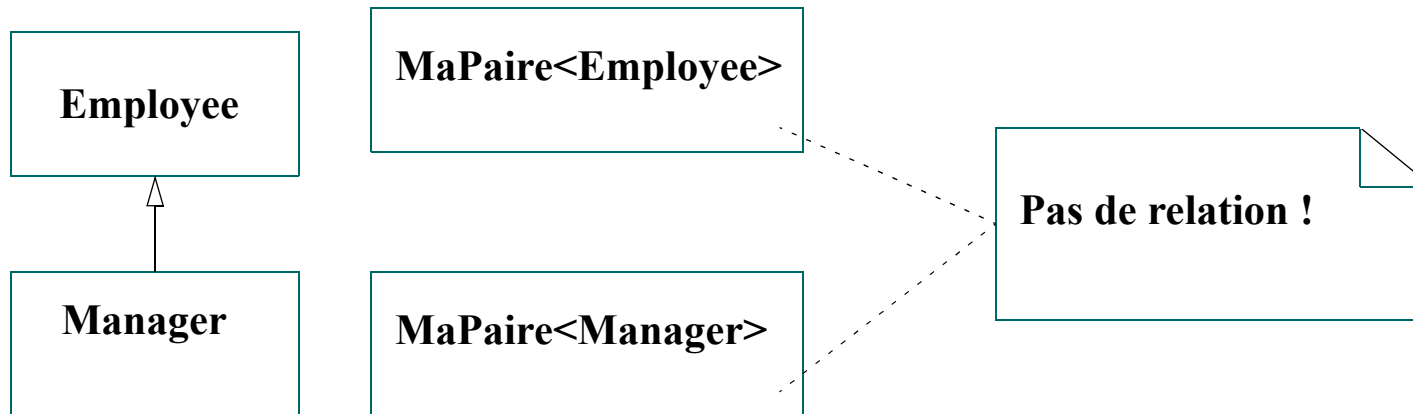
```
ArrayList<MaPaire<String>> table = new ArrayList<MaPaire<String>>
```

Restrictions et limites

Notes

Règle d'héritage pour les types génériques

MaPaire<classe> et MaPaire<sous-classe> sont de types différents



Le code suivant ne fonctionne donc pas

```
MaPaire<Employee> paireEmployee = new MaPaire<Employee>();  
MaPaire<Manager> paireManager = paireEmployee;
```

Ce code retourne une erreur "type mismatch"

Règle d'héritage pour les types génériques

Notes

Les paramètres de type joker

Permettent de ne pas spécifier un type précis

```
MaPaire<? extends Employee> paireJoker;
```

Dans l'exemple précédent, le type sera Employee ou une classe qui en hérite

Permet d'affecter n'importe quel type ou sous-type

```
Manager manager1 = new Manager();  
Manager manager2 = new Manager();  
paireJoker = new MaPaire<Manager>(manager1, manager2);  
Employee emp = paireJoker.getPremier();
```

Ne permet pas une affectation des données de la classe

```
paireJoker.setPremier(emp); // Erreur, pas d'affectation possible
```

Le compilateur sait qu'il attend une sous classe, mais il ne sait pas laquelle

Les paramètres de type joker

Notes

Les paramètres de type joker

Possibilité d'utiliser une limite de super-type pour les jokers

```
MaPaire<? super Manager> paireSuper;
```

Permet d'effectuer une affectation avec un objet du type indiqué

```
MaPaire<? super Manager> paireSuper = new MaPaire<Manager>();  
paireSuper.setPremier(manager1);
```

Attention, l'exemple ne fonctionne pas si l'on affecte un objet de type Employee, mais fonctionne avec un sous-type de Manager

Les paramètres de type joker

Notes

Mécanisme d'effacement de type

L'effacement de type (ou "type erasure") s'effectue à la compilation.

- Il s'agit d'effacer les paramètres génériques du code au moment de la compilation afin de ne garder qu'une seule classe représentative de tous les différents types déclarés.
- Le compilateur effectue alors:
 - Des vérifications de typage.
 - Le remplacement des types paramétrés par des types non génériques dites bruts.
 - Le remplacement des types paramétrés par un type plus général et englobant
 - L'ajout des casts si besoin.
 - L'ajout des méthodes "pont" aux sous-classes pour la redéfinition de certaines méthodes génériques en cas de polymorphisme.

Exemples simples d'effacement:

- `ArrayList<Integer>` et `ArrayList<String>` sont considérés comme l'interface `ArrayList`.
- `<T>` est remplacé par `Object`.
- `ArrayList<String>[]` est considéré comme `ArrayList[]`.

Mécanisme d'effacement de type

Notes

Mécanisme d'effacement de type

Limites et conséquences

- Il n'est pas possible d'implémenter plusieurs interfaces de même type brut.
- Il n'est pas possible de donner la même signature pour différentes méthodes (ex: `get (T)` et `get (Object)`).
- Impossibilité d'utiliser la méthode `instanceof` sur un type paramétré sauf avec le type joker.
- Impossibilité de créer un objet avec un paramètre de type (ex: `new T ()`).
- Impossibilité de créer des tableaux dont le type est un paramètre de type ou encore si le type des éléments est un type paramétré.
- Dans un contexte de variable ou de méthode statique, il est interdit d'utiliser le paramètre d'une classe générique.

Mécanisme d'effacement de type

Notes

Interopérabilité entre types génériques et non génériques

- Il est possible d'utiliser les types bruts pour les types paramétrés et inversement, que ce soit pour l'affectation ou les paramètres de méthode.

```
public static void m(ArrayList al) {  
    System.out.println(al.get(0));  
}  
  
public static void m2(ArrayList<Integer> al)  
{  
    System.out.println(al.get(0));  
}
```

```
public static void main(String[] args)  
{  
    ArrayList<Integer> alInt =  
        new ArrayList<Integer>();  
  
    ArrayList al = new ArrayList();  
  
    alInt.add(2);  
    al.add("chose");  
  
    m(alInt);  
    m2(al);  
  
    alInt = al; // ou al = alInt;  
  
}
```

Interopérabilité entre types génériques et non génériques

Notes

Le compilateur émettra un avertissement précisant qu'il ne peut garantir la cohérence du type donné en paramètre.

Java, Programmation avancée

Les collections en Java

Version 2.2

- Le concept de Collection
- Les types de collections
- Les listes

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Qu'est-ce qu'une collection

Concept de structures de données

- Stockage de données sous forme de liste chaînées, tables de hachage etc..
- Devenu courant à travers les différents langages Objet
 - STL en C++
 - Collections Smalltalk, ...
- Basé sur des design-patterns

Met également à disposition des algorithmes, des opérations de tri ...

- sort
- binarySearch
- copy
- etc.

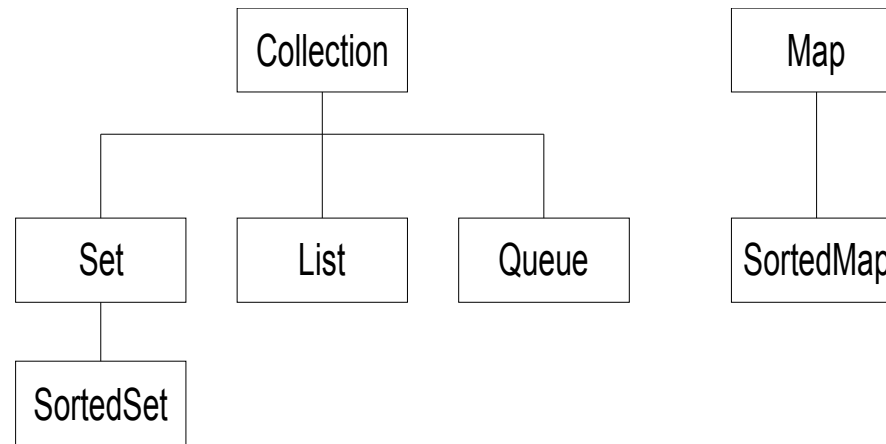
Une collection peut être:

- triée (*sorted*): tri selon l'ordre naturel des éléments ou selon un ordre personnalisé.
- ordonnée (*ordered*): parcours des éléments selon in ordre donné (ordre d'insertion par exemple).

Qu'est-ce qu'une collection

Notes

Les interfaces de Collection



Les interfaces de base de Collection

Permettent de représenter des concepts de base

- Listes (interface `List`): liste d'éléments,
- Sets (interface `Set`): ensemble d'éléments uniques,
- Queues (interface `Queue`): ensemble d'éléments ordonnés dans l'ordre où ils doivent être traités (FIFO),
- Maps (interface `Map`) : ensemble de paires clé/élément; les clés étant uniques.
- Itérateurs (interface `Iterator`): parcours des éléments d'une collection

Toutes les interfaces de collections sont génériques

Les interfaces de Collection

Notes

Attention: bien distinguer les interfaces des implémentations !

Les différentes classes de collection

Interface	Classe d'implémentation	Description
List	ArrayList	Une séquence indexée qui grandit et se réduit de manière dynamique
List	LinkedList	Une séquence ordonnée qui permet des insertions et des retraits rapides à n'importe quel endroit
List	Vector	Identique à ArrayList : les accès sont synchronisés
Set	HashSet	Une collection qui n'autorise pas les répliques (non ordonnée)
Set	TreeSet	Un ensemble trié d'éléments uniques
Set	EnumSet	Un ensemble de valeurs de type énuméré
Set	LinkedHashSet	Un ensemble d'éléments uniques qui se souvient de l'ordre d'insertion des éléments
Queue	PriorityQueue	Une collection qui permet un retrait de l'élément le plus petit
Map	HashMap	Une collection qui stocke des associations clé/valeur (clé uniques)
Map	Hashtable	Identique à HashMap mais les accès sont synchronisés
Map	TreeMap	Une collection qui stocke des associations clé/valeur : les clés sont triées dans l'ordre croissant
Map	EnumMap	Identique à TreeMap mais pour des types énumérés
Map	LinkedHashMap	Identique à une HashMap mais qui se souvient de l'ordre d'ajout des entrées

Table 2: Classes d'implémentation des collections

Les différentes classes de collection

Notes

L'interface Collection

Opérations de base

- ajout d'un élément: `boolean add(E)`
- suppression d'un élément: `E remove(E)`
- recherche d'un élément: `boolean contains(E)`
- parcours de la collection: `Iterator<E> iterator()`

Exemple d'utilisation de l'interface Collection

```
Collection<String> liste = new ArrayList<String>();

liste.add("un");
liste.add("deux");
liste.add("trois");
liste.add("quatre");
for (int i = 0; i < liste.size(); i++)
    System.out.println(liste.toArray()[i]); // Affichage de l'élément courant de la collec-
```

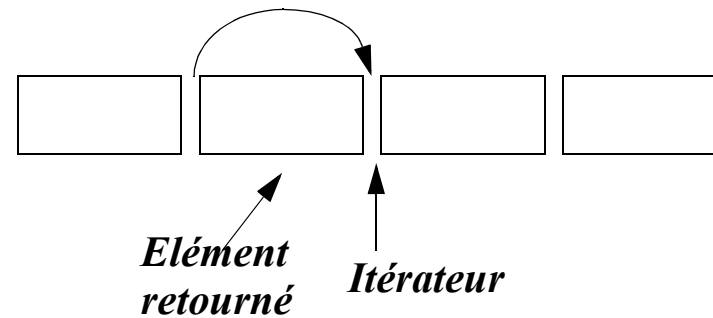

L'interface `Collection`

Notes

L'interface `Iterator`

Simplifie le parcours des éléments des collections un par un

Passage d'un élément à un autre à l'aide de la méthode *next()*



Fonctionnement des itérateurs

Parcours d'une collection avec un `Iterator`

```
Iterator<String> iter = liste.iterator();  
// parcours avec un itérateur  
while(iter.hasNext())  
    System.out.println(iter.next());  
  
// parcours avec une boucle for each  
for (String valeur: liste){  
    System.out.println(valeur);  
}
```

L'interface `Iterator`

Notes

- Syntaxe de la boucle `for each`:

```
for (Type variable_locale : structure_iterable) {  
    //utilisation de la variable_locale.  
}
```

- Pour qu'une structure soit parcourable avec la boucle "for each", cette structure doit implémenter l'interface `Iterable`. Cette interface définit une unique méthode:

```
public Iterator iterator()
```

L'interface `List`

Ensemble d'éléments ordonnés : tient compte d'un index

Opérations principales

- récupération d'un élément à un index donné: `E get(int index)`
- recherche de l'index d'un élément: `int indexOf(E element)`
- ajout d'un élément à un index donné: `void add(int index, E element)`
- remplacement un élément à un index donné: `E set(int index, E Element)`
- suppression d'un élément à un index donné: `E remove (int index)`

Principales implémentations

- `ArrayList`: parcours rapide et accès à n'importe quelle position rapide mais ajout/suppression au milieu lentes
- `LinkedList`: liste doublement chaînée. Plus rapide pour les ajouts/suppression au milieu mais plus lente en parcours que l'`ArrayList`.

Exemple d'utilisation d'`ArrayList`

```
List<String> liste = new ArrayList<String>();  
liste.add("Texte 1");  
System.out.println(liste.get(0));
```

L'interface `List`

Notes

L'interface Set

Ensemble d'éléments uniques: unicité basée sur la méthode `equals()`

Opérations identiques à celles de l'interface `Collection`

Principales implémentations

- `HashSet`: set non trié et non ordonné. S'appuie sur la valeur retournée par la méthode `hashCode()`
- `LinkedHashSet`: version ordonnée de `HashSet`: ordre d'insertion des éléments. Liste doublement chaînée.
- `TreeSet`: ensemble trié d'éléments uniques.

Exemple d'utilisation d'`LinkedHashSet`

```
Set<String> set = new LinkedHashSet<String>();  
boolean res1 = set.add("Texte 1");  
boolean res2 = set.add("Texte 1");  
System.out.println(res1); // affiche true  
System.out.println(res2); // affiche false
```

L'interface **Set**

Notes

L'interface Queue

Interface permettant d'implémenter une pile FIFO

Ne donne aucune information sur l'implémentation de la queue

- Tableau circulaire
- Liste chaînée

Principales implémentations

- `LinkedList`: liste doublement chaînée.
- `PriorityQueue`: les éléments sont triés selon leur ordre naturel ou un ordre personnalisé. Le premier élément à traiter est celui est *le plus petit*

Opérations principales

- Récupération du premier élément à traiter (sans suppression de l'élément): `E element()` ou `E peek()`
- Récupération du premier élément à traiter (avec sa suppression de la pile): `E poll()` ou `E remove()`
- Ajout d'un élément à la pile: `boolean offer(E element)`

L'interface Queue

Notes

Implémentations

- `ArrayBlockingQueue`
- `ConcurrentLinkedQueue`
- `DelayQueue`
- `LinkedBlockingQueue`
- `LinkedList`
- `PriorityBlockingQueue`
- `PriorityQueue`
- `SynchronousQueue`

Queues manipulables par les deux extrémités (Java 6)

- Interface `java.util.Dequeue`
- Interface `java.util.BlockingDequeue`

L'interface Map

Permettent de réaliser des associations clé/valeur

- Utile pour la mise en place de tables de hachage

Opérations principales

- ajout d'une paire clé/valeur: `V add(K key, V value)`
- récupération d'un élément à partir de sa clé: `V get(K key)`
- suppression d'un élément à partir de sa clé: `V remove(K key)`
- récupération de la liste des paires : `Set<Map.Entry<K, V>> entrySet()`
- récupération de la liste des clés: `Set<K> keySet()`

Principales implémentations

- `HashMap`: Map non ordonnée
- `LinkedHashSet`: version ordonnée du `HashMap`
- `TreeMap`: version triée selon l'ordre naturel ou un ordre personnalisé

L'interface Map

Notes

L'interface Map

Exemple

```
Map<String, Personne> abonne = new HashMap<String, Personne>();
abonne.put("1", new Personne("Louis XIV"));
abonne.put("3", new Personne("Charles V"));
abonne.put("3", new Personne("Napoléon"));
abonne.put("4", new Personne("Dagobert"));
//suppression d'une personne
abonne.remove("3");
// Parcourt de l'objet
for(Map.Entry<String, Personne> valeur: abonne.entrySet()){
    System.out.println("L'abonné n° " + valeur.getKey() + " est "
        + valeur.getValue().getNom());
}
```

L'interface **Map**

Notes

Il faut faire attention aux objets stockés dans les `HashMap`, en effet pour ceux-ci il est nécessaire de redéfinir les méthodes **`equals`** et **`hashCode`**.

Critères de choix d'une implémentation

Compromis

- Unicité des éléments
- Accès aux éléments: par index, par ordre d'insertion, par clé
- Ordre de parcours des éléments
- Éléments triés ou non
- Opérations les plus courantes: insertions/suppressions ou parcours

Critères de choix d'une implémentation

Notes

Java, Programmation avancée

Les structures de hashage

Version 2.2

- Utilité du hashage dans les collections
- Les méthodes hashCode et equals

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Utilité du hashcode

HashSet et LinkedHashSet

- Unicité d'un élément garanti dans un ensemble.
- Avant chaque insertion, on doit vérifier que l'élément inséré n'existe pas déjà dans l'ensemble.
 - Le parcours de la totalité de l'ensemble serait trop long.
 - Nécessité de gérer plus efficacement le stockage des éléments pour facilement les retrouver.

Hashtable, HashMap

- Unicité de la clé garantie dans l'ensemble.
- Avant chaque insertion, on doit vérifier si la clé n'est pas déjà utilisée.
- Pour récupérer un élément, on doit rechercher sa clé.
 - Le parcours de la totalité des éléments serait trop long.
 - Nécessité de gérer plus efficacement le stockage des éléments pour optimiser leur récupération.

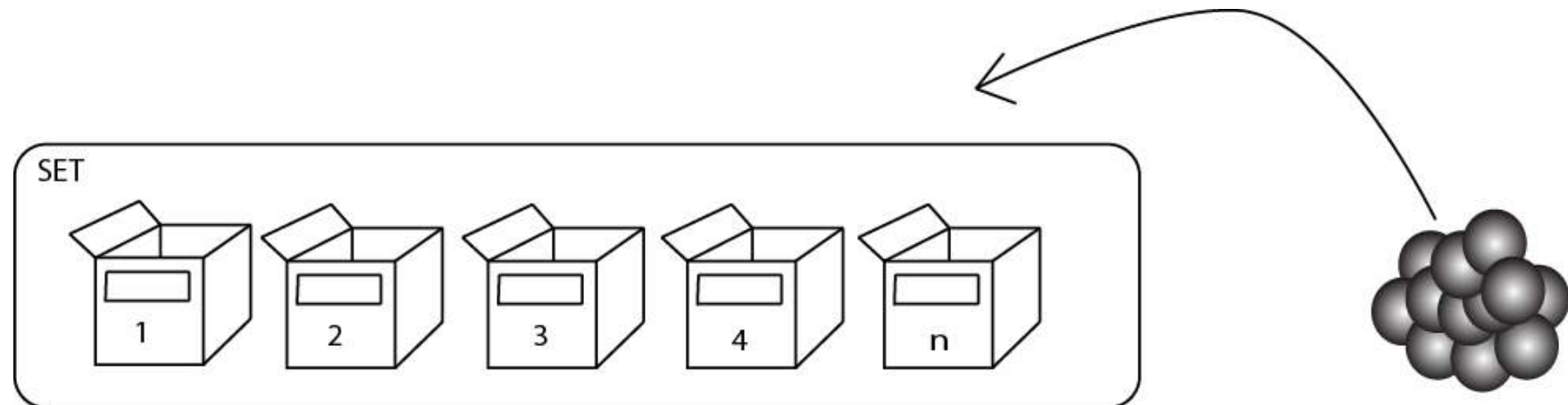
Utilité du hashcode

Notes

Utilité du hashcode

Répartition des éléments par hashcode

- L'ensemble est découpé en n sous-ensembles.
- A chaque sous-ensemble est associé un numéro.
- Quand on ajoute un élément à l'ensemble, on calcule une valeur qui lui est associée pour déterminer dans quel sous-ensemble le stocker.



- C'est la méthode `hashCode` de chaque objet qui permet de calculer cette valeur de répartition.

Utilité du hashCode

Notes

Redéfinir la méthode hashCode

- Méthode de la classe `java.lang.Object`.
- `public int hashCode()`
- Quelques règles à respecter :
 - L'invocation de la méthode `hashCode` sur un objet renvoie toujours la même valeur pour peu que les attributs utilisés dans sa méthode `equals` n'aient pas été modifiés entre temps.
 - Deux objets égaux, au sens `equals` du terme, doivent retourner la même valeur de hascode.
 - Deux objets non égaux, au sens `equals` du terme, peuvent renvoyer la même valeur de hashcode. Cependant, une telle implémentation, même si elle est légale, ne serait pas efficace.

Les méthodes `equals` et `hashCode` sont intimement liées.

Redéfinir la méthode hashCode

Notes

Redéfinir la méthode hashCode

Quelques exemples

```
public class Cafe {  
  
    private String variete;  
  
    public int hashCode() {  
        // En supposant que variete soit non nul  
        return variete.hashCode();  
    }  
}
```

```
public class TasseACafe {  
  
    private int diametre;  
  
    public int hashCode() {  
        // Valide, mais peu efficace  
        return 2;  
    }  
}
```


Redéfinir la méthode hashCode

Notes

Redéfinir la méthode hashCode

Un contre-exemple

```
public class Cafetiere {  
  
    private String marque;  
  
    public int hashCode() {  
        // Invalide, illégal, posera des problèmes  
        return (int) Math.random();  
    }  
}
```

Redéfinir la méthode hashCode

Notes

La méthode equals

Utilité

- Permet de spécifier des règles Métier pour vérifier l'égalité entre deux objets.
- Dans le cadre des collections, elle est utilisée :
 - dans les Set pour vérifier l'unicité d'un élément d'un ensemble,
 - dans les Map pour retrouver une clé.
- Le bon fonctionnement des `HashSet`, `LinkedHashSet`, `HashMap` et `Hashtable` dépend :
 - de la bonne implémentation de la méthode `hashCode`,
 - de la bonne implémentation de la méthode `equals`,
 - de la cohérence entre ses deux méthodes.

La méthode equals

Notes

Redéfinir la méthode equals

- Méthode de la classe `java.lang.Object`.
- `public boolean equals(Object obj)`
- Quelques règles à respecter :
 - **Reflexivité** : `x.equals(x)` est toujours vrai si `x` est non null.
 - **Symétrie** : `x.equals(y)` si et seulement si `y.equals(x)`.
 - **Transitivité** : si `x.equals(y)` et `y.equals(z)` alors `x.equals(z)`.
 - **Consistence** : si `x.equals(y)` renvoie `true` (resp. `false`) à un instant `t`, un nouvel appel de `x.equals(y)` doit renvoyer `true` (resp. `false`) si aucune information utilisée dans la comparaison effectuée dans `equals` n'a été modifiée.
 - `x.equals(null)` renvoie toujours `false`.

Redéfinir la méthode equals

Notes

Redéfinir la méthode equals

Exemple

```
public class Cafe {  
  
    private String variete;  
  
    ...  
  
    public boolean equals(Object obj) {  
        if (obj == null || !(obj instanceof Cafe)) {  
            return false;  
        }  
  
        return variete.equals(((Cafe) obj).variete);  
    }  
}
```


Redéfinir la méthode equals

Notes

Redéfinir la méthode equals

Contre-exemple

```
public class TasseACafe {  
  
    private int diametre;  
  
    ...  
  
    @Override  
    public boolean equals(Object obj) {  
        return true;  
    }  
}
```

Redéfinir la méthode equals

Notes

Cohérence equals / hashCode

Tableau récapitulatif des règles de cohérence

Condition	Obligatoire	Permis
<code>x.equals(y) == true</code>	<code>x.hashCode() == y.hashCode()</code>	
<code>x.hashCode() == y.hashCode()</code>		<code>x.equals(y) == true</code>
<code>x.equals(y) == false</code>		
<code>x.hashCode() != y.hashCode()</code>	<code>x.equals(y) == false</code>	

Cohérence equals / hashCode

Notes

Java, Programmation avancée

Collections triées

Version 2.2

- SortedSet, SortedMap et autres structures triées
- Ordre naturel
- Comparable et Comparator
- Tri de listes

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Collections triées

Interface SortedSet

- Ensemble d'éléments avec :
 - garantie de l'unicité des éléments (`Set`),
 - tri des éléments.
- Les éléments sont triés :
 - en utilisant une implémentation de `java.util.Comparator`, ou
 - en utilisant l'ordre naturel des éléments (`java.lang.Comparable`).
- Implémentation : `java.util.TreeSet`.

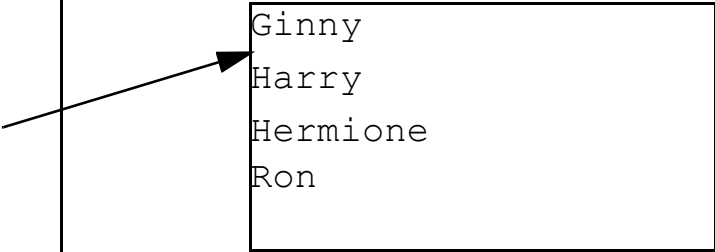
Collections triées

Notes

Collections triées

Exemple SortedSet

```
Set<String> ensemble = new TreeSet<String>();  
  
ensemble.add("Ron");  
ensemble.add("Ginny");  
ensemble.add("Harry");  
ensemble.add("Hermione");  
  
for (String prenom : ensemble) {  
    System.out.println(prenom);  
}
```



Ginny
Harry
Hermione
Ron

L'ordre utilisé est l'ordre naturel de `String`.

Collections triées

Notes

Collections triées

Interface SortedMap

- Ensemble d'éléments avec :
 - stockage et récupération à partir d'une valeur de clé (Map),
 - tri des clés.
- Les clés sont triées :
 - en utilisant une implémentation de `java.util.Comparator`, ou
 - en utilisant l'ordre naturel des clés (`java.lang.Comparable`).
- Implémentation : `java.util.TreeMap`.

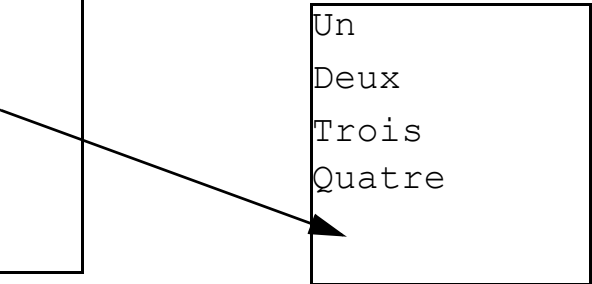
Collections triées

Notes

Collections triées

Exemple SortedMap

```
Map<Integer, String> nombres = new TreeMap<Integer, String>();  
  
nombres.put(4, "Quatre");  
nombres.put(2, "Deux");  
nombres.put(3, "Trois");  
nombres.put(1, "Un");  
  
for (String nombre : nombres.values()) {  
    System.out.println(nombre);  
}
```



Un
Deux
Trois
Quatre

L'ordre utilisé est l'ordre naturel de Integer.

Collections triées

Notes

Ordre naturel

- Si aucun ordre n'est spécifié lors de la création d'un `SortedSet` ou d'un `SortedMap`, l'**ordre naturel** de la classe des éléments est utilisé.
- Une classe ne dispose d'un ordre naturel que si elle implémente l'interface `java.lang.Comparable`.

L'interface Comparable

- Interface générique : `interface Comparable<T>`
- Impose l'implémentation de la méthode `int compareTo(T o)` :
 - Renvoie une valeur négative si l'objet courant est inférieur à l'objet passé en paramètre.
 - Renvoie 0 si l'objet courant est égal à l'objet passé en paramètre
 - Renvoie une valeur positive l'objet courant est supérieur à l'objet passé en paramètre

Ordre naturel

Notes

Ordre naturel

Exemple : sans implémenter Comparable

```
public class Cafe {  
  
    private String variete;  
  
    public Cafe(String variete) {  
        this.variete = variete;  
    }  
    ...  
    public static void main(String[] args) {  
        Set<Cafe> cafes = new TreeSet<Cafe>();  
  
        cafes.add(new Cafe("Moka"));  
        cafes.add(new Cafe("Arabica"));  
        cafes.add(new Cafe("Robusta"));  
  
        for (Cafe cafe : cafes) {  
            System.out.println(cafe.variete);  
        }  
    }  
}
```

```
Exception in thread "main" java.lang.ClassCastException:  
Cafe cannot be cast to java.lang.Comparable  
    at java.util.TreeMap.put(Unknown Source)  
    at java.util.TreeSet.add(Unknown Source)  
at Cafe.main(Cafe.java:33)
```

Ordre naturel

Notes

Ordre naturel

Exemple d'implémentation de Comparable

```
public class Cafe implements Comparable<Cafe> {  
  
    private String variete;  
    ...  
    public int compareTo(Cafe o) {  
        return variete.compareTo(o.variete);  
    }  
  
    public static void main(String[] args) {  
        Set<Cafe> cafes = new TreeSet<Cafe>();  
  
        cafes.add(new Cafe("Moka"));  
        cafes.add(new Cafe("Arabica"));  
        cafes.add(new Cafe("Robusta"));  
  
        for (Cafe cafe : cafes) {  
            System.out.println(cafe.variete);  
        }  
    }  
}
```

Arabica
Moka
Robusta

Ordre naturel

Notes

L'interface Comparator

- Peut être utilisée pour spécifier un ordre lors de la création d'une collection triée.
 - Soit parce que les éléments ne possèdent pas d'ordre naturel.
 - Soit pour utiliser un autre ordre que l'ordre naturel des éléments.
- Interface générique : `interface Comparator<T>`
- Impose l'implémentation de la méthode `int compare(T o1, T o2)`
 - Renvoie une valeur négative si o1 est inférieur à o2.
 - Renvoie 0 si o1 est égal à o2.
 - Renvoie une valeur positive si o1 est supérieur à o2.

L'interface Comparator

Notes

L'interface Comparator

Exemple

```
class CafeInverseComparator implements Comparator<Cafe> {  
  
    public int compare(Cafe o1, Cafe o2) {  
        return o2.variete.compareTo(o1.variete);  
    }  
}
```

```
public static void main(String[] args) {  
    Set<Cafe> cafes = new TreeSet<Cafe>(new CafeInverseComparator());  
  
    cafes.add(new Cafe("Moka"));  
    cafes.add(new Cafe("Arabica"));  
    cafes.add(new Cafe("Robusta"));  
  
    for (Cafe cafe : cafes) {  
        System.out.println(cafe.variete);  
    }  
}
```

```
Robusta  
Moka  
Arabica
```


L'interface Comparator

Notes

Tri de listes quelconques

- La classe `java.util.Collections` propose deux méthodes statiques permettant de trier des listes :
 - `public static <T extends Comparable<? super T>> void sort(List<T> list)`
 - `public static <T> void sort(List<T> list, Comparator<? super T> c)`
- La première trie suivant l'ordre naturel.
- La seconde permet de spécifier un comparator.

Tri de listes quelconques

Notes

Java, Programmation avancée

Nouveautés sur les collections avec Java 5 et Java 6

Version 2.2

- Les Queues
- Les ConcurrentMap
- NavigableSet et NavigableMap

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

L'interface Queue

Présentation

- `java.util.Queue<E>`
- Implémente `java.util.Collection<E>` et `java.util.Iterable<E>`.
- Structure de types FIFO ou LIFO.
- La récupération d'un élément de la structure le retire de la structure.
- Sous interface : `java.util.BlockingQueue<E>`

L'interface Queue

Notes

L'interface Queue

Méthodes

- `E element()` : récupère l'élément suivant de la structure, sans le retirer. Lève une exception si la structure est vide.
- `boolean offer(E o)` : Insert l'élément dans la structure (si possible).
- `E peek()` : récupère l'élément suivant de la structure, sans le retirer. Renvoie null si la structure est vide.
- `E poll()` : récupère l'élément suivant de la structure, et le retire. Renvoie null si la structure est vide.
- `E remove()` : récupère l'élément suivant de la structure, et le retire. Lève une exception si la structure est vide.

L'interface Queue

Notes

L'interface Queue

Implémentations

- `ArrayBlockingQueue`
- `ConcurrentLinkedQueue`
- `DelayQueue`
- `LinkedBlockingQueue`
- `LinkedList`
- `PriorityBlockingQueue`
- `PriorityQueue`
- `SynchronousQueue`

Queues manipulables par les deux extrémités (Java 6)

- Interface `java.util.Dequeue`
- Interface `java.util.BlockingDequeue`

L'interface Queue

Notes

L'interface ConcurrentMap

Présentation

- `java.util.ConcurrentMap<K, V>`
- Interface spécialisant `java.util.Map<K, V>` en y ajoutant 3 méthodes.
- Permet de gérer finement la présence, ou pas, des éléments dans la map.

Méthodes

- `V putIfAbsent(K key, V value)`
- `boolean remove(Object key, Object value)`
- `V replace(K key, V value)`
- `boolean replace(K key, V oldValue, V newValue)`

Implémentation

- `java.util.ConcurrentHashMap`

L'interface `ConcurrentMap`

Notes

L'interface NavigableSet

Présentation

- Spécialisation de l'interface `SortedSet<E>` offrant des méthodes de navigation.

Méthodes

- `E ceiling(E e)` : renvoie le plus petit élément supérieur ou égal à l'élément passé en paramètre.
- `Iterator<E> descendingIterator()`
- `NavigableSet<E> descendingSet()`
- `E floor(E e)` : renvoie le plus grand élément plus petit que ou égal à l'élément passé en paramètre.
- `SortedSet<E> headSet(E toElement)`
- `NavigableSet<E> headSet(E toElement, boolean inclusive)`
- `E higher(E e), E lower(E e)`
- `E pollFirst(), E pollLast()`
- `SortedSet<E> tailSet(E fromElement)`
- `SortedSet<E> subSet(E fromElement, E toElement)`
- ...

L'interface NavigableSet

Notes

L'interface NavigableSet

Implémentations

- `java.util.ConcurrentSkipListSet`
- `java.util.TreeSet`

L'interface `NavigableSet`

Notes

L'interface NavigableMap

Présentation

- `java.util.NavigableMap<K,V>`
- Spécialisation de `SortedMap<K,V>` offrant des méthodes de navigation supplémentaires.

Méthodes

- `Map.Entry<K,V> ceilingEntry(K key)`
- `K ceilingKey(K key)`
- `NavigableSet<K> descendingKeySet()`
- `NavigableMap<K,V> descendingMap()`
- `Map.Entry<K,V> floorEntry(K key)`
- `K floorKey(K key)`
- ...

Implémentations

- `java.util.ConcurrentSkipListMap`
- `java.util.TreeMap`

L'interface NavigableMap

Notes

Java, Programmation avancée

Les annotations en Java

Version 2.2

- Annotations
- Méta-annotations

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Qu'est-ce qu'une annotation ?

Indicateurs ajoutés au code

- Indicateurs traités par des outils (compilateur)
- Donnent lieu à une génération de code
- Mise en place d'injection de code

Qu'est-ce qu'une annotation ?

Notes

Annotations standard

Deprecated (applicable à tous)

- Désigne un élément comme obsolète

SuppressWarnings (applicable à tous sauf les annotations)

- Supprime les avertissements sur les données

Override (applicable aux méthodes)

- Vérification de la surcharge d'une méthode de la classe mère

Target (applicable aux annotations)

- Spécification des éléments auxquels cette annotation peut-être appliquée

Retention (applicable aux annotations)

- Spécification de la durée de conservation de l'annotation

Annotations standard

Notes

Annotations standard

Documented (applicable aux annotations)

- Inclusion de l'annotation dans la documentation des éléments annotés

Inherited (applicable aux annotations)

- Indication lorsque l'annotation est appliquée à une classe qu'elle est héritée automatiquement

Annotations standard

Notes

Annotations standard

L'annotation `Deprecated` sera utilisée pour tout élément dont l'utilisation est déconseillée

```
@Deprecated
void deprecatedMethod() {
}
```

L'annotation `SuppressWarnings` indique au compilateur de supprimer un type d'avertissement

```
Vector<Integer> v1 = new Vector<Integer>();
Object obj = (Object) v1;
Vector<Integer> v2 = (Vector<Integer>) obj;
```

```
AnnotationsExemples.java:15: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.Vector<java.lang.Integer>
    Vector<Integer> v2 = (Vector<Integer>) obj; //unchecked cast
                        ^
```

Avec la ligne suivante l'avertissement disparaît :

```
@SuppressWarnings("unchecked")
```

Annotations standard

Notes

Annotations standard

@Override

- L'annotation Override est utilisée pour demander au compilateur de s'assurer qu'une fonction en surcharge réellement une autre

```
package annotations;
```

```
public class ClasseMere {
```

```
    public void afficheValeur() {  
        System.out.println("Classe Mère");  
    }  
}
```

```
package annotations;
```

```
public class ClasseFille extends ClasseMere {
```

```
    @Override  
    public void afficheValeur(string s) {  
        System.out.println("Classe Filles");  
    }  
}
```

fonction différente de celle surchargée

```
ClasseFille.java:5: method does not override or implement a method from a supert  
ype  
    @Override  
    ^
```

Annotations standard

Notes

Les méta-annotations

La méta-annotation Target

- Est appliquée à une annotation, d'où une restriction sur les éléments auxquels s'applique cette annotation.
- Les types d'éléments de l'annotation Target sont

Table 3: Les types d'éléments de l'annotation Target

Type	Applicable à
ANNOTATION_TYPE	Déclaration de type d'annotations
PACKAGE	Packages
TYPE	Classes, énumérations et interfaces
METHOD	Méthodes
CONSTRUCTOR	Constructeurs
FIELD	Les champs (constants Enum y compris)
PARAMETER	Les paramètres de méthodes, les paramètres du constructeur
LOCAL_VARIABLE	Les variables locales

Les méta-annotations

Notes

Les méta-annotations

La méta-annotation Target

```
package annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
public @interface TestTarget{
    String societe() default "Leuville Objects";
}

@TestTarget(societe = "une société")
public void societe(){
    System.out.println("Test de l'annotation Target");
}
```

Les méta-annotations

Notes

Les méta-annotations

La méta-annotation Retention

- Permet de déterminer la durée de retenue de l'annotation
- Applicable en fonction des règles suivantes

Table 4: Règles de rétention

Règle	Description
SOURCE	Annotations non incluses dans les fichiers de classe (générées)
CLASS	Annotations incluses dans les fichiers classe. La machine virtuelle n'a pas besoin de les charger
RUNTIME	Annotations incluses dans les fichiers classe et chargé par la machine virtuelle

Les méta-annotations

Notes

Java, Programmation avancée

Utilisation avancée des annotations

Version 2.2

- Création d'annotations
- Réflexivité des annotations
- Utilisation de l'outil apt
- ...

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Création d'annotations

Utilisation de la déclaration `@interface`

```
public @interface FirstAnnotation {  
  
}
```

- Peut-être associé aux méta-annotation de Java (`@Retention ...`)

```
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
  
@Retention(RetentionPolicy.SOURCE)  
public @interface FirstAnnotation {  
  
}
```

- Peut contenir des attributs

```
import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
  
@Retention(RetentionPolicy.CLASS)  
public @interface FirstAnnotation {  
    String firstName();  
    String lastName();  
    int age() default 0;  
}
```


Création d'annotations

Notes :

Création d'annotations

- Limitation des types de paramètres
 - Les types primitifs (int, char, ...)
 - String
 - Class (optionnelement sous la forme Class<? extends UneClasse)
 - Les type énumérés
 - Des annotations
 - Un tableau d'un type indiqué précédemment
- Annotations utilisables pour
 - packages
 - classes (y compris les Enum) et interfaces (y compris les interfaces d'annotation)
 - méthodes et constructeurs
 - champs d'instances (y compris les constantes enum)
 - variables locales et variables de paramètres

Pour les variables locales traitement au niveau de la source

Création d'annotations

Notes :

Création d'annotations

- Il est impossible d'utiliser plusieurs fois une annotation pour un même élément

```
@FirstAnnotation(age=10, firstName="prénom", lastName="nom")
@FirstAnnotation(age=11, firstName="prénom1", lastName="nom1")
public void annotatedMethod() {
    // Du code
}
```

- Générera une erreur de compilation
- Possibilité d'utiliser une autre annotation qui contiendra celles-ci

```
public @interface AnnotationContainer {
    public FirstAnnotation[] values();
}

@AnnotationContainer(values={
    @FirstAnnotation(age=10, firstName="prénom", lastName="nom"),
    @FirstAnnotation(age=11, firstName="prénom1", lastName="nom1")})
public void annotatedMethod() {

}
```

Création d'annotations

Notes

Réflexivité des annotations

- Avec Java 5.0, l'API de réflexion `java.lang.reflect` a été étendu, pour pouvoir lire les annotations visibles à l'exécution.
- Les annotations sont extraites de l'interface `AnnotatedElement`, implémentée directement par la classe `Package` et `Class`, et indirectement par `Method`, `Constructor` et `Field`.
- Pour récupérer les annotations sur les paramètres, il existe la méthode `getParameterAnnotations()`.
- Il est également possible de récupérer la valeur d'un membre d'une annotations avec `getAnnotation(classeAnnotee.class).value()`.

Réflexivité des annotations

Notes

Réflexivité des annotations

Exemple

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface Instable {}

@Retention(RetentionPolicy.RUNTIME)
@interface EnCharge {
    String value();
}

public class TestAnnotation {

    @Instable
    @EnCharge("Toto")
    public double module1() { return 0.0; }

    @Instable
    @EnCharge("Titi")
    public double module2() { return 0.0; }

    public static void main(String[] args)
        throws NoSuchMethodException {
        Class classe = TestAnnotation.class;
        Method mthode = classe.getMethod("module1");
        boolean instable = mthode.isAnnotationPresent(Instable.class);
        String auteur = mthode.getAnnotation(EnCharge.class).value();
        System.out.println(auteur);
    }
}
```


Réflexivité des annotations

Notes

L'outil Annotation Processing Tool (apt)

Permet d'effectuer des traitements sur les annotations avant la compilation

- Permet de générer des messages (notes, warning et error)
- Permet de générer des fichiers (texte, binaire, sour et classe Java)

APT utilise les mêmes options de base que la commande javac et en plus :

- -s dir : Indique le répertoire de base où placer les fichiers générés
- -nocompile : Ne pas effectuer la compilation après le traitement des annotations
- -A[key[=val]] : Permet de passer des paramètres supplémentaires aux "fabriques"
- -factorypath path : Indique le ou les chemins de recherche des "fabriques", par défaut le classpath
- -factory classname : Indique le nom de la classe qui sert de fabrique

Basé sur l'utilisation de factory pour créer les éléments nécessaires

- AnnotationProcessorFactory : La fabrique qu'utilisera APT
- AnnotationProcessor : Créé par la fabrique puis utilisé pour le traitement des fichiers source
- Visitors : Permettent de visiter les déclarations/types d'un fichier source

L'outil Annotation Processing Tool (apt)

Notes

Java, Programmation avancée

Aperçu des nouveautés introduites dans Java 6

Version 2.2

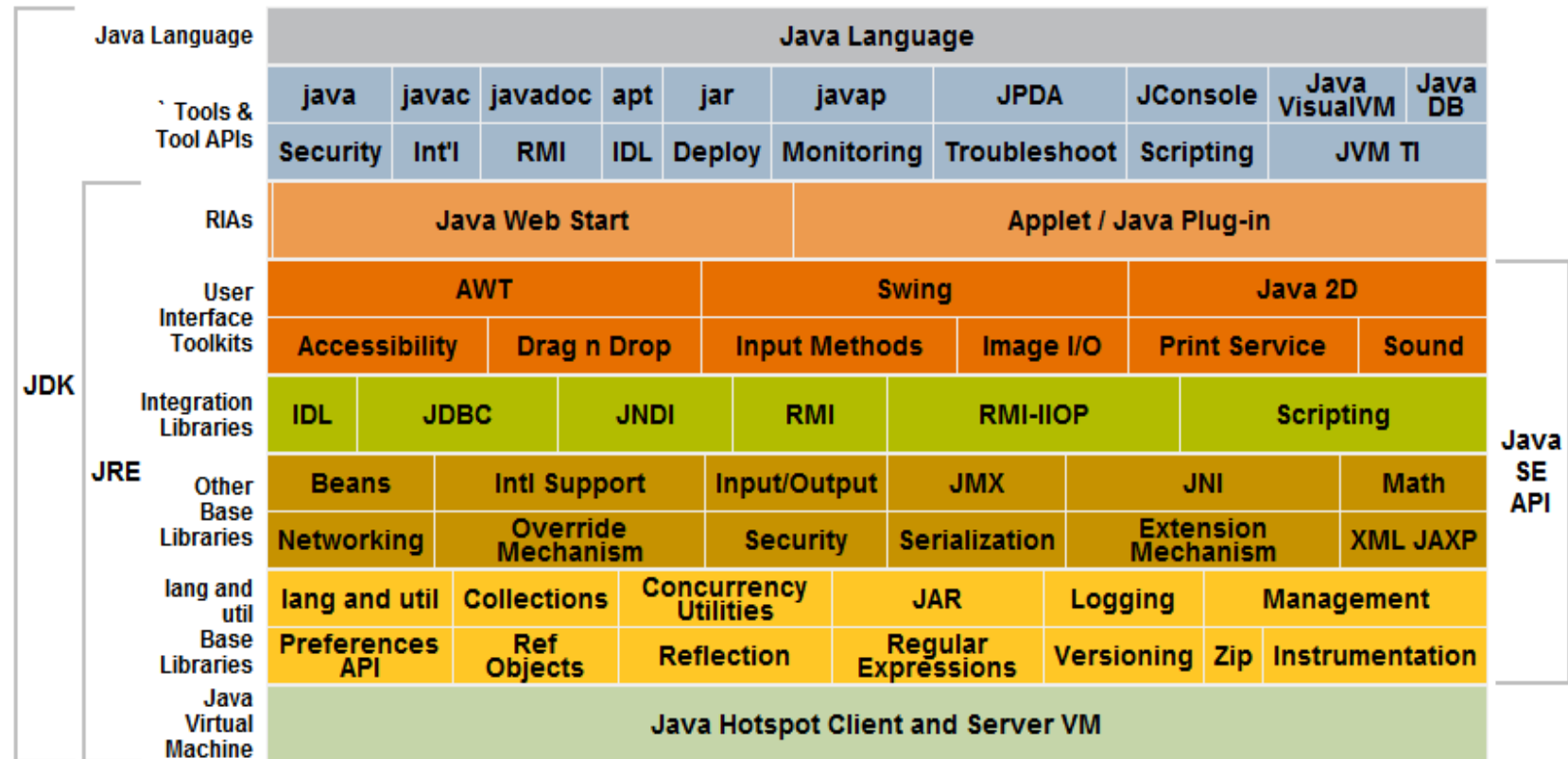
- Annotations
- Classpath
- ...

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Java 6

- Version du langage Java sortie en 2006
- Version introduisant de nouveaux concepts et de nouvelles syntaxes dans le langage.



- Liste de nouveautés : <http://docs.oracle.com/javase/6/docs/#NewFeature>

Java 6

Notes

Nouvelles annotations

- Elle se définissent comme les annotations communes (Common Annotation) et s'ajoutent aux annotations standard.

@Generated

- marque une classe, une méthode ou un champ comme étant généré par un outil.

@PostConstruct

- méthode exécutée après l'instanciation d'un objet.

@PreDestroy

- méthode de type callback appelée juste avant la suppression d'un objet.

@Resource

- permet de déclarer une référence vers une ressource requise pour une classe.

@Resources

- permet de déclarer plusieurs ressources.

Nouvelles annotations

Notes

Nouvelles annotations

@Generated

L'attribut 'value' est obligatoire et définit l'entité permettant la génération

```
@Generated(  
    value = "entite.de.generation",  
    comments = "commentaires",  
    date = "25/01/2012"  
)  
public void generatedCode()  
{}
```

Les autres attributs sont facultatifs et permet de donner une description.

Nouvelles annotations

Notes

Nouvelles annotations

@PostConstruct et **@PreDestroy**

Exemple d'utilisation

```
@Path("/taches")
public class TacheAccesseur {

    private EntityManager entityManager;

    @PostConstruct
    public void init() {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu1");
        entityManager = emf.createEntityManager();
    }

    @PreDestroy
    public void close() {
        entityManager.close();
    }
}
```

Nouvelles annotations

Notes

Nouvelles annotations

@Resource et **@Resources**

Plusieurs attributs sont définis

Attribut	Applicable à
name	Nom JNDI de la ressource
type	Type de la ressource
shareable	Booléen précisant si la ressource est partagée
authenticationType	Type d'authentification (Resource.AuthenticationType.CONTAINER ou Resource.AuthenticationType.APPLICATION)
description	Description de la ressource
mappedName	Nom associé au serveur utilisé

Nouvelles annotations

Notes

Classpath

Support du wildcard *

- Possibilité d'ajouter une collection d'archives jar d'un répertoire dans le Classpath à l'aide du wildcard *.
- Passage par ligne de commande ou variable d'environnement ou l'attribut du manifest du Jar.

```
java -cp "C:\chemin_archives\lib\*" testClasse.java
```

- Par précaution, ajouter toujours les guillemets car il peut y avoir des erreurs.
- Sur le système d'exploitation Linux, il peut y avoir des problèmes en fonction du shell/console utilisé. Par exemple avec tcsh ou csh, le wildcard * ne fonctionne pas. Par contre sous bash, la commande fonctionne.

Classpath

Notes

Java, Programmation avancée

Remote Method Invocation, les bases

Version 2.2

- Appel de méthodes distantes avec RMI
- Définition d'une interface de services distants
- Invocation de méthodes distantes et règles de passage de paramètres
- Applications clientes et serveurs
- Architecture-type d'une solution RMI

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

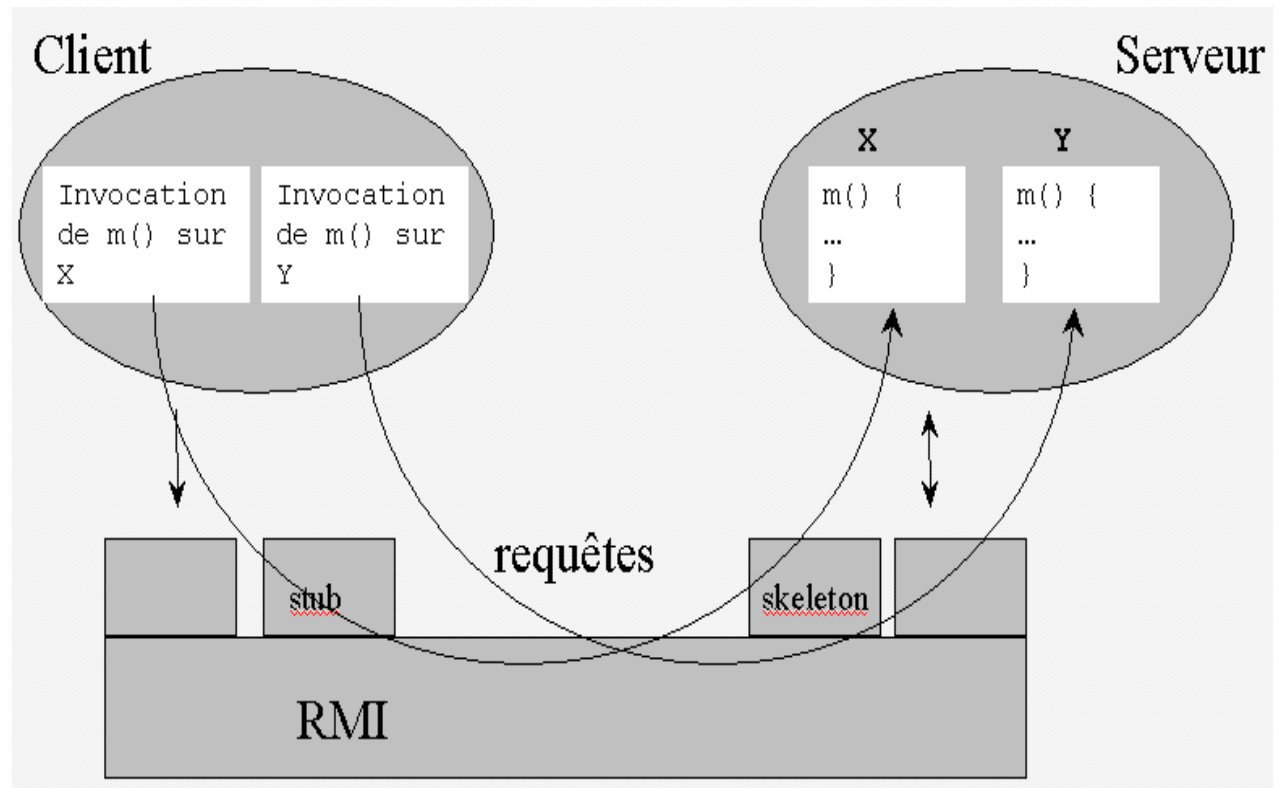
Présentation

Invocation de méthodes distantes

- A mi-chemin entre RPC du langage C et CORBA de l'OMG
- Fonctionne sur tout type de réseau IP

Caractéristiques

- Définition des services utilisables à distance à l'aide d'interfaces JAVA
- Transport d'objets par valeur et par référence
- Identification des objets serveur par nommage à syntaxe de type URL



Présentation

Invocation de méthodes distantes

Java permet de réaliser des applications client-serveur au dessus d'un réseau IP à l'aide de différentes techniques: Sockets UDP ou TCP, Programmation CORBA, Programmation RMI.

Cette dernière technique est propre à Java et permet d'invoquer une méthode sur un objet Java depuis une autre application Java, située sur la même machine ou sur tout autre machine connectée à la première par un réseau IP.

Caractéristiques

RMI est un mécanisme utilisable uniquement depuis le langage de programmation Java. Une implémentation RMI-IIOP permet de faire inter-opérer des objets RMI et des objets CORBA.

Les méthodes invocables à distance doivent être 'publiées' à l'aide d'interfaces Java et implémentées sur le serveur.

Elles seront alors accessibles depuis toute application cliente, à partir de représentants locaux des objets serveur. Ces représentants sont appelées des stubs.

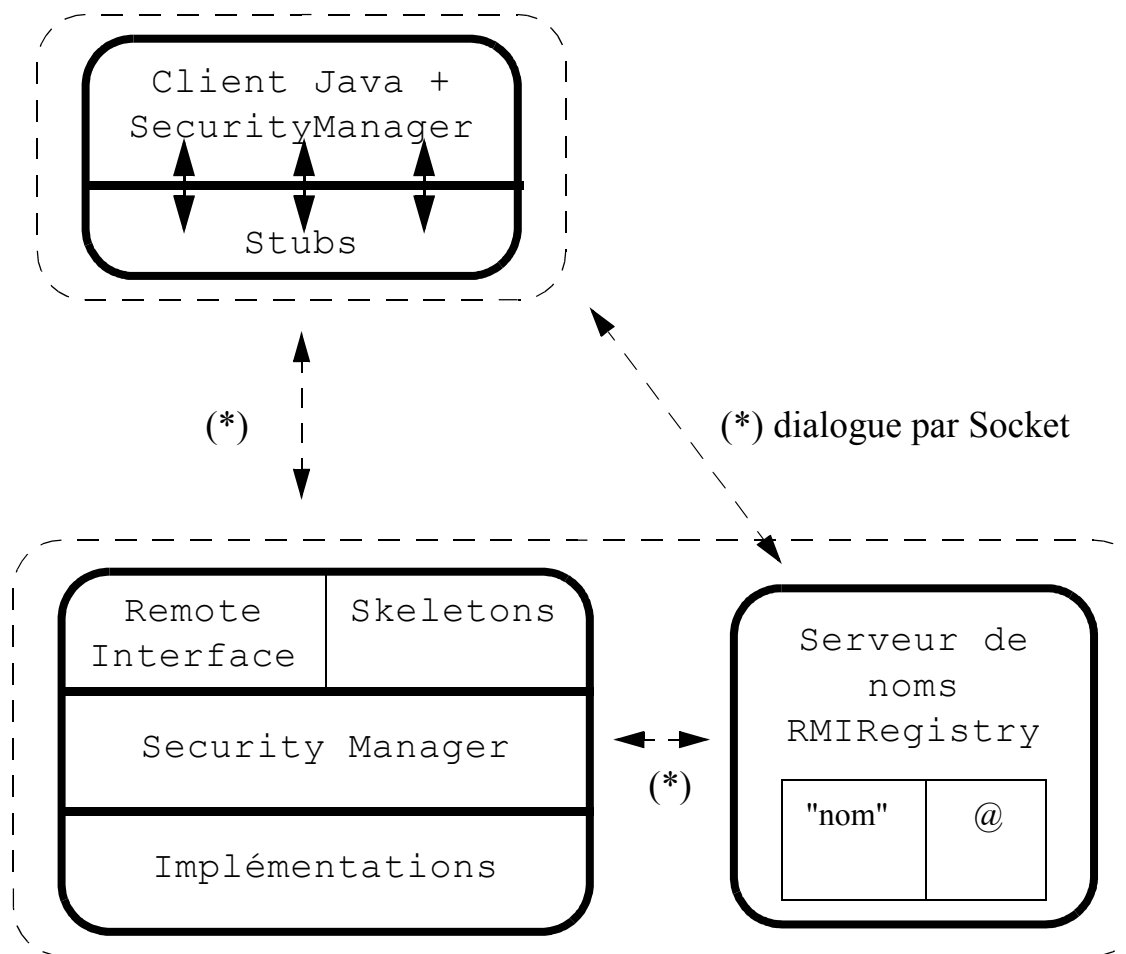
L'obtention d'un représentant local se fait par interrogation d'un serveur de noms qui répertorie l'ensemble des racines d'objets serveur.

Le transfert des informations sur le réseau est effectué suivant le protocole JRMP, propre à RMI. Ce protocole autorise le transport d'objet par référence et par valeur.

Architecture RMI

Schéma général

- Serveur: application autonome
- Client: application autonome ou applet
- Le serveur de noms fournit les objets initiaux aux clients
- Les communications s'effectuent par sockets



Architecture RMI

Schéma général

- Un serveur RMI est une application autonome Java dont les tâches sont les suivantes:
 - installation éventuelle d'une politique de sécurité spécifique pour protéger le serveur si celui-ci télécharge des classes, par exemple en provenance d'un client qui jouerait également le rôle d'un serveur
 - instanciation des objets serveurs
 - enregistrement des objets dans le serveur de noms
 - l'écoute des requêtes clients est assurée automatiquement par RMI
- Un client RMI peut être:
 - une application autonome
 - une applet
- rmiregistry est un serveur de noms minimal qui offre uniquement un service permettant de retrouver un objet serveur d'après son nom. Si l'on désire un service plus évolué (authentification par exemple), il est tout-à-fait possible de lui substituer un autre mécanisme (LDAP, serveur de noms CORBA, ...)
- Les communications entre client-serveur, client-rmiregistry et serveur-rmiregistry s'effectuent par Sockets. Les numéros de ports utilisés sont paramétrables. Il est également possible d'utiliser des sockets de types particuliers (SSL par exemple).

Etapes de réalisation d'une application RMI

Conception de l'architecture

- Services et relations entre objets serveur
- Nature des objets paramètres et valeurs de retour des services
- Politique de sécurité pour le client et le serveur

Code Java

- Définition des interfaces des classes d'objets serveur
- Implémentation des classes d'objets serveur conformes aux interfaces
- Réalisation du processus serveur
- Réalisation de l'application cliente

Mise en place

- Compilation des différents fichiers à l'aide du compilateur habituel
- Génération des stubs et skeletons avec le compilateur rmic
- Mise en oeuvre d'un démon httpd en fonction du mode de fonctionnement choisi

Etapes de réalisation d'une application RMI

Conception de l'architecture

Il convient tout d'abord de définir le modèle Objet des différentes classes d'objets serveur. Ce modèle devra intégrer l'ensemble des interfaces nécessaires et mettre en évidence les relations entre objets serveur.

Le choix de la nature de certains objets peut s'effectuer en considérant différents facteurs comme la fréquence d'utilisation, le volume des données, le nombre de services invoqués sur l'objet, ...

Code Java

Quelques règles d'écriture doivent être respectées quant à l'écriture des interfaces RMI:

- héritage de l'interface `java.rmi.Remote`
- ajout d'une clause `throws RemoteException` sur chaque méthode invocable à distance

Mise en place

Différents modes d'installation sont possibles:

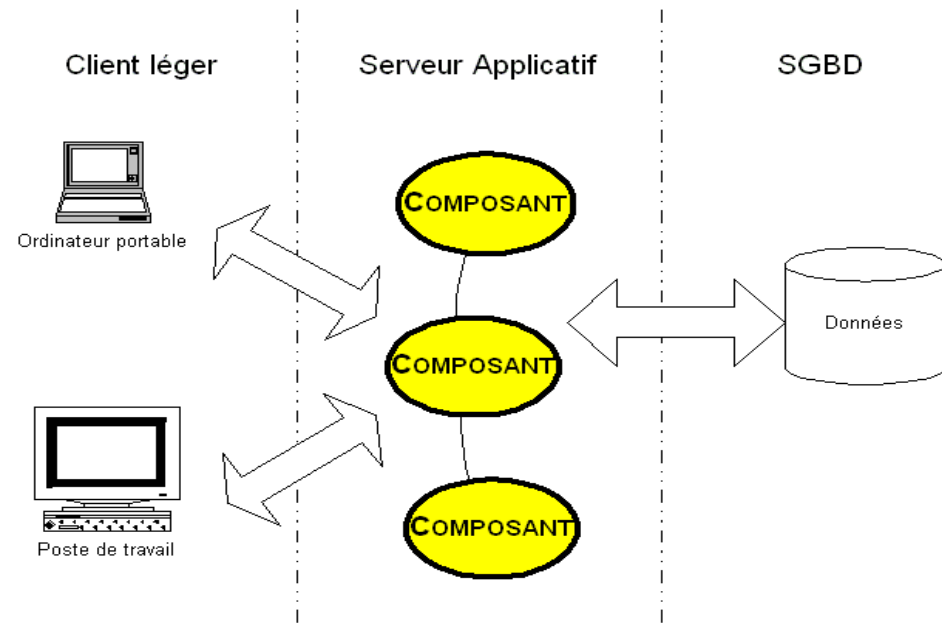
- client de type applet avec téléchargement de toutes les classes, y compris les classes RMI (stubs)
- client de type application autonome à installation fixe
- client de type application autonome possédant la capacité de se télécharger à partir d'une classe de chargement (bootstrap)

Les modes offrant un téléchargement de classes requièrent la mise en oeuvre d'un serveur WEB.

Déroulement sur un exemple

Mise en place d'un serveur bancaire

- Description de l'architecture
 - Un serveur central stocke les comptes des clients dans un SGBD
 - Accès aux opérations classiques de gestion de comptes depuis les postes des agences
- Services disponibles sur le serveur
 - ouverture d'un compte bancaire
 - accès à un compte à partir de son numéro de compte
 - obtention du solde d'un compte
 - dépôt et retrait d'une somme sur un compte
- Contraintes particulières
 - pouvoir accéder aux services à partir d'un simple navigateur
 - limiter les opérations de configuration du poste client



Déroulement sur un exemple

Mise en place d'un serveur bancaire

- Description de l'architecture

Une application en grandeur réelle comporterait trois niveaux:

- une base de données comportant les informations sur les comptes et les clients,
- un serveur applicatif proposant l'ensemble des services à des clients distants,
- un ensemble de clients distants légers.

Ici, nous allons nous focaliser sur le serveur applicatif intermédiaire et le client léger, qui communiquent à l'aide de RMI.

- Services disponibles sur le serveur

Le serveur offre deux niveaux de services distincts:

- la création de comptes bancaires,
- la réalisation d'opérations sur un compte particulier.

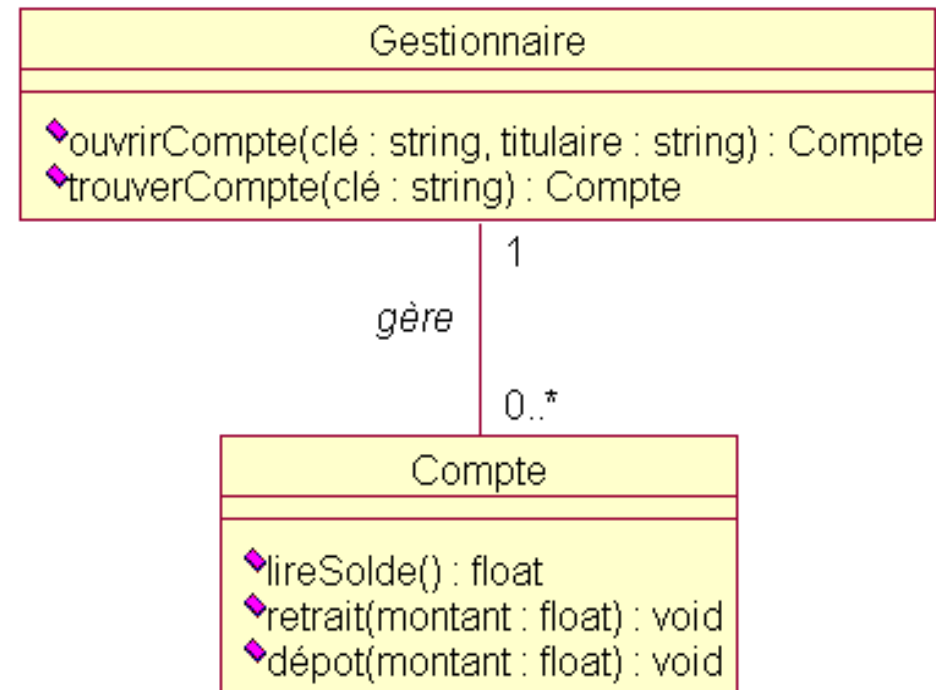
- Contraintes particulières

Un client de type applet permettra de bénéficier des fonctionnalités Java de téléchargement de classes, y compris les classes spécifiques à RMI (stubs).

Modélisation du problème

Deux niveaux de services distincts

- Gestion de la liste des comptes bancaires
 - Création
 - Modification
 - Destruction
- Opérations sur chaque compte bancaire
 - Obtention du solde
 - Retrait
 - Dépot



La solution RMI doit respecter la modélisation Objet.

Modélisation du problème

Deux niveaux de services distincts

La modélisation Objet de notre problème montre qu'il existe deux niveaux distincts de services:

- opérations sur la liste des comptes bancaires: ajout, retrait, accès ...
- opérations sur un compte: dépôt, solde, retrait.

Ces deux types de services vont être assurés par deux classes différentes.

La solution RMI doit respecter cette modélisation Objet afin d'en conserver tous les avantages:

- identification simple du rôle de chaque classe
- plus grande simplicité du code
- meilleure encapsulation.

Nous allons donc définir deux interfaces RMI correspondant aux classes d'objets que nous souhaitons manipuler à partir de l'application cliente.

Définition des interfaces des services

- Dériver l'interface *java.rmi.Remote*
- Prototyper chaque méthode, en utilisant des types de données Remote ou Serializable
- Ajouter une clause *throws RemoteException* sur chaque signature de méthode

Interface Compte avec gestion d'exceptions "métier"

```
import java.rmi.*;
public interface Compte extends Remote {

    public float lireSolde() throws RemoteException;
    public void dépot(float montant) throws RemoteException, TransactionException;
    public void retrait(float montant) throws RemoteException, TransactionException;
}
// avec

public class TransactionException
    extends RemoteException {}
```

Définition des interfaces des services

Interface Compte avec gestion d'exceptions "métier"

Chaque méthode est prototypée avec son type de retour, son nom et les types de ses paramètres éventuels. Nous y ajoutons une clause `throws RemoteException` imposée par RMI. Cette clause impose au code client de prendre en compte les éventuels problèmes qui peuvent survenir pendant l'invocation des méthodes:

- serveur ne répondant plus,
- rupture du réseau,
- ...

Les prototypes des méthodes `retrait()` et `solde()` sont enrichis pour signaler au code client qu'une exception `TransactionException` peut être levée.

Cette classe d'exception appartient au domaine des classes applicatives. Lorsqu'une exception de cette classe sera levée par le code serveur, le client pourra la traiter à l'aide d'un code `try {} catch() {}` classique.

Typage des valeurs de retour et paramètres

Types autorisés

- **Types primitifs**
 - boolean, int, float,
- Classes implémentant l'interface **java.io.Serializable**
 - de nombreuses classes du JDK: String, Vector,
 - classes utilisateur implémentant explicitement Serializable
- Interfaces dérivées de l'interface **java.rmi.Remote**
 - interfaces de description des méthodes invocables à distance par RMI

Deux modes de passage de paramètres

- **Par valeur:** types primitifs et java.io.Serializable
 - la totalité de la donnée est transférée sur le réseau
- **Par référence:** les sous-interfaces de java.rmi.Remote
 - seule la référence de la donnée est transférée sur le réseau
 - l'objet reste dans la machine virtuelle où il a été instancié

Typage des valeurs de retour et paramètres

Types autorisés

Les seuls types qui ne peuvent pas être utilisés en RMI pour spécifier les valeurs de retour ou paramètres sont constitués par les classes qui ne sont:

- ni Serializable,
- ni Remote.

Dans ce cas, l'erreur est détectée à l'exécution uniquement.

Deux modes de passage de paramètres

- Par valeur
Lorsque l'on utilise des objets dont la classe est Serializable, la totalité des objets est passée sur le réseau lors d'un passage de paramètre ou lors d'un retour d'invocation. Si l'objet est volumineux, cela peut augmenter le temps d'invocation des méthodes.
Il existe différents moyens d'agir sur le mécanisme de Serialization. Ils sont décrits par la suite.
- Par référence
Seule une référence permettant d'agir sur l'objet est transférée sur le réseau. Toutes les méthodes décrites dans l'interface de type Remote sont utilisables.

Contrôle de la serialization

Exemple

- Une classe est `Serializable` si tous ses attributs le sont
- Un attribut `static` n'est jamais serialisé
- `transient` permet d'éliminer un attribut
- Les références circulaires sont gérées par le mécanisme
- Les classes des objets ne sont pas sérialisées

```
import java.io.*;

public class Param implements Serializable
{
    // cet attribut sera Serializable si AutreClasse
    // implémente java.io.Serializable
    private AutreClasse unObjet;

    // type primitif toujours Serializable
    private int x;

    // les variables statiques ne sont jamais sérialisées
    static int y;

    // le mot-clé transient permet de ne pas sérialiser
    // un attribut
    transient private float taux;
}

class AutreClasse implements Serializable
{
    // référence circulaire possible
    Param monParam;

    String nom;
}
```

Contrôle de la serialization

Notes

Lorsqu'un objet est sérialisé, un graphe à plat contenant non seulement l'objet initial mais également tous ceux qu'ils référence est fabriqué. Ce graphe ne contient pas les classes des objets, mais uniquement leurs noms complets ainsi que des identifiants de version.

Ce graphe est transporté sur le réseau sous la forme d'un paquet d'octets. A son arrivée, le mécanisme crée une arborescence d'objets identiques aux objets originaux, à condition que toutes les classes des objets sérialisés soient présentes localement.

La serialization peut également être utilisée pour sauvegarder des objets dans un fichier. Il s'agit alors d'un mécanisme de persistance.

Un attribut marqué comme transient n'est pas sérialisé. Lors de la reconstruction de l'objet, la valeur par défaut du type de l'attribut est affectée à l'attribut:

- 0 pour les entiers,
- false pour boolean,
- 0.0 pour les flottants,
- null pour les références d'objets.

Interface Gestionnaire

Solution 1

- Transformer Compte en objet Serializable et définir Gestionnaire en objet Remote
- l'accès à un Compte entraîne un transfert par valeur sur le réseau
- le Compte est ensuite utilisable localement sur le poste client

Attention aux problèmes de synchronisation clients - serveur.

Solution 2

- Conserver Compte sous la forme d'un objet Remote et définir gestionnaire en objet Remote
- l'accès à un Compte entraîne seulement le transfert de sa référence sur le réseau
- l'utilisation de l'objet se fait obligatoirement par invocation de méthodes à travers le réseau

Solution retenue.

Interface gestionnaire

```
public interface Gestionnaire extends Remote {  
    public Compte ouvrirCompte(String clé, String titulaire) throws RemoteException;  
    public Compte trouverCompte(String clé) throws RemoteException;  
}
```

Interface Gestionnaire

Gestionnaire permet de construire des objets de type Compte, manipulables à distance. En cela, il remplace le constructeur que Java ne permet pas de spécifier dans une interface.

Cette interface permet également de retrouver un Compte à partir de sa clé, afin de pouvoir effectuer les opérations bancaires de dépôt, retrait et obtention de solde.

Solution 1

Cette solution est intéressante si l'on est amené à utiliser la quasi-totalité des possibilités de l'objet Compte sur le poste client.

Dans ce cas, le coût initial du transfert est compensé par la possibilité d'accéder en local à l'objet Compte.

Mais il ne faut pas perdre de vue que le Compte présent sur le poste local n'est qu'une copie de l'objet présent sur le serveur. En cas de modification sur plusieurs clients, des problèmes de synchronisation sont à prévoir.

Solution 2

Cette solution sera plus intéressante si l'on accède à un sous-ensemble des possibilités de l'objet Compte. En effet, chaque invocation de méthode doit transiter par le réseau.

Cette solution est retenue pour la suite de la présentation.

Implémentation des interfaces Remote

Services à fournir

- Une classe pour chaque interface de type Remote
- Une implémentation de toutes les méthodes spécifiées par les différentes interfaces Remote
- Convention d'usage
Pour toute interface **X**, nommer la classe d'implémentation **XImpl**

Compléments

- Constructeur(s)
- Sections critiques éventuelles (accès concurrents possibles)

Implémentation des interfaces Remote

Services à fournir

Une classe d'implémentation d'une interface de type Remote doit fournir une implémentation pour chaque méthode de l'interface.

Elle doit également être en mesure de traiter les différentes invocations de méthodes qui seront effectuées depuis les postes clients. Pour cela, il faut se mettre à l'écoute d'un port socket particulier, décoder les requêtes entrantes, les exécuter et renvoyer les résultats dans la socket.

Les requêtes RMI provenant d'un client sont automatiquement exécutées au sein d'un thread créé par RMI. Le concepteur devra donc éventuellement définir des sections critiques à l'aide de **synchronized**.

Implémentation de l'interface Compte

CompteImpl

- La classe implémente l'interface Compte
- Toutes les méthodes de l'interface doivent être implémentées
- Les implémentations n'ont pas à signaler la levée de RemoteException
- Il est possible d'ajouter d'autres méthodes:
 - publiques
 - privées ou protégées

```
import java.rmi.*;
import java.rmi.server.*;

public class CompteImpl implements Compte {
    private float solde = 0.0f;
    private String titulaire;

    // constructeur
    public CompteImpl(String titulaire) {
        this.titulaire = titulaire;
    }
    // implémentation de l'interface Compte
    public float lireSolde() {
        return solde;
    }
    public void dépot(float montant)
        throws TransactionException {
        if (montant <= 0)
            throw new TransactionException();
        solde += montant;
    }
    public void retrait(float montant)
        throws TransactionException {
        if ((montant <= 0) || (solde < montant))
            throw new TransactionException();
        solde -= montant;
    }
    // autres méthodes possibles ...
}
```


Implémentation de l'interface Compte

CompteImpl

La classe CompteImpl peut comporter:

- des définitions d'attributs statiques ou non, suivant tous les types et possédant toutes les visibilitées,
- toute autre méthode supplémentaire.

Le code applicatif est très peu perturbé par RMI.

Implémentation de l'interface Gestionnaire

GestionnaireImpl

- La méthode ouvrirCompte instancie un autre objet d'implémentation
- Cet objet est retourné suivant son type d'interface, seul connu côté client

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class GestionnaireImpl implements Gestionnaire {
    private Hashtable comptes;

    // constructeur
    public GestionnaireImpl() {
        comptes = new Hashtable();
    }

    // implémentation de l'interface Gestionnaire
    public Compte ouvrirCompte(String clé, String nom) {
        Compte c = new CompteImpl(nom);
        comptes.put (clé, c);
        return c;
    }

    public Compte trouverCompte(String clé) {
        Compte c = comptes.get(clé);
        return c;
    }

    // autres méthodes
    public String toString() {
        return comptes.toString();
    }
}
```

Implémentation de l'interface Gestionnaire

GestionnaireImpl

Cette implémentation met en évidence les différences de fonctionnement entre le serveur et le client:

- les objets d'implémentation peuvent instancier d'autres objets d'implémentation, directement à partir des classes d'implémentation,
- ces objets sont manipulés côté client uniquement à travers leurs interfaces Remote.

Réalisation du serveur de Banque

Tâches à effectuer

- Installer un SecurityManager
- Instancier un ou plusieurs objets d'implémentation
- Créer des souches qui seront utilisées par les clients
- Obtenir la référence du serveur de noms
- Exporter leurs souches dans le serveur de noms

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class Serveur {
    public static void main(String[] args) {
        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new SecurityManager());
            }
            Compte c = new CompteImpl("MoiMême");
            Compte stub = (Compte)
                UnicastRemoteObject.exportObject(c, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind("UnCompte", stub);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

UnicastRemoteObject.exportObject (objet Remote, numéro de port)

- Crée la classe et l'instance de Stub qui sera utilisée par le client
- Permet de définir le port socket utilisé pour la communication entre le client et le serveur, par l'emploi d'un second paramètre après la référence du stub.
 - 0 correspond à un numéro de port quelconque

Réalisation du serveur de Banque

Tâches à effectuer

- Le SecurityManager permet de protéger le serveur contre les opérations indésirables, et bien qu'il ne soit pas utile dans tous les cas de figure, son utilisation est fortement recommandée.
- L'application serveur doit instancier un ou plusieurs objets serveur qui permettront, à leur tour d'accéder à d'autres objets au travers des méthodes de leurs interfaces Remote. Ces objets sont des racines d'arborescences d'objets comparables à des racines de persistance par exemple.
- Le serveur de noms gère une liste de couples (nom, objet). Il permet à l'application cliente d'accéder aux objets serveur à partir de leur nom. La composition du nom est libre, mais un nom doit être lié à un unique objet.

La classe Registry comporte des méthodes d'accès aux fonctionnalités du serveur:

- bind() permet de nommer un objet,
- rebind() permet de nommer ou renommer un objet (elle est souvent utilisée de préférence à bind() car elle peut être utilisée plusieurs fois sans arrêter le serveur de noms),
- lookup() permet de rechercher un objet à partir de son nom depuis l'application cliente.

UnicastRemoteObject.exportObject (objet Remote, numéro de port)

Attention, il existe une méthode exportObject qui accepte en paramètre unique l'objet Remote et qui génère un Stub à condition que sa classe ait été préalablement créée par appel au compilateur rmic. Cette opération était indispensable avant l'arrivée de JAVA 5.

Le serveur de noms

Lancement manuel

- Lancer **rmiregistry** en ligne de commande

```
rmiregistry 1099
```

- Obtenir sa référence dans le code de l'application serveur

```
Registry registry = LocateRegistry.getRegistry(N° port);
```

Lancement effectué par l'application serveur

- Ne pas lancer le processus **rmiregistry** en ligne de commande
- Placer dans le code de l'application serveur:

```
Registry registry = LocateRegistry.createRegistry(N° port)
```

Ce mode de lancement interdit le téléchargement des classes Remote par le client.

Le serveur de noms

Notes

rmiregistry est un binaire présent parmi les binaires du JDK.

Réalisation de l'application cliente

Tâches à effectuer

- Installer un `SecurityManager` si le client est une application autonome
- Obtenir une référence sur le serveur de noms
- Obtenir une référence sur un objet Remote
- Invoquer des méthodes en tenant compte des exceptions de type `RemoteException`

```
import java.rmi.*;

public class Client {
    public static void main(String[] args) {
        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new SecurityManager());
            }
            Registry registry =
                LocateRegistry.getRegistry("localhost", 1099);
            Compte c = (Compte) registry.lookup ("UnCompte");

            c.dépot(1000.0f);
            c.retrait(500);
            System.out.println (c.lireSolde());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```


Réalisation de l'application cliente

Tâches à effectuer

La syntaxe à utiliser pour obtenir une référence distante est de la forme:

```
rmi://host:port/nom
```

- serveur est le nom ou l'adresse IP de la machine sur laquelle tourne le serveur de noms rmiregistry. Par raison de sécurité, cette machine est forcément la même que celle du processus serveur.
- port est le numéro de port utilisé par le serveur de noms rmiregistry (valeur par défaut: 1099).
- nom est la chaîne de caractères utilisée côté serveur pour enregistrer l'objet (premier paramètre de bind() ou rebind()).

Client de type applet

Exemple

- Le SecurityManager est déjà installé
- Si la sécurité n'est pas spécifiquement paramétrée, le serveur RMI doit se trouver sur le serveur WEB

```
import java.rmi.*;
import java.rmi.server.*;

public class Client extends java.applet.Applet {
    private Gestionnaire gref;

    public void init() {
        try {
            // obtention d'une référence Remote
            String host = getCodeBase().getHost();
            Registry registry =
                LocateRegistry.getRegistry(host, 1099);
            gref = (Gestionnaire) registry.lookup("MaBanque");
        } catch (Exception e) {
            // traitement d'erreur
        }
    }

    public void start() {
        try {
            // invocations de méthodes distantes
            Compte cref = gref.trouverCompte("12345");
            cref.dépot(1000);
            cref.retrait(2000);
            repaint();
        } catch (Exception e) {
            // traitement d'erreur
        }
    }
}
```

Client de type applet

Exemple

La référence du serveur RMI peut être obtenue par l'emploi de `getCodeBase().getHost()`:

- `getCodeBase()` retourne l'URL de la classe d'applet sur le serveur,
- `getHost()` retourne le nom d'hôte inclus dans cette URL.

Compilation des différents fichiers

Fichiers Java

- Interfaces de type Remote
- Classes d'implémentation et classes associées
- Applications serveur et client

```
javac -d . package/Interface.java  
javac -d . package/InterfaceImpl.java  
javac -d . Serveur.java  
javac -d . Client.java
```

Stubs

- Il n'est plus nécessaire d'employer le compilateur rmic depuis JAVA 5.
- Les classes des Stubs sont générées dynamiquement et stockées en mémoire.

Compilation des différents fichiers

Fichiers Java

La compilation des différents fichiers sources s'effectuent de façon habituelle, sans aucune option spécifique à RMI.

Types d'installations RMI et procédures de lancement

Client de type Applet

- RMI est totalement transparent pour le client
- Une procédure de lancement particulière doit être respectée pour rmiregistry et le serveur
- Le dialogue client-serveur peut franchir un firewall

Client application autonome non téléchargée

- Les applications sont installées manuellement

Client application autonome téléchargée

- Un module générique présent sur le poste client permet de télécharger l'application cliente
- Ce mode requiert un serveur HTTP sur le serveur
- La procédure de lancement des applicatifs serveur est identique à celle du mode applet

Types d'installations RMI et procédures de lancement

Client de type Applet

L'application cliente doit pouvoir télécharger les classes RMI (stubs notamment) depuis le serveur. Pour cela, ce dernier doit être paramétré spécifiquement lors de son lancement.

RMI est capable d'encapsuler ses requêtes au sein de requêtes HTTP pour franchir un firewall. Dans ce cas, un script CGI doit être utilisé côté serveur afin de transmettre les requêtes du serveur HTTP au serveur RMI. Ce script est fourni dans le répertoire bin de l'installation standard.

Client application autonome non téléchargée

Dans ce mode, les différentes classes sont installées sur chaque poste:

- toutes les classes à l'exception de celles du client sur le poste serveur,
- toutes les classes relatives à l'applicatif client sur le poste client.

Client application autonome téléchargée

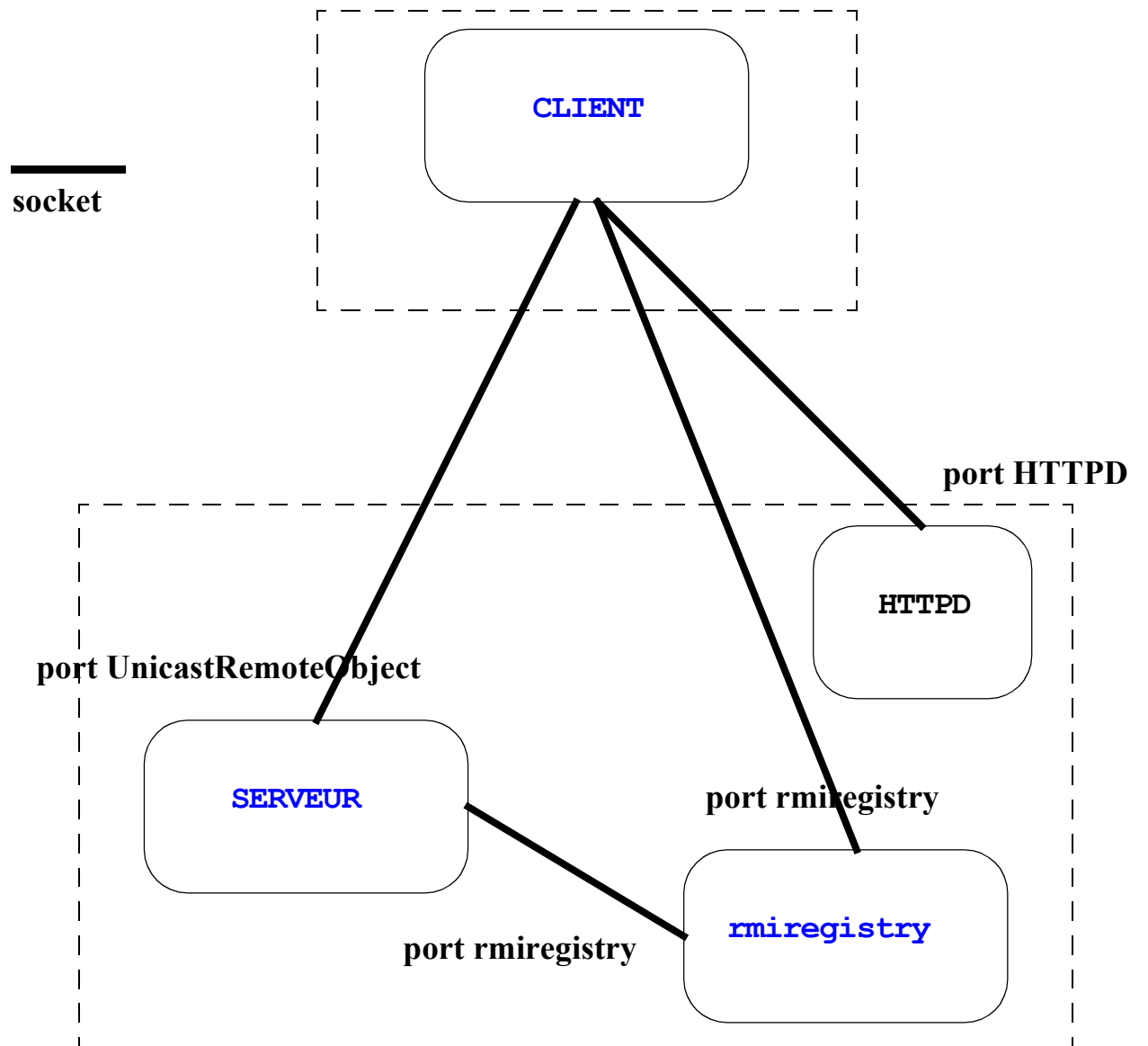
Ce mode permet de bénéficier du mécanisme de téléchargement des classes habituellement dévolu aux applets. Un applicatif de téléchargement est installé sur le poste client. Son rôle est de télécharger le véritable applicatif client depuis le serveur et de le lancer. Pour cela, il s'appuie sur un démon HTTP tournant sur le poste serveur.

Ce mode n'est pas étudié dans ce chapitre. Il est présenté de façon détaillée dans les spécifications RMI disponibles sur le site Javasoft (www.javasoft.com).

Echanges entre processus et rmiregistry

Schéma général

- port UnicastRemoteObject peut être déterminé par le programmeur pour chaque objet d'implémentation
- port rmiregistry fixé par défaut à 1099



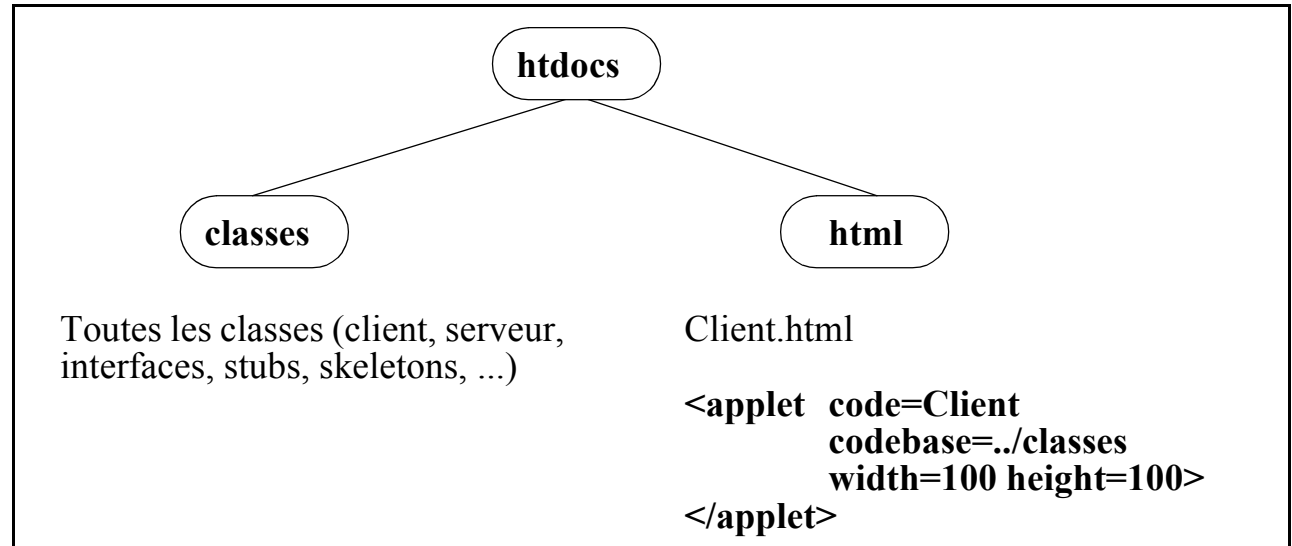
Echanges entre processus et rmiregistry

Schéma général

- Le serveur de noms rmiregistry écoute les requêtes sur un port qui est fixé par défaut à **1099** et qui peut être modifié au moment du lancement.
Ce serveur de noms est destiné à offrir au minimum une référence d'objet de type **Remote** au client distant. Les autres objets de type Remote peuvent être obtenus par le moyen d'envoi de messages à cet objet initial. rmiregistry a donc essentiellement une fonction permettant d'initier les échanges client-serveur (**bootstrap**).
- Chaque objet d'implémentation RMI peut utiliser un port socket spécifique dans son dialogue avec le client. Ce port peut être déterminé lors de l'appel de la méthode exportObject de UnicastRemoteObject.
- HTTPD n'est pas utile dans le cas où l'application RMI est de type autonome et ne fait appel à aucun téléchargement de classe. Ce point est étudié plus en détail par la suite.

Client RMI de type applet

Installation du serveur



Lancement du serveur

- Lancer **rmiregistry**

```
rmiregistry 1111
```

CLASSPATH ne doit pas permettre d'atteindre les stubs

- Lancer le **serveur** (hypothèse: httpd utilise le port 8080)

```
java -Djava.rmi.server.codebase=http://serveur:8080/classes/ Serveur
```

Client RMI de type applet

Installation du serveur

Le répertoire destiné à contenir les classes de l'application cliente (l'applet ici) doit notamment contenir:

- la classe de l'applet et toutes les classes qu'elle référence,
- les interfaces de type Remote.

En ce qui concerne le serveur RMI, nous aurons besoin:

- de la classe du serveur et de toutes les classes qu'il référence,
- des interfaces de type Remote et toutes les classes d'implémentation associées.

Lancement du serveur

- rmiregistry

Il faut veiller à ce que rmiregistry ne dispose pas d'un CLASSPATH permettant d'atteindre les stubs localement. Si c'était le cas, le client ne serait plus en mesure de télécharger ces classes.

- Le serveur

Le serveur doit connaître l'URL du répertoire contenant les stubs utilisés par le client. Cette URL est passée à rmiregistry, qui s'en servira pour permettre au client de télécharger les stubs.

Client RMI de type application autonome non téléchargée

Installation

- Serveur
 - L'application serveur: `Serveur.class`
 - Les interfaces Remote: `Compte.class` et `Gestionnaire.class`
 - Les classes d'implémentation: `CompteImpl.class` et `GestionnaireImpl.class`
- Client
 - L'application cliente: `Client.class`
 - Les interfaces Remote: `Compte.class` et `Gestionnaire.class`

Lancement

- Pour chaque application, CLASSPATH doit être définie de façon à permettre un chargement local de toutes les classes.

- Serveur

```
rmiregistry 1099  
java Serveur
```

- Client

```
java Client
```

Client RMI de type application autonome non téléchargée

Installation

Les applications doivent être installées de façon identique aux applications autonomes Java classiques.

Lancement

Il faut veiller à ce que la variable d'environnement CLASSPATH permette d'atteindre les classes dans tous les cas:

- rmiregistry doit pouvoir atteindre tous les stubs,
- le serveur doit pouvoir charger toutes ses classes, les interfaces Remote,
- le client doit pouvoir référencer sa classes et toutes celles qu'il utilise ainsi que toutes les interfaces Remote et stubs.

Java, Programmation avancée

Remote Method Invocation: aspects avancés

Version 2.2

- Réalisation d'un client RMI autonome téléchargeable
- Utilisation de sockets spécifiques avec les SocketFactory
- Mise en place de callbacks entre le serveur et les clients
- Activation des objets serveurs

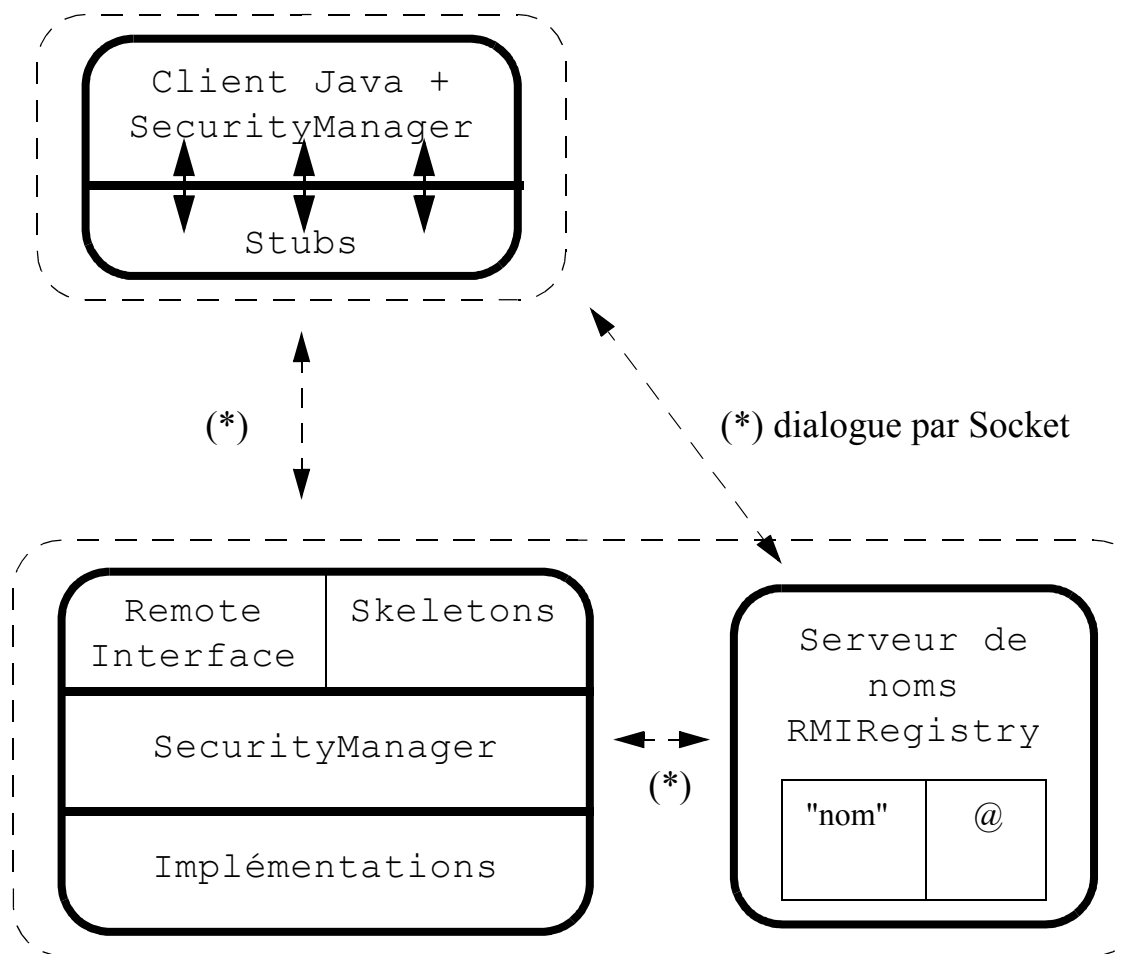
(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Architecture RMI

Schéma général

- Serveur: application autonome
- Client: application autonome ou applet
- Le serveur de noms fournit les objets initiaux aux clients
- Les communications s'effectuent par sockets



Architecture RMI

Schéma général

- Un serveur RMI est une application autonome Java dont les tâches sont les suivantes:
 - installation éventuelle d'une politique de sécurité spécifique pour protéger le serveur si celui-ci télécharge des classes, par exemple en provenance d'un client qui jouerait également le rôle d'un serveur
 - instanciation des objets serveurs
 - enregistrement des objets dans le serveur de noms
 - l'écoute des requêtes clients est assurée automatiquement par RMI
- Un client RMI peut être:
 - une application autonome
 - une applet
- rmiregistry est un serveur de noms minimal qui offre uniquement un service permettant de retrouver un objet serveur d'après son nom. Si l'on désire un service plus évolué (authentification par exemple), il est tout-à-fait possible de lui substituer un autre mécanisme (LDAP, serveur de noms CORBA, ...)
- Les communications entre client-serveur, client-rmiregistry et serveur-rmiregistry s'effectuent par Sockets. Les numéros de ports utilisés sont paramétrables. Il est également possible d'utiliser des sockets de types particuliers (SSL par exemple).

Types d'installations RMI et procédures de lancement

Client de type Applet

- RMI est totalement transparent pour le client
- Une procédure de lancement particulière doit être respectée pour rmiregistry et le serveur
- Le dialogue client-serveur peut franchir un firewall

Client application autonome non téléchargée

- Les applications sont installées manuellement

Client application autonome téléchargée

- Un module générique présent sur le poste client permet de télécharger l'application cliente
- Ce mode requiert un serveur HTTP sur le serveur
- La procédure de lancement des applicatifs serveur est identique à celle du mode applet

Types d'installations RMI et procédures de lancement

Client de type Applet

L'application cliente doit pouvoir télécharger les classes RMI (stubs notamment) depuis le serveur. Pour cela, ce dernier doit être paramétré spécifiquement lors de son lancement.

RMI est capable d'encapsuler ses requêtes au sein de requêtes HTTP pour franchir un firewall. Dans ce cas, un script CGI doit être utilisé côté serveur afin de transmettre les requêtes du serveur HTTP au serveur RMI. Ce script est fourni dans le répertoire bin de l'installation standard.

Client application autonome non téléchargée

Dans ce mode, les différentes classes sont installées sur chaque poste:

- toutes les classes à l'exception de celles du client sur le poste serveur,
- toutes les classes relatives à l'applicatif client sur le poste client.

Client application autonome téléchargée

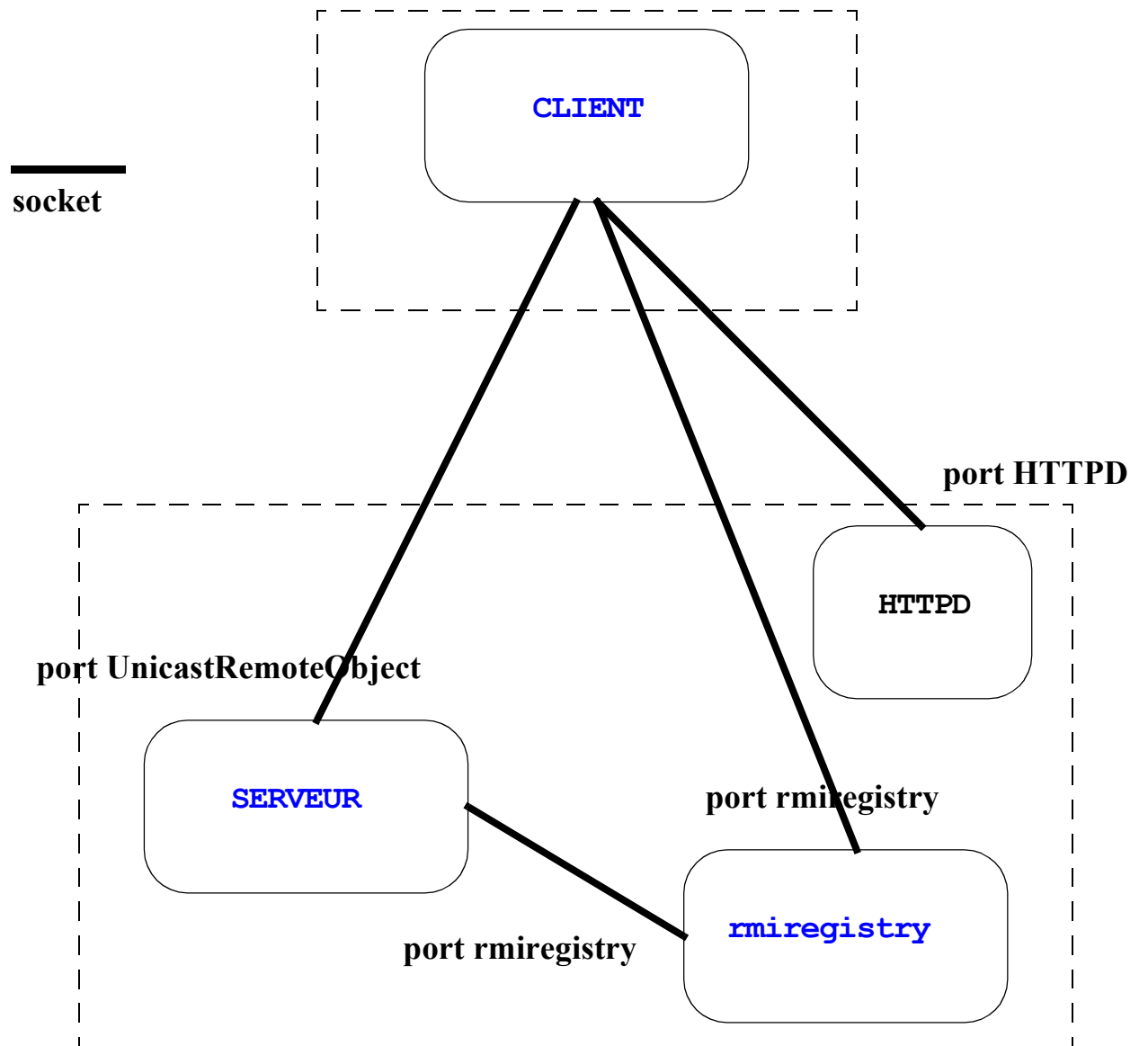
Ce mode permet de bénéficier du mécanisme de téléchargement des classes habituellement dévolu aux applets. Un applicatif de téléchargement est installé sur le poste client. Son rôle est de télécharger le véritable applicatif client depuis le serveur et de le lancer. Pour cela, il s'appuie sur un démon HTTP tournant sur le poste serveur.

Ce mode n'est pas étudié dans ce chapitre. Il est présenté de façon détaillée dans les spécifications RMI disponibles sur le site Javasoft (www.javasoft.com).

Echanges entre processus et rmiregistry

Schéma général

- port UnicastRemoteObject peut être déterminé par le programmeur pour chaque objet d'implémentation
- port rmiregistry fixé par défaut à 1099



Echanges entre processus et rmiregistry

Schéma général

- Le serveur de noms rmiregistry écoute les requêtes sur un port qui est fixé par défaut à **1099** et qui peut être modifié au moment du lancement.
Ce serveur de noms est destiné à offrir au minimum une référence d'objet de type **Remote** au client distant. Les autres objets de type Remote peuvent être obtenus par le moyen d'envoi de messages à cet objet initial. rmiregistry a donc essentiellement une fonction permettant d'initier les échanges client-serveur (**bootstrap**).
- Chaque objet d'implémentation RMI peut utiliser un port socket spécifique dans son dialogue avec le client. Ce port peut être déterminé lors de l'appel de la méthode exportObject de UnicastRemoteObject.
- HTTPD n'est pas utile dans le cas où l'application RMI est de type autonome et ne fait appel à aucun téléchargement de classe. Ce point est étudié plus en détail par la suite.

Client RMI de type application autonome téléchargeable

Principes de fonctionnement

- Une application cliente générique télécharge la véritable application cliente depuis le serveur
- Ce processus requiert un démon **httpd** sur le serveur
- Code du client de chargement

```
import java.rmi.*;
import java.rmi.server.*;

public class Bootstrap {
    public static void main(String[] args)
    {
        try {
            // SecurityManager
            System.setSecurityManager(
                new SecurityManager());
            // téléchargement de la classe du client
            Class c = RMIClassLoader.loadClass("Client");
            // instantiation du client
            Runnable r = (Runnable)c.newInstance();
            // lancement du 'véritable' client
            r.run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- Lancement du client

```
java -Djava.rmi.server.codebase=http://serveur:8080/classes/ Boot
```

Client RMI de type application autonome téléchargeable

Principes de fonctionnement

- RMIClassLoader offre la possibilité de télécharger une classe à l'aide HTTP
- Cette classe est manipulable à travers le type `java.lang.Class`, qui offre, outre les services d'auto-description du package `java.lang.reflect`, la capacité d'instanciation
- L'appel à `newInstance()` permet d'instancier la classe téléchargée, à condition que celle-ci contienne un constructeur public sans paramètre
- L'instance obtenue est manipulée à travers le type `java.lang.Runnable`, qui n'a pas à être téléchargé car toujours présent dans la machine virtuelle locale. Cela impose que la classe du 'véritable client' implémente l'interface `Runnable`. Ce choix a été fait afin de faciliter l'écriture de la classe Bootstrap
- Lorsque l'application Bootstrap est lancée, il faut définir la propriété **`java.rmi.server.codebase`** afin de spécifier l'URL permettant d'accéder au code du véritable client

Client RMI de type application autonome téléchargeable

Lancement du serveur

- Processus identique au lancement du serveur RMI avec des clients de type applet
- Lancer **httpd** (par exemple sur le port 8080)
- Lancer **rmiregistry**

```
rmiregistry 1111
```

CLASSPATH ne doit pas permettre d'atteindre les stubs

- Lancer le **serveur** avec la propriété **java.rmi.server.codebase**

```
java -Djava.rmi.server.codebase=http://serveur:8080/classes/ Serveur
```

Ne pas oublier le / final dans la définition de java.rmi.server.codebase

Client RMI de type application autonome téléchargeable

Lancement du serveur

- httpd

Le démon httpd est utilisé par le client de type Bootstrap pour télécharger les classes. Cette opération est effectuée par RMIClassLoader.

- rmiregistry

Le serveur de noms rmiregistry est une application Java qui permet au client distant d'obtenir au moins la référence d'un objet Remote initial.

Il faut veiller à ce que rmiregistry ne dispose pas d'un CLASSPATH permettant d'atteindre les stubs localement. Si c'était le cas, le client ne serait plus en mesure de télécharger ces classes.

- Le serveur

Le serveur doit connaître l'URL du répertoire contenant les stubs utilisés par le client. Cette URL est passée à rmiregistry, qui s'en servira pour permettre au client de télécharger les stubs.

Sockets spécifiques

Les interfaces `SocketFactory`

- Une paire de sockets factory serveur/client par objet d'implémentation
- Il faut paramétrer la sécurité pour pouvoir créer des sockets spécifiques

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

public class MiseAJourImpl implements MiseAJour { // ... }
public class ServSF implements RMIServerSocketFactory {
    public ServerSocket createServerSocket(int port) { // ... }
}
public class ClientSF implements RMIClientSocketFactory {
    public Socket createSocket(String host, int port) { // ... }
}
public class Serveur {
    public static void main(String[] args) {
        try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new SecurityManager());
            }
            MiseAJour m = new MiseAJourImpl();
            MiseAJour stub = (MiseAJour)
                UnicastRemoteObject.exportObject(
                    c, 0, new ClientSF(), new ServerSF()
                );
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
grant {
    permission java.lang.RuntimePermission "setFactory";
};
```

Sockets spécifiques

Les interfaces SocketFactory

Ces interfaces permettent de définir des objets responsables de la création des Sockets utilisées par RMI. Chaque objet d'implémentation RMI peut être utilisé avec une paire de SocketFactory différente.

La création de sockets spécifiques est une opération assez complexe, qui n'est pas abordée par ce cours.

Ce mécanisme permet d'agir aisément sur les couches de communication de RMI. Il est par exemple possible de s'appuyer sur des sockets de type SSL plutôt que sur des sockets standards. Plusieurs éditeurs proposent des solutions prêtes à l'emploi:

- www.phaos.com
- www.baltimore.ie
- www.jcp.co.uk

Réalisation de callbacks RMI

Définition d'une callback

- Méthode appelée par le serveur sur un client
- Avec RMI, l'objet client doit également être de type **java.rmi.Remote**

Exemple: application bancaire

- Un service permet à des clients distants d'obtenir en temps-réel les informations concernant les opérations passées sur le compte
- Interface du service Serveur MiseAJour

```
import java.rmi.*;
public interface MiseAJour extends Remote {
    public void connecter(Client c) throws RemoteException;
    public void deconnecter(Client c)
        throws RemoteException;
}
```

- Interface du Client avec méthode informer() de type callback

```
import java.rmi.*;
public interface Client extends Remote {
    public void informer(String operation)
        throws RemoteException;
}
```

Réalisation de callbacks RMI

Définition d'une callback

Une callback est une méthode que le serveur invoque sur un client, généralement à la suite d'une action de ce dernier sur le serveur.

Avec RMI

Le serveur doit connaître l'interface des méthodes accessibles à distance du client. Il faut donc que celles-ci soient décrites par une interface de type `java.rmi.Remote`.

Exemple

Une extension du programme Bancaire permet à des utilisateurs distants d'obtenir en temps-réel des informations sur les opérations concernant leurs comptes en banque. Le service fonctionne à partir du moment où l'utilisateur s'est préalablement connecté. Il reçoit alors toutes les mises-à-jour, et ce jusqu'à sa déconnexion.

La méthode `informe()` est invoquée par le serveur dès qu'une opération est effectuée.

Sur le poste client doit se trouver la classe d'implémentation des services de l'interface `Client`. Cette classe **ClientImpl** héritera de `UnicastRemoteObject` ou redéfinira les méthodes d'écoute des requêtes RMI.

Réalisation de callbacks RMI

Enregistrements et notifications

- Ne pas oublier de prendre en charge:
 - `UnicastRemoteObject.exportObject`
 - `UnicastRemoteObject.unexportObject`

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class MiseAJourImpl implements MiseAJour {
    private ArrayList connectes;

    public MiseAJourImpl(String nom)
        throws RemoteException {
        connectes = new ArrayList()
    }
    public synchronized void connecter(Client c) {
        connectes.add(c);
        notifier(c, "Bienvenue");
    }
    public synchronized void deconnecter(Client c) {
        connectes.remove(connectes.lastIndexOf(c));
        notifier(c, "Au revoir");
    }
    // invocation de la callback
    public void notifier (Client client, String info)
        try {
            client.informer(info);
        } catch (RemoteException e) {
            try {
                deconnecter(client);
            } catch (RemoteException e2) {
            }
        }
    }
}
```

Réalisation de callbacks RMI

Invocation des callbacks

La callback est invoquée en suivant les mêmes règles que pour l'invocation de méthodes distantes classiques:

- passage de paramètres de type **java.io.Serializable** ou **java.rmi.Remote** uniquement,
- gestion des exceptions de type **java.rmi.RemoteException**

Le concepteur doit veiller à l'intégrité des données manipulées en utilisant éventuellement des sections critiques définies à l'aide de **synchronized**.

Politique d'activation des objets serveurs

- **java.rmi.server.UnicastRemoteObject** permet de définir des objets serveurs tout le temps actifs
- **java.rmi.activation.Activatable** permet de définir des objets serveurs activables sur demande

Caractéristiques

- les objets Activable sont chargés en mémoire uniquement lorsqu'ils reçoivent une requête
- un démon **rmid** est chargé de lancer des machines virtuelles Java si nécessaire
- il faut préparer un ensemble d'informations qui seront utilisées pour paramétrer l'objet au moment de son activation
- la politique d'activation dépend uniquement du serveur
 - aucune modification du code client
 - pas de modification des interfaces Remote

Rendre un objet serveur activable

- Hériter de `java.rmi.activation.Activatable`
- Définir un constructeur pour enregistrer l'objet dans le système d'activation
- Créer une classe de paramétrage de l'activation (setup)

Politique d'activation des objets serveurs

Caractéristiques

JAVA 2 introduit la possibilité d'instancier les objets serveurs RMI uniquement à la demande, c'est-à-dire au moment où une requête RMI leur est envoyée.

C'est un programme particulier, **rmid**, qui est chargé d'activer les objets d'implémentation en cas de besoin. Ce démon tourne au sein d'une machine virtuelle Java et est capable de lancer d'autres machines virtuelles.

Rendre un objet serveur activable

La classe de paramétrage de l'activation devra notamment effectuer les tâches suivantes:

- installer un SecurityManager
- définir les paramètres d'activation
 - groupes d'objets activables attachés à différentes machines virtuelles d'exécution,
 - informations permettant d'instancier les objets serveurs
- enregistrer les informations dans rmid
- nommer l'objet serveur

Activation: adaptation de la classe GestionnaireImpl

Exemple

- Fournir un constructeur à deux arguments pour enregistrer l'objet serveur dans le système d'activation
- Le second paramètre contient les données d'initialisation sous la forme d'un objet sérialisé

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;

public class ActivatableGestionnaire extends Activatable
                                   implements Gestionnaire {
    private Hashtable comptes;

    public ActivatableGestionnaire(
        ActivationID id, MarshaledObject data)
        throws RemoteException {
        super (id, 0);
        comptes = new Hashtable();
    }

    public Compte ouvrirCompte(String clé, String nom)
        throws RemoteException {
        Compte c = new CompteImpl(nom);
        comptes.put (clé, c);
        return c;
    }

    public Compte trouverCompte(String clé)
        throws RemoteException {
        Compte c = comptes.get(clé);
        return c;
    }

    // autres méthodes
    public String toString() {
        return comptes.toString();
    }
}
```

Activation: adaptation de la classe GestionnaireImpl

```
public ActivatableGestionnaire(  
    ActivationID id, MarshaledObject data)  
    throws RemoteException {  
    super (id, 0);  
    ...  
}
```

- id est un identificateur unique de l'objet,
- data contient les données d'initialisation de l'objet, définies au moyen de la classe de type Setup étudiée par la suite,
- l'appel à `super (id, 0)` permet d'enregistrer l'objet d'implémentation dans le système d'activation. Le paramètre 0 indique qu'un port anonyme sera utilisé. Il est également possible d'utiliser un numéro de port explicite. Cela peut-être utile dans un cadre Internet, où l'on souhaiterait paramétrer le filtrage d'un firewall.

De façon identique à ce qui existe dans `UnicastRemoteObject`, la classe `Activatable` offre des constructeurs permettant de spécifier des fabriques de sockets spécifiques.

Classe de paramétrage de l'activation (setup)

Structure générale

- Installation d'un SecurityManager

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;

public class Setup {
    public static void main (String[] args) {
        try {
            System.setSecurityManager(new SecurityManager());
            //
            // code de paramétrage de l'activation
            // ...
            //
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Classe de paramétrage de l'activation (setup)

Structure générale

La classe de paramétrage de l'activation remplace la classe de lancement du serveur que l'on utilise avec une solution de type `UnicastRemoteObject`.

Ses tâches sont les suivantes:

- installation d'un `SecurityManager`,
- création d'un `ActivationGroup`,
- création d'un `ActivationDesc`,
- enregistrement auprès de `rmid`,
- nommage du stub créé par la précédente opération.

Le code présenté par la suite est à insérer à la place des commentaires du bloc **`try { } catch () {}`**.

Classe de paramétrage de l'activation (setup)

ActivationGroup

- Regroupement d'objets serveurs activables
- Une machine virtuelle d'exécution est attachée à un groupe
- Contrôle l'activation/désactivation des objets serveurs

```
Properties props = new Properties();
props.put ("java.security.policy",
          "/home/moi/mypolicy");
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc grDesc =
    new ActivationGroupDesc(props, ace);
ActivationGroupID id =
    ActivationGroup.getSystem().registerGroup(grDesc);
ActivationGroup.createGroup(id, grDesc, 0);
```

ActivationDesc

- Indique l'endroit où réside la classe de l'objet activable
- Permet de définir les paramètres qui seront passés au constructeur de l'objet serveur

```
// URL de la classe de l'objet à activer
String location = "file:/home/moi/mesclasses/";
// paramètres
MarshaledObject par = new MarshaledObject("RMI Bank");
//
ActivationDesc desc = new ActivationDesc (
    "ActivatableGestionnaire", location, par);
```

Classe de paramétrage de l'activation (setup)

ActivationGroup

Pour définir un ActivationGroup, il faut disposer du privilège `java.lang.RuntimePermission` intitulé `"setFactory"`.

ActivationDesc

Ce descripteur permet au groupe d'avoir toutes les informations nécessaires à l'activation de l'objet:

- nom de la classe d'objet serveur (attention le nom complet de la classe est requis si celle-ci est membre d'un package),
- localisation de la classe compilée sous la forme d'une chaîne à la syntaxe de type URL,
- paramètres à passer éventuellement au constructeur de l'objet serveur,
- groupe de rattachement.

Classe de paramétrage de l'activation (setup)

Enregistrement dans le système d'activation

- rmid doit être actif avant l'exécution de ce code
- le nommage est effectué sur le bouchon et non pas sur l'objet d'implémentation, qui n'existe pas encore

```
//  
// enregistrement dans rmid  
//  
Gestionnaire gest =  
    (Gestionnaire)Activatable.register(desc);  
//  
// nommage du stub  
//  
Naming.rebind ("MaBanque", gest);  
//  
// fin de setup  
//  
System.exit(0);
```


Classe de paramétrage de l'activation (setup)

Enregistrement dans le système d'activation

- Enregistrement dans rmid

La méthode d'enregistrement du descripteur d'activation retourne une instance dont le type est celui de l'interface Remote. Il faut que rmid soit actif au moment de l'exécution de ce code.

- Nommage du stub

L'objet d'implémentation n'est plus nommé directement car il n'existe pas encore. C'est le bouchon destiné au poste client qui est nommé.

- Fin de setup

Cette classe sert uniquement au paramétrage de l'activation et n'est pas responsable de l'écoute des requêtes des clients. Il est donc possible de quitter l'application.

Lancement côté serveur

Dans cet ordre

- rmiregistry

- rmid

```
rmid -J-Djava.security.policy=mysecurity.policy
```

- la classe Setup

Lancement côté serveur

Notes

Il faut penser à lancer `rmiregistry` et `rmid` avant la classe de paramétrage de l'activation.

Suivant le mode d'installation et de lancement désiré, on se référera à la procédure concernant les serveurs de type `UnicastRemoteObject`.

Java, Programmation avancée

JMS (Java Messaging Service)

Version 2.2

- Définition et utilité de JMS
- Exemple d'utilisation

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Service de messagerie

Définition

- API Java permettant aux applications de créer / envoyer / recevoir / lire des messages.
- Permet une communication faiblement couplée entre applications

Propriétés

- Standard
- Synchrone / asynchrone
- Fiable

Service de messagerie

Définition

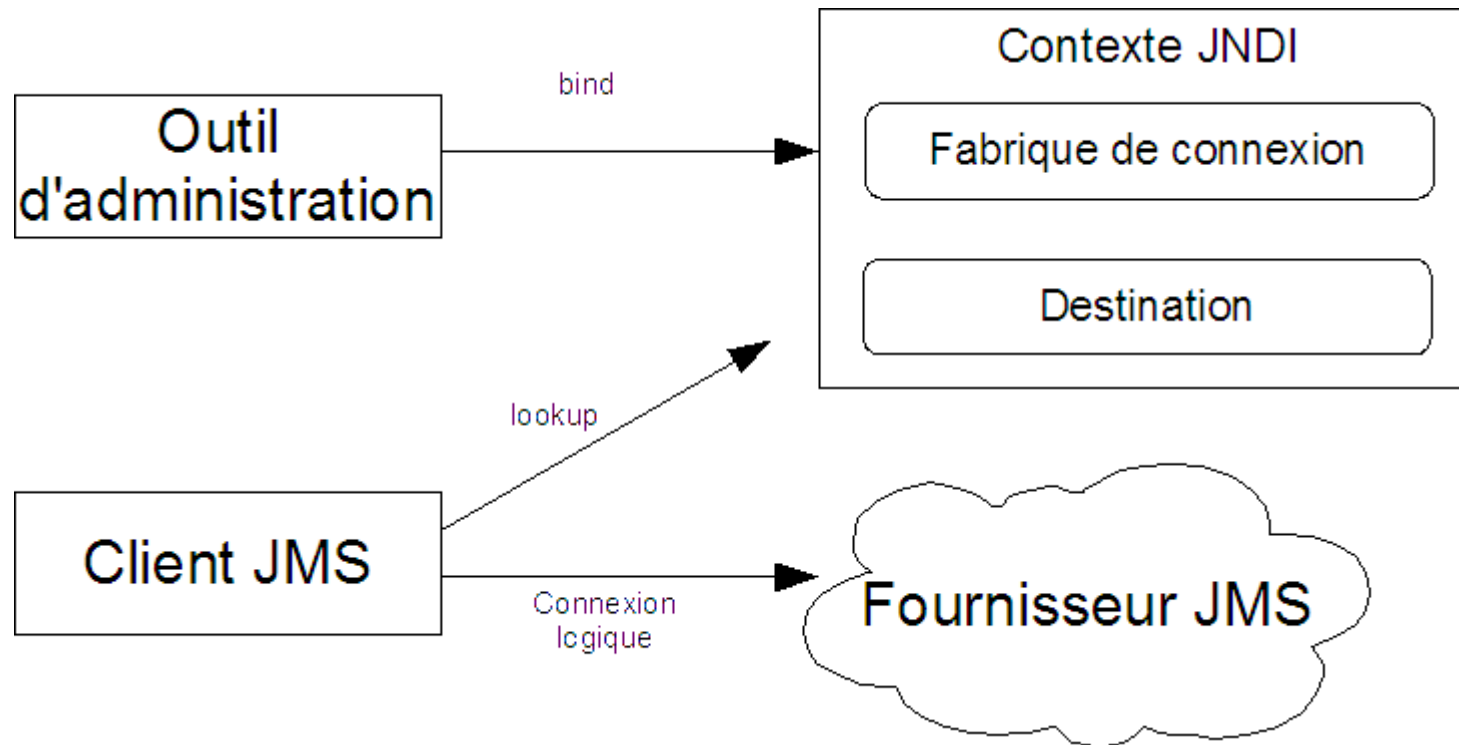
- JMS décrit un ensemble d'interfaces permettant de manipuler des messages et de les échanger d'une application à une autre.
- Les applications communiquent entre elles via ce système de messagerie et n'ont donc pas besoin de se "voir" les unes les autres.

Propriétés

- Les implémentations de JMS sont nombreuses et interchangeables de façon transparente. Parmi les providers :
 - IBM WebSphere MQ
 - BEA WebLogic Server JMS
 - JBoss Messaging
 - OpenJMS
- JMS supporte un fonctionnement en mode synchrone (attente et récupération des messages) ou asynchrone (modèle Listener).
- L'API JMS peut contrôler que tout message est traité une et une seule fois. Le niveau de contrôle peut être réduit pour les applications acceptant la perte ou la duplication de messages.

Principe et acteurs

- Fournisseur JMS (JMS Provider) : implémentation de l'API JMS
- Client JMS : produit et/ou consomme des messages
- Message : objet véhiculant des informations entre clients JMS
- Objets administrés : les fabriques de connexion (Connection Factory) et les destinations sont configurées et placées dans un contexte JNDI par un administrateur



Principe et acteurs

Notes

Les fabriques de connexion et les destinations sont placées dans un espace de nommage JNDI par un administrateur. Un client JMS peut alors récupérer une référence sur ces objets (look up) et établir une connexion logique via le fournisseur JMS.

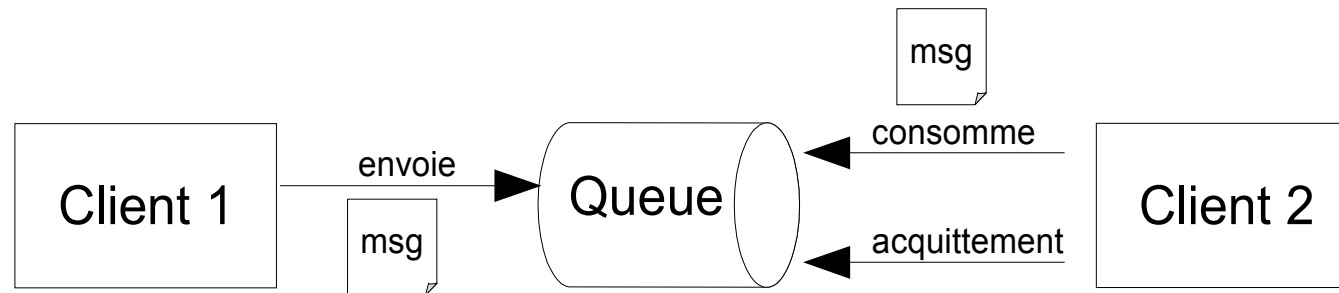
Domaines de messagerie

Deux domaines de messagerie supportés

- Messagerie point-à-point
- Publication / souscription
- Unifiés avec JMS 1.1

Messagerie Point à Point (PTP)

- Destination de type Queue



Domaines de messagerie

Messagerie Point à Point

Messagerie Point à Point : Chaque message est adressé à une queue spécifique. Les clients lisent cette queue afin d'en extraire les messages. Les queues conservent les messages qui leur sont envoyés jusqu'à ce qu'ils soient lus ou périmés.

Chaque message ne peut être consommé qu'une seule fois

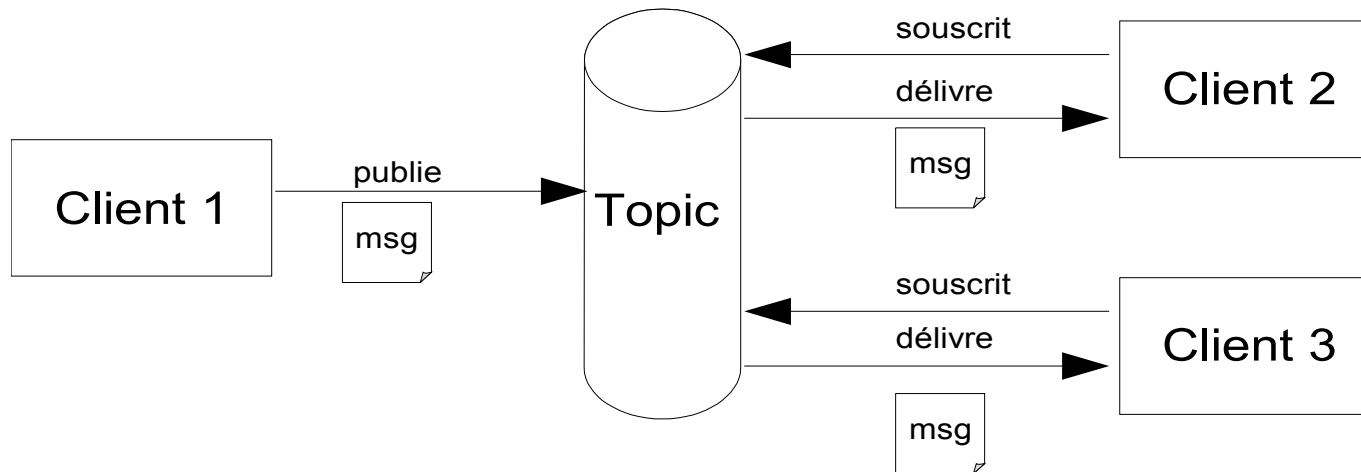
Les processus émetteur et destinataire du message peuvent ne pas être exécutés en même temps. Le message envoyé sera conservé jusqu'à ce que le destinataire l'extrait de la queue.

Le destinataire peut envoyer un message d'acquittement pour signaler la fin du traitement du message

Domaines de messagerie

Système publication / souscription (pub/sub)

- Destination de type Topic



Domaines de messagerie

Système publication / souscription (pub/sub)

Le message est publié dans un "Topic". Les destinataires des messages sont les clients ayant souscrit au topic. Le système transmet automatiquement tout message souscrit dans un topic aux clients y ayant souscrit. Les messages sont conservés le temps qu'ils soient distribués à tous les clients.

Chaque message peut être consommé plusieurs fois

Les clients souscrivant à un topic ne peuvent consommer que les messages ayant été publiés après leur souscription. Le souscripteur doit être actif au moment de la publication du message pour que celui-ci lui soit délivré.

Description de l'API

ConnectionFactory : Objet placé dans un contexte JNDI et que le client utilise pour établir une connexion vers un fournisseur JMS. Deux spécialisations existent, chacune dédiée à un type de Destination donné.

Destination : Objet placé dans un contexte JNDI et que le client utilise pour spécifier la cible des messages qu'il produit et la source des messages qu'il consomme. Une destination est de type Queue ou Topic.

Connection : Une connection représente une connexion entre un client et un fournisseur JMS (Socket TCP/IP par exemple). On utilise les connections pour créer des sessions.

Session : Les sessions sont utilisées pour créer des producteurs de messages, des consommateurs de messages ainsi que des messages. Les sessions fournissent également un contexte transactionnel permettant de grouper un ensemble d'émissions et de consommations de messages en une unité transactionnelle.

MessageProducer : Producteur de messages, créé par une session et utilisé pour envoyer des messages à une destination.

MessageConsumer : Consommateur de messages, créé par une session et utilisé pour réceptionner des messages.

MessageListener : Objet réceptionnant des messages de façon asynchrone.

Message : Objet échangé par les applications, composé d'un en-tête, de propriétés et d'un corps.

Connexions JMS

Connexion de l'API JMS

- Encapsulation d'une connexion ouverte avec le fournisseur JMS
- C'est lors de leur création que s'effectue l'authentification de l'utilisateur
- Elle peut spécifier un identifiant d'utilisateur unique
- C'est une fabrique d'objets Session
- Elle fournit des méta-données sur la connexion
- Elle peut éventuellement gérer un `ExceptionListener`

Connexions JMS

Notes

L'interface **Connection** a pour but principal d'encapsuler la connexion avec le fournisseur JMS (connexion TCP/IP par socket), mais elle fournit également d'autres services.

C'est lors de la création de la connexion que l'on peut spécifier des données d'identification afin de s'authentifier auprès du serveur JMS.

Certains fournisseurs JMS offrent la possibilité de maintenir un état lié à un client. Pour profiter de cette fonctionnalité, il est possible de spécifier un identifiant de client sur l'objet connexion.

Les objets **Session** sont créés à partir de la connexion.

Les méta-données fournies par l'API JMS permettent d'avoir des informations sur le fournisseur JMS (nom, version, ...) et sur la version de l'API JMS supportée.

En cas de problème avec une connexion, une instance de **ConnectionListener** peut en être avertie pour peu qu'elle se soit enregistrée auprès de la connexion (méthode `setExceptionListener`).

Sessions JMS

Une session JMS est un contexte mono-threadé permettant de produire et de consommer des messages.

- C'est une fabrique de MessageProducer et de MessageConsumer
- C'est une fabrique de destinations temporaires
- Elle fournit des méthodes pour créer des objets Queue ou Topic
- C'est une fabrique de messages
- Elle permet de regrouper des soumissions et des consommations de messages en une série de transactions
- Elle permet d'ordonner la soumission et la consommation des messages
- Elle retient les messages consommés jusqu'à ce que ceux-ci soient acquittés
- Elle se charge de l'exécution des MessageListeners qui se sont enregistrés auprès d'elle

Sessions JMS

Les destinations temporaires créées à partir d'une session sont en fait disponibles sur l'ensemble de la connexion.

La récupération d'une destination se fait habituellement en effectuant un look up sur un contexte JNDI. Il est cependant parfois nécessaire de pouvoir récupérer une destination "à la volée" en faisant appel aux spécificités du fournisseur JNDI.

Les messages créés par une session sont spécifiques au fournisseur JMS.

L'utilisation d'une session doit toujours se faire dans un processus mono-threadé. En effet, les MessageProducer et MessageConsumer créés à partir d'une même session ne peuvent pas travailler de manière concurrentielle (il est impossible d'émettre un message tout en attendant la réception d'un autre).

Une session peut être déclarée "transacted", c'est à dire que toute soumission ou réception de message se fait dans un contexte transactionnel. Ceci permet de regrouper un ensemble d'opérations dans une même unité afin de s'assurer de la bonne exécution de chacune des opérations (commit) ou de leur annulation totale en cas d'erreur (rollback).

L'ordre de traitement des messages envoyés par une même session, pour une destination donnée, est assuré. Il n'y a cependant aucune règle ordonnant la réception de messages soumis à partir de sessions différentes ou envoyés par une même session à diverses destinations. De plus, certaines caractéristiques des messages (priorité, persistance) peuvent influencer sur leur ordre de traitement.

Toujours fermer une session devenue inutile afin de libérer les ressources.

Messages JMS

Structure

- Des en-têtes (obligatoires)
 - JMSDestination, JMSMessageID, JMSCorrelationID, ...
- Des propriétés
 - Informations supplémentaires, utiles pour les sélecteurs de messages.
- Un corps
 - Différents types de données ou modalités d'exploitation sont disponibles

Messages JMS

Structure

Les en-têtes d'un message JMS sont un ensemble de champs pré-définis contenant des valeurs que le client et le fournisseur JMS peuvent exploiter afin d'identifier et de router le message. L'interface **Message** de l'API JMS propose un accesseur en lecture et en écriture pour chacun des en-têtes. Certaines valeurs d'en-têtes peuvent être positionnées par le client, mais la plupart sont valorisés automatiquement par les méthodes `send` ou `publish` (exemple : l'en-tête `JMSDestination` indiquant la queue ou le topic cible du message).

Les propriétés sont des informations supplémentaires (couple nom / valeur) , non disponibles dans les en-têtes et nécessaires à certains types de fournisseurs JMS. Les propriétés sont également utilisées dans le cadre de l'utilisation de Sélecteur de message. L'interface **Message** de l'API JMS expose un ensemble de méthodes pour aisément valoriser et récupérer ces propriétés.

Le corps du message est son contenu, c'est à dire l'information métier utile au consommateur du message. Le type de données que l'on peut insérer dans le corps d'un message est déterminé par le type de message.

Messages JMS

Types de messages

- **TextMessage** : information textuelle
- **StreamMessage** : information de type flux (lecture séquentielle)
- **ObjectMessage** : objet sérialisé
- **MapMessage** : ensemble de couples clé / valeur
- **ByteMessage** : flux non interprété rendu sous la forme d'une séquence d'octets
- **Message** : corps vide, ce type de message est uniquement composé d'en-tête et de propriétés.

Création

- En appelant la méthode createXxxMessage de l'interface **Session** de l'API JMS (où Xxx est le type du message).

Messages JMS

Types de messages

Chaque sous-interface de **Message** dispose de ses propres méthodes pour valoriser et récupérer le corps de ses implémentations.

- `TextMessage` : `setText` / `getText`
- `StreamMessage` : `writeXxx` / `readXxx` (où Xxx est le type de l'information à ajouter au flux)
- `ObjectMessage` : `setObject` / `getObject` (l'objet doit être sérialisable).
- `MapMessage` : `getXxx` / `setXxx` (où Xxx est le type de l'information à ajouter au flux)
- `BytesMessage` : `writeXxx` / `readXxx` (où Xxx est le type de l'information à ajouter au flux)

Selecteurs de messages

Possibilité de filtrer les messages en paramétrant un sélecteur de Messages.

- exemple : " (NewsDomain = 'J2EE' OR NewsDomain = 'JAVA') AND NewsSubject LIKE '% JMS %' "
- Les valeurs testées sont celles des en-têtes ou des propriétés des messages entrants.
- Les méthodes `createConsumer` et `createDurableSubscriber` de l'interface **Session** permettent de définir des sélecteurs de messages lors de la création d'un consommateur de message.

Selecteurs de messages

Notes

Le sélecteur de message est une chaîne de caractères permettant de filtrer les messages acceptés par le consommateur.

Les tests effectués dans le sélecteurs portent sur les en-têtes ou les propriétés des messages. Il n'est pas possible de faire un sélecteur portant sur le corps des messages.

L'exemple donné ici permet de ne laisser passer que les messages ayant des propriétés nommées "NewsDomain" et "NewsSubject". NewsDomain doit être égal à 'Java' ou 'J2EE' et NewsSubject doit contenir le mot JMS.

Exemples de programmation avec JMS

Emission d'un message

```
// Création d'un contexte JNDI
Context jndiContext = new InitialContext();

// Lookup de la fabrique de connexion et de la destination
ConnectionFactory cnxFactory = (ConnectionFactory) jndiContext.lookup("jms/CnxFactory");
Destination dest = (Destination) jndiContext.lookup("jms/Queue");

// Création d'une Connexion et d'une Session:
Connection connection = cnxFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

// Création d'un MessageProducer et d'un message de type Text
MessageProducer producer = session.createProducer(dest);
TextMessage message = session.createTextMessage("foo=bar");

// Envoi d'un message
producer.send(message);

// Fermeture de la connexion
connection.close();
```

Exemples de programmation avec JMS

Emission d'un message

On note que le code est totalement indépendant du type de la destination du message.

Le premier paramètre de la méthode **createSession** de **Session** indique si la session s'exécute dans un contexte transactionnel. Si c'est le cas, le mode d'acquittement (second paramètre) est ignoré. Dans le cas d'une session "transacted", il est d'usage de déclarer un mode d'acquittement à "0".

Le code présenté dans cet exemple ne montre pas la gestion des exceptions éventuelles (NamingException, JMSException). Les instructions pouvant entraîner ces exceptions doivent être traitées dans un bloc "try", la fermeture de la connection doit, quant à elle, se trouver dans un bloc "finally" après avoir pris soin de vérifier que l'objet connection est non null.

Exemples de programmation avec JMS

Réception et traitement d'un message

```
// Création d'un contexte JNDI
Context jndiContext = new InitialContext();

// Lookup de la fabrique de connexion et de la destination
ConnectionFactory cnxFactory = (ConnectionFactory) jndiContext.lookup("jms/CnxFactory");
Destination dest = (Destination) jndiContext.lookup("jms/Queue");

// Création d'une Connexion et d'une Session
Connection connection = cnxFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

// Création d'un MessageConsumer
MessageConsumer consumer = session.createConsumer(dest);

// Réception des messages jusqu'à obtention d'un message non texte
while (true) {
    Message m = consumer.receive(1);
    if (m instanceof TextMessage) {
        traiterMessage(m); // traiterMessage( m : TextMessage ) : void
    } else {
        break;
    }
}

// Fermeture de la connexion
connection.close();
```

Exemples de programmation avec JMS

Réception d'un message

La récupération de la `ConnectionFactory` et de la destination ainsi que la création de la connexion et de la session sont identiques à l'exemple précédent.

La lecture des messages d'une destination s'effectue via un objet `MessageConsumer`, créé à partir de la session en spécifiant la destination.

La méthode `receive` du consumer attend la réception d'un message pendant un temps donné (passé en paramètre, exprimé en millisecondes, 0 signifiant que le processus reste bloqué jusqu'à l'obtention d'un message).

Dans l'exemple, c'est la récupération d'un message de type autre que `Text` qui met fin à la consommation de messages. Un moyen simple de créer un tel message est d'utiliser la méthode `createMessage` de `Session`.

Encore une fois la gestion des exceptions n'est pas montrée, mais les remarques de l'exemple précédent s'appliquent également ici.

Exemple de programmation avec JMS

Traitement asynchrone

Création d'une implémentation de `MessageListener` (définition de la méthode `onMessage`) :

```
import javax.jms.*;

public class MonListener implements MessageListener {

    public void onMessage(Message message) {
        try {
            if (message instanceof TextMessage) {
                String msgText = ((TextMessage) message).getText();
                traiterTexteMessage(msgText);
            }
        } catch (JMSEException jmse) {
            jmse.printStackTrace();
        }
    }
    [...]
}
```

Ajout de ce listener à un `Consumer` dans une méthode d'une classe cliente :

```
// Création d'un MessageConsumer
MessageConsumer consumer = session.createConsumer(dest);
// Création du listener
MonListener listener = new MonListener();
// Mise en liaison
consumer.setMessageListener(listener);
```

Exemple de programmation avec JMS

Traitement asynchrone

Le principe est de créer une implémentation de `MessageListener` puis d'associer ce listener à un `messageConsumer`.

Création d'applications JMS robustes

- Gestion de la fiabilité de l'application grâce aux mécanismes basiques suivants :
 - Contrôle du message d'acquittement
 - Spécification de la persistance des messages
 - Spécification des niveaux de priorité des messages
 - Gestion de l'expiration des messages
 - Création de destinations temporaires
- Gestion avancée de la fiabilité :
 - Souscriptions durables
 - Mode transactionnel

Création d'applications JMS robustes

- JMS propose divers mécanismes pour traiter des messages de façon fiable et robuste :
 - Il est possible de spécifier différents niveaux de contrôle du message d'acquittement
 - Les messages peuvent être persistents afin de se protéger d'une interruption du service de messagerie
 - L'ordre de traitement des messages peut être modifié en leur attribuant un niveau de priorité
 - Une durée de vie peut être associée à chaque message afin de garantir le non-traitement de messages obsolètes
 - Il est possible de créer des destinations temporaires d'une durée de vie égale à la connection utilisée pour les créer.
- D'autres mécanismes plus élaborés viennent s'ajouter :
 - Dans un système publication / souscription, il est possible de créer des souscriptions valides même quand le souscripteur est inactif.
 - Il est possible de grouper un ensemble de soumission / traitement / acquittement de message au sein d'une même unité transactionnelle afin d'en garantir la totale exécution.

JMS 1.1

Evolution attendue

- Uniformisation des deux domaines Queue et Topic
- Compatibilité avec l'API version 1.0.2

Entités des versions 1.0.2 et 1.1

- La plupart des entités 1.0.2 sont des sous-classes des entités 1.1

Table 5: Entités des versions 1.0.2 et 1.1

Entité JMS 1.1	Entité 1.0.2 Queue	Entité 1.0.2 Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber
Destination	Queue	Topic

JMS 1.1

Notes

Java, Programmation avancée

Programmation réseau à base de Sockets

Version 2.2

- Principes
- Sockets TCP avec ServerSocket et Socket
- Sockets UDP avec Datagram

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Socket réseau

Moyen de communication inter-processus

- Notion introduite dans les distributions Unix de type Berkeley
- Permet à deux processus de communiquer
 - sur la même machine
 - sur deux machines connectées par un réseau TCP/IP
- Deux modes de communication
 - mode connecté / protocole TCP
 - mode non connecté / protocole UDP

En Java

- `java.net` : propose le support des deux mode de communication par sockets
- `java.io` : propose le support des flux permettant d'envoyer / recevoir des données
- `java.nio` : entrées / sorties optimisées

Sockets réseau

Notes

Sockets en mode connecté

ServerSocket

- Partie serveur de la communication par socket
- ServerSocket = la partie qui attend les connexions entrantes
 - attente bloquante
 - optimisation possible : package java.nio

Socket

- Deux usages
 - partie cliente de la communication par Socket
 - extrémité côté serveur d'une socket ouverte par un client
- Permet d'établir une communication bidirectionnelle
 - fournit un InputStream pour les lectures
 - fournit un OutputStream pour les écritures

Sockets en mode connecté

Notes

ServerSocket

Usage typique

```
ServerSocket server = new ServerSocket(5555);
System.out.println ("Serveur prêt");
while (! end) {
    try {
        Socket client = server.accept();
        InputStream is = client.getInputStream();
        OutputStream os = client.getOutputStream();
        // traitement des données
        // ...
    } catch (IOException e) {
        e.printStackTrace();
    }
}
server.close();
```

- `accept()` est bloquant
- Idéalement, les traitements sont à exécuter au sein d'un thread séparé
 - le serveur redevient plus rapidement disponible pour traiter un client

ServerSocket

Notes

Socket

Usage typique

```
Socket socket = new Socket("monserveur", 5555);  
InputStream is = socket.getInputStream();  
OutputStream out = socket.getOutputStream();  
  
// envoi et/ou réception de données  
  
socket.close();
```

- Nécessité de définir un protocole d'échange de données commun entre le serveur et les clients
- Nécessité de s'entendre sur les modalités de fermeture des sockets

Socket

Notes

Sockets en mode non connecté

UDP

- Le protocole UDP est plus simple que TCP. Il fournit un transfert non fiable de groupe d'octets (datagrammes) entre un client et le serveur.
- UDP ne permet pas de garantir l'ordre d'envoi de messages. Les données transmises peuvent être reçues dans le désordre ou perdues (arrivé, temps d'arrivé et contenu non garantis)
 - plus efficace en bande passante.
 - moins fiable puisqu'il n'y a pas d'accusé de réception.
- Le serveur se met en attente de réception de paquets sur sa socket.
- Le client envoie un paquet via sa socket en précisant l'adresse du destinataire. Si le client envoie un paquet avant que le serveur ne soit prêt à recevoir, le paquet est perdu.

Socket

Notes

Sockets en mode non connecté

UDP

- 2 classes pour UDP: les datagrammes sont émis et reçus par l'intermédiaire d'un objet de la classe DatagramSocket:
- DatagramSocket() pour un client
- DatagramSocket(ServerPort) pour un serveur

Socket

Notes

Sockets en mode non connecté

Etape d'une communication client serveur en UDP avec Java, coté serveur:

- Création d'un datagram socket sur un port qui permet au serveur de communiquer avec tous ses clients.

```
socket = new DatagramSocket(5555) ;  
  
//Créer un socket UDP et l'attacher à un port spécifique de la machine  
//locale.
```

- Création d'un paquet d'entrée (un datagram, c.-à-d. tableau d'octet qui recevra les données). Attente de données en entrées.

```
byte[] donnees = new byte[4000];  
//création de tableau qui contiendra les données reçues.  
  
DatagramPacket d = new DatagramPacket(donnees, donnees.length);  
//Constructeur pour la réception de paquet UDP de 1 octets de données. La partie  
//"données" du paquet reçu est mappée sur le buffer b
```

Socket

Notes

Java n'impose aucune limite en taille pour les tableaux d'octets circulant dans les paquets UDP, mais pour tenir dans un seul datagramme IP, le datagramme UDP ne doit pas contenir plus de 65467 octets de données. Un datagramme UDP est rarement envoyé via plusieurs datagrammes

Sockets en mode non connecté

Etape d'une communication client serveur en UDP avec Java, coté serveur:

- Réception et analyse des données en entrée.

```
DatagramSocket serveur = new DatagramSocket (port) ;  
  
serveur.receive (paquet) ;  
//attente de réception d'un paquet.
```

- Création d'un paquet de sortie. Préparation et envoie de la réponse. Une réponse est mise dans un DatagramPacket et envoyée sur le DatagramSocket du coté du demandeur.

Socket

Notes

méthode receive() : méthode bloquante sans contrainte de temps. Peut rester en attente indéfiniment si aucun paquet n'est jamais reçu. Cependant il y a la possibilité de préciser un délai maximum:

```
public void setSoTimeout(int timeout) throws SocketException
```

Sockets en mode non connecté

Etape d'une communication client serveur en UDP avec Java, coté client:

- Création d'un paquet de sortie

```
String s = "le paquet";  
// le message à envoyer.  
  
byte[] donnee = s.getBytes();  
//données à envoyer (chaîne de caractères)  
  
InetAddress adresse = InetAddress.getByName("le nom de l'hôte");  
//add contient l'adresse IP de la partie serveur.  
  
int port = 5555;  
  
DatagramPacket paquet = new DatagramPacket(donnees,donnees.lenght,adresse, port);  
//creation du paquet avec les données et l'ad. du serveur et port sur  
//lequel il écoute
```

Socket

Notes

Le destinataire est identifié par le couple @IP/port précisé dans le paquet.

Sockets en mode non connecté

Etape d'une communication client serveur en UDP avec Java, coté client:

- Préparer et envoyer une requête

```
DatagramSocket client = new DatagramSocket() ;  
// création d'une socket, sans la lier à un port particulier.  
  
client.send(paquet) ;  
// envoie du paquet via la socket
```

- Créer un paquet en entrée
- Attendre des données en entrée
- Les recevoir et les traiter

Socket

Notes

Le destinataire est identifié par le couple @IP/port précisé dans le paquet

Sockets en mode non connecté

Exemple UDP- coté émetteur 1

```
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.net.*;

public class EmetteurUDP {
    static int serverPort = 0;
    static String serverName = null;
    static int msgSize = 100;

    public static void main(String[] args){
        DatagramSocket emmetteurSocket = null;
        InetAddress serverIpAdr = null;
        DatagramPacket paquet = null;
        byte[] msg;
```

Socket

Notes

Sockets en mode non connecté

Exemple UDP- coté émetteur2

```
//initialisation des variables en utilisant la ligne de commande
serverName = args[0];
serverPort = (Integer.valueOf(args[1])).intValue();
try {
    msg = args[2].getBytes("UTF-16");
} catch (UnsupportedEncodingException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}

//création de socket
try {
    emmetteurSocket = new DatagramSocket();
    System.out.println("La socket de l'émetteur a été créée");
} catch (SocketException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

Socket

Notes

Sockets en mode non connecté

Exemple UDP- coté émetteur 3

```
//Initialiser l'add IP du serveur en utilisant son nom
try {
    serverIpAdr = InetAddress.getByName(serverName);
} catch (UnknownHostException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
//utilisation de la socket
msg = new byte[msgSize];
paquet=new DatagramPacket(msg,msgSize, serverIpAdr, serverPort);
try {
    emmetteurSocket.send(paquet);
    System.out.println("Le message a été envoyé");
} catch (IOException e) {
    e.printStackTrace();
}
```

Socket

Notes

Sockets en mode non connecté

Exemple UDP- coté émetteur 4

```
        //fermer la socket
        emmetteurSocket.close();
        System.out.println("La socket de l'émmeteur a été fermée!");
    }
}
```


Socket

Notes

Sockets en mode non connecté

Exemple UDP- coté récepteur 1

```
import java.io.IOException;
import java.net.*;
public class RecepteurUDP {
    static int serverPort = 0;
    static int msgSize = 100;
    public static void main(String[] args){
        DatagramSocket serverSocket = null;
        DatagramPacket paquet = null;
        byte[] msg;
        //Initialisation de serverPort en utilisant la ligne de commande
        serverPort = (Integer.valueOf(args[0])).intValue();
        // création de socket
        try {
            serverSocket = new DatagramSocket(serverPort);
        } catch (SocketException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();} }
```

Socket

Notes

Sockets en mode non connecté

Exemple UDP- coté récepteur 2

```
//serveur reçoit un message
msg = new byte[msgSize];
paquet = new DatagramPacket(msg, msgSize);

try {
    serverSocket.receive(paquet);
    //récupération et affichage des données
    String chaine =
        new String(paquet.getData(), 0, paquet.getLength());
    System.out.println(" reçu : "+chaine);
} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
serverSocket.close();

}
```

Socket

Notes

Sockets en mode non connecté multicast

UDP multicast:

- UDP offre un autre mode de communication: multicast.
- Multicast: plusieurs récepteur pour une seul émission d'un paquet. L'objectif est d'éviter de créer plusieurs connexions et d'envoyer un message plusieurs fois.
- Multicast IP: envoie d'un datagramme sur une add IP particulière. Classe d'add IP entre 224.0.0.0 et 239.255.255.255. Classe entre 225.0.0.0 et 238.255.255.255 sont utilisables par un programme quelconque et les autres sont réservées.
- Une add IP multicast n'identifie pas une machine sur un réseau mais un groupe multicast.
- Le DatagramPacket est construit à partir de l'add de plusieurs clients.

Socket UDP multicast:

- Il faut rejoindre un groupe avant l'envoi d'un paquet.
- Un paquet envoyé par un membre de groupe, est reçu par tous les membres de ce groupe.
- C'est UDP, donc non fiable et non connecté
- Classe `java.net.MulticastSocket`

Socket

Notes

Rappels:

Broadcast (diffusion) : envoi de données à tous les éléments d'un réseau

Multicast : envoie de données à un sous-groupe de tous les éléments d'un réseauSockets en mode non connecté multicast

Sockets en mode non connecté multicast

UDP multicast:

- Création de DatagramSocket est comme UDP standard
- Gestion des groupes

```
public void joinGroup(InetAddress mcastaddr) throws IOException  
//Rejoint le groupe dont l'adresse IP multicast est passée en paramètre.  
//L'exception est levée en cas de problèmes, notamment si l'adresse IP  
//n'est pas une adresse IP multicast valide.
```

```
public void leaveGroup(InetAddress mcastaddr) throws IOException  
Quitte un groupe de multicast. L'exception est levée si l'adresse IP  
n'est pas une adresse IP multicast valide. Pas d'exception levée ou de  
problème quand on quitte un groupe auquel on appartient pas
```

- Emission/réception de données: tout comme avec une socket UDP standard. (En utilisant les services send() et receive() avec des paquets de type DatagramPacket)

Socket

Notes

Sockets en mode non connecté multicast

UDP multicast, coté serveur:

```
...;  
InetAddress group = InetAddress.getByName("230.0.0.0");  
...  
packet = new DatagramPacket(buf, buf.length, group, 4446);  
...
```

- L'add 230.0.0.0 correspond à un identificateur de group.
- Tous les client doit avoir un multicast socket lié à no 4446.
- Le DatagramPacket est destiné à tous les clients qui écoutent le port 4446 et qui sont membres de groupe 230.0.0.0

UDP multicast, coté client:

- Pour écouter le port 4446 le program client doit créer son multicastSocket avec ce numéro et il doit utiliser la méthode joinGroupe() pour rejoindre le groupe 230.0.0.0

Socket

Notes

Sockets en mode non connecté multicast

Exemple UDP multicast, coté récepteur 1:

```
import java.net.*;
import java.io.*;
public class UDPMulticastReceiver {
    public static void main(String[] args){
        InetAddress group = null;
        try{group= InetAddress.getByName("230.0.0.0"); //l'adr multicast
        } catch (UnknownHostException e) {e.printStackTrace();}

        MulticastSocket ms = null;
        try {ms = new MulticastSocket(4446); // le port local
        } catch (NumberFormatException e) {e.printStackTrace();}
        catch (IOException e) {e.printStackTrace();}
        try {ms.joinGroup(group);
        } catch (IOException e) {e.printStackTrace();}
```

Socket

Notes

Sockets en mode non connecté multicast

Exemple UDP multicast, coté récepteur 2:

```
DatagramPacket p = new DatagramPacket(new byte[100], 100);
try {ms.receive(p);
} catch (IOException e) {e.printStackTrace();}

String s = "";
try {s = new String(p.getData(), "UTF-16");
} catch (UnsupportedEncodingException e) {e.printStackTrace();}

System.out.println("Recu : "+ s);

try {ms.leaveGroup(group);
} catch (IOException e) {e.printStackTrace();}
ms.close();
}
```

Socket

Notes

Sockets en mode non connecté multicast

Exemple UDP multicast, coté émetteur 1:

```
import java.io.*;
import java.net.*;
public class UDPMulticastSender {
    public static void main(String[] args){
        // args 0 : le message à envoyer
        InetAddress group = null;
        try{group=InetAddress.getByName("230.0.0.0");} //l'adr multicast
        catch (UnknownHostException e) {}

        MulticastSocket ms = null;
        try{
            ms = new MulticastSocket();
            ms.setTimeToLive(1);
        } catch (IOException e) {}
    }
}
```


Socket

Notes

Sockets en mode non connecté multicast

Exemple UDP multicast, coté émetteur 2:

```
DatagramPacket paquet = null;
byte[] buf = null;
try {buf = args[0].getBytes("UTF-16");}
catch (UnsupportedEncodingException e) {}

paquet = new DatagramPacket(buf,buf.length,group, 4446);
try {ms.send(paquet);}
catch (IOException e) {e.printStackTrace();}
}
}
```

Socket

Notes

Java, Programmation avancée

Entrées-sorties de type NIO

Version 2.2

- Principes et objectifs
- Channel et Buffer
- Exemples de mise en oeuvre avec des fichiers
- Problématiques d'encodage/décodage des caractères

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Caractéristiques

Entrées-sorties très performantes

- Utilisation plus poussée des mécanismes offerts par le système d'exploitation
- Nouvelle métaphore : le bloc
 - les données sont traitées par blocs plutôt que de façon unitaire (philosophie des streams)
 - c'est plus efficace

Nouveau paquetage : `java.nio`

- Intégré à partir de Java 1.4
- `java.io` est toujours proposé, mais réécrit avec certaines fonctionnalités nio.
- Nouveaux concepts
 - Channel : fournit et/ou reçoit des données, sous la forme de Buffers
 - Buffer : objet typé stockant des données, utilisé en lecture ou écriture

Caractéristiques

Notes

Exemple: fichier

Lecture

```
FileInputStream fin = new FileInputStream("monfichier.txt" );
FileChannel fc = fin.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
int status = fc.read( buffer );
if (status == -1)
    System.out.println ("plus rien à lire");
```

Ecriture

```
FileOutputStream fout = new FileOutputStream( "writesomebytes.txt" );
FileChannel fc = fout.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
for (int i=0; i<message.length; ++i) {
    buffer.put( message[i] );
}
buffer.flip();
fc.write( buffer );
```


Exemple: fichier

Notes

Buffer

Plusieurs types

- ByteBuffer
- CharBuffer
- ShortBuffer
- IntBuffer
- LongBuffer
- FloatBuffer
- DoubleBuffer

Variables d'état

- capacity : capacité maximale du stockage
- limit
 - données restant à lire dans le cas de l'écriture d'un Buffer dans un Channel
 - emplacements libres dans le cas d'un buffer de lecture
- position : position de lecture / écriture

Buffer

Notes

Buffer.flip()

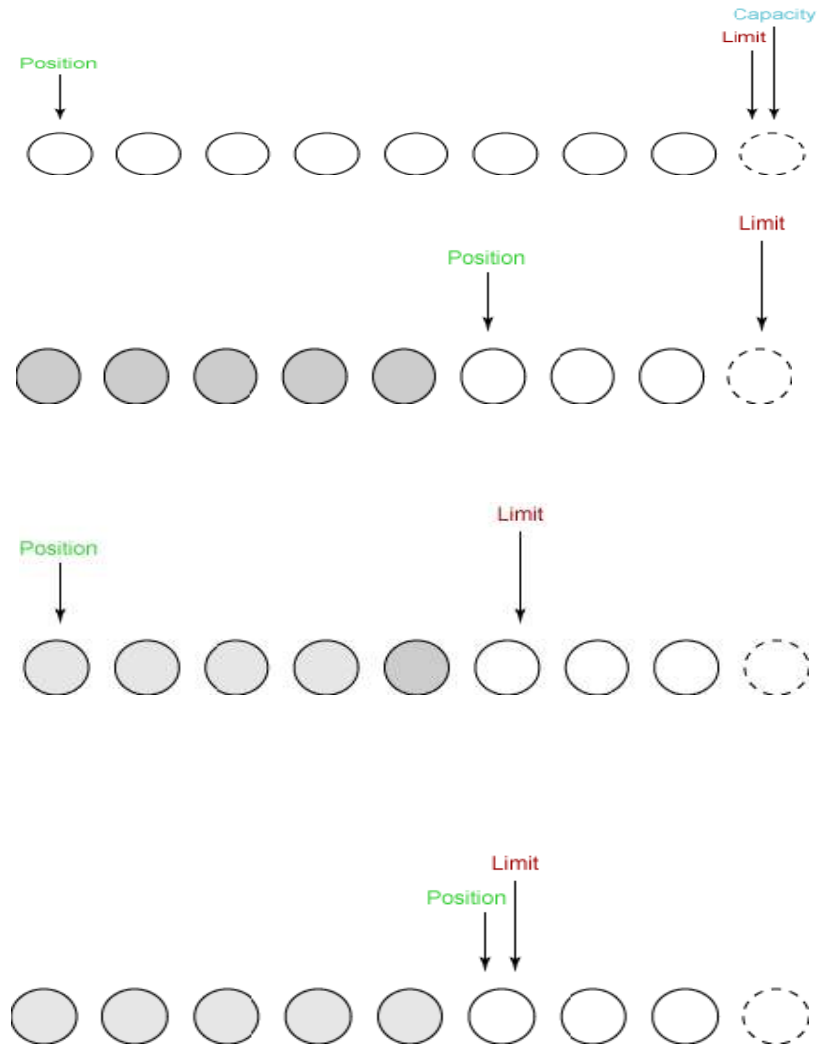
Exemple

- création
- lecture de données

flip()

- Prépare le buffer à l'écriture dans un channel
 - limit = position
 - position = 0

Ecriture des données



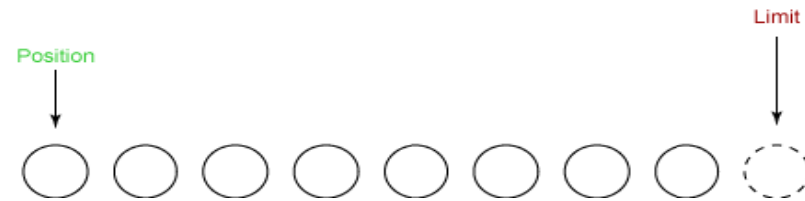
Buffer.flip()

Notes

Buffer.clear()

clear()

- Prépare le buffer à recevoir des données
 - `limit = capacity`
 - `position = 0`



Buffer.clear()

Notes

Opérations de lecture

ByteBuffer.get()

- Lecture relative
 - `byte get();`
 - `get(byte dst[]);`
 - `get(byte dst[], int offset, int length);`
 - `getXXX();` avec XXX étant (Byte, Char, Short, Int, Long, Float, Double)
- Lecture absolue
 - `byte get(int index);`

Opération de lecture

Notes

Opérations d'écriture

ByteBuffer.write()

- Méthodes relatives
 - `put(byte b);`
 - `put(byte src[]);`
 - `put(byte src[], int offset, int length);`
 - `put(ByteBuffer src);`
 - `putXXX(XXX data);` avec XXX étant (Byte, Char, Short, Int, Long, Float, Double)
- Méthodes absolues
 - `put(int index, byte b);`

Opérations d'écriture

Notes

Allocation d'un buffer

Directe

- Buffer standard

```
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
```

- Buffer sur lequel les opération d'E/S sont exécutées directement (fonction de la JVM)

```
ByteBuffer buffer = ByteBuffer.allocateDirect( 1024 );
```

A partir d'un tableau

- Données partagées entre le tableau et le buffer

```
byte array[] = new byte[1024];  
ByteBuffer buffer = ByteBuffer.wrap( array );
```

En lecture seule

```
buffer.asReadOnlyBuffer();
```

Allocation d'un buffer

Notes

Mapper un fichier en mémoire

Caractéristiques

- Utilisation d'une fonctionnalité du système d'exploitation
- Attention : modifier la mémoire revient à modifier directement le contenu du fichier mappé

Mise en oeuvre

- `FileChannel.map()`
- `MappedByteBuffer` sous-classe de `ByteBuffer`

```
FileInputStream fin = new FileInputStream("monfichier.txt" );  
FileChannel fc = fin.getChannel();  
  
// on mappe les 1024 premiers octets du fichier  
//  
MappedByteBuffer mb = fc.map (FileChannel.MapMode.READ_WRITE, 0, 1024 );
```

Mapper un fichier en mémoire

Notes

Gestion des verrous

Obtenir un verrou

```
RandomAccessFile raf = new RandomAccessFile( "usefilelocks.txt", "rw" );  
FileChannel fc = raf.getChannel();  
FileLock lock = fc.lock( start, end, false );
```

Le relacher

```
lock.release();
```


Gestion des verrous

Notes

Utiliser plusieurs buffers simultanément

- Utile lors de la gestion de formats de données complexes

ScatterByteChannel

- Interface implémentée par les Channels
- Méthodes supplémentaires
 - `long read(ByteBuffer[] dsts);`
 - `long read(ByteBuffer[] dsts, int offset, int length);`

GatheringByteChannel

- Interface implémentée par les Channels
- Méthodes supplémentaires
 - `long write(ByteBuffer[] srcs);`
 - `long write(ByteBuffer[] srcs, int offset, int length);`

Utiliser plusieurs buffers simultanément

Notes

Encodage / décodage octet-unicode

Problème

- Java stocke les chaînes de caractères en Unicode
- Les caractères en dehors de Java sont souvent gérés sous forme d'octets
- Il faut gérer des conversions OCTET - UNICODE

java.nio.charset

- Charset : représentation nommée en format 8-bits d'un jeu de caractères Unicode 16-bits
- CharsetDecoder : transforme un jeu de caractères 8-bits d'un charset donné en un jeu de caractères Unicode 16-bits
- CharsetEncoder : transforme des caractères Unicode 16-bits en bytes pour un charset donné

Encodage / décodage octet-unicode

Notes

Exemple octets-unicode

Décodage

```
Charset latin1 = Charset.forName( "ISO-8859-1" );  
CharsetDecoder decoder = latin1.newDecoder();  
CharBuffer cb = decoder.decode( inputData ); // inputData est un ByteBuffer
```

Encodage

```
Charset latin1 = Charset.forName( "ISO-8859-1" );  
CharsetEncoder encoder = latin1.newEncoder();  
ByteBuffer outputData = encoder.encode( cb ); // cb est un CharBuffer
```

Exemple octets-unicode

Notes

Java, Programmation avancée

Entrées-sorties de type NIO appliquées au réseau

Version 2.2

- Utiliser les NIO pour gérer les E/S réseau

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

E/S réseau asynchrones

Sans NIO

- Un ServerSocket instancie une Socket pour chaque connexion entrante
- La lecture des données de chaque socket est bloquante, ce qui impose l'usage d'un thread séparé

Avec NIO

- On s'enregistre à l'aide d'un Selector comme étant intéressé par certains événements
 - OP_ACCEPT : connexion d'un client
 - OP_READ : présence de données à lire
- Au sein d'une boucle infinie
 - on se met en attente des événements pour lesquels on s'est enregistré
 - on les traite de façon non bloquante

E/S réseau asynchrones

Notes

ServerSocketChannel

Exemple

- Soit un programme qui traite des connexions sur plusieurs ports à la fois
- Pour chacune des adresses d'écoute
 - création du ServerSocketChannel
 - enregistrement sur l'événement OP_ACCEPT (arrivée nouvelle connexion)

```
Selector selector = Selector.open();

for (int i = 0; i < NBPORTS; i++) {
    ServerSocketChannel ssc = ServerSocketChannel.open();
    ssc.configureBlocking( false );
    ServerSocket ss = ssc.socket();
    InetSocketAddress address = new InetSocketAddress( ports[i] );
    ss.bind( address );
    SelectionKey key = ssc.register( selector, SelectionKey.OP_ACCEPT );
}
```

ServerSocketChannel

Notes

SocketChannel

Exemple (suite)

```
while (true) {
    int num = selector.select();
    Set selectedKeys = selector.selectedKeys();
    Iterator it = selectedKeys.iterator();
    while (it.hasNext()) {
        SelectionKey key = (SelectionKey)it.next();
        if ((key.readyOps() & SelectionKey.OP_ACCEPT) == SelectionKey.OP_ACCEPT) {
            // on accepte la connexion entrante
            // on s'enregistre à l'évènement OP_READ
            ServerSocketChannel ssc = (ServerSocketChannel)key.channel();
            SocketChannel sc = ssc.accept();
            sc.configureBlocking( false );
            SelectionKey newKey = sc.register( selector, SelectionKey.OP_READ );
            // it.remove(); pas indispensable ici, mais à considérer de façon générale
        } else if ((key.readyOps() & SelectionKey.OP_READ) == SelectionKey.OP_READ) {
            SocketChannel sc = (SocketChannel)key.channel();
            // lecture et exploitation des données
            // ...
            // it.remove(); pas indispensable ici, mais à considérer de façon générale
        }
    }
}
```

SocketChannel

Notes

SocketChannel

Autres considérations

- `selector.selectedKeys()` retourne un Set
 - il faut être certain de ne pas exploiter deux fois la même clé
 - il peut être utile de retirer une clé déjà traitée du Set
- L'exploitation des données peut être effectuée à l'aide d'un thread séparé

SocketChannel

Notes

Java, Programmation avancée

Threads et MultiThreading

Version 2.2

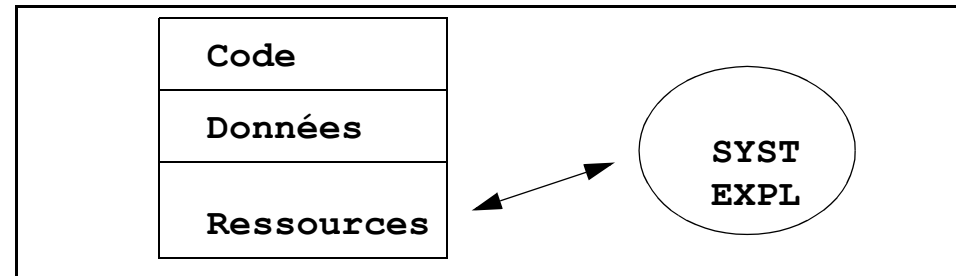
- Déclarer et définir un thread
- Présenter les intérêts de la programmation multithreadée
- Maîtriser en Java les deux types de threads utilisables
- Définir les différents états d'un thread et les méthodes permettant de modifier ces états
- Choisir la priorité d'un thread

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

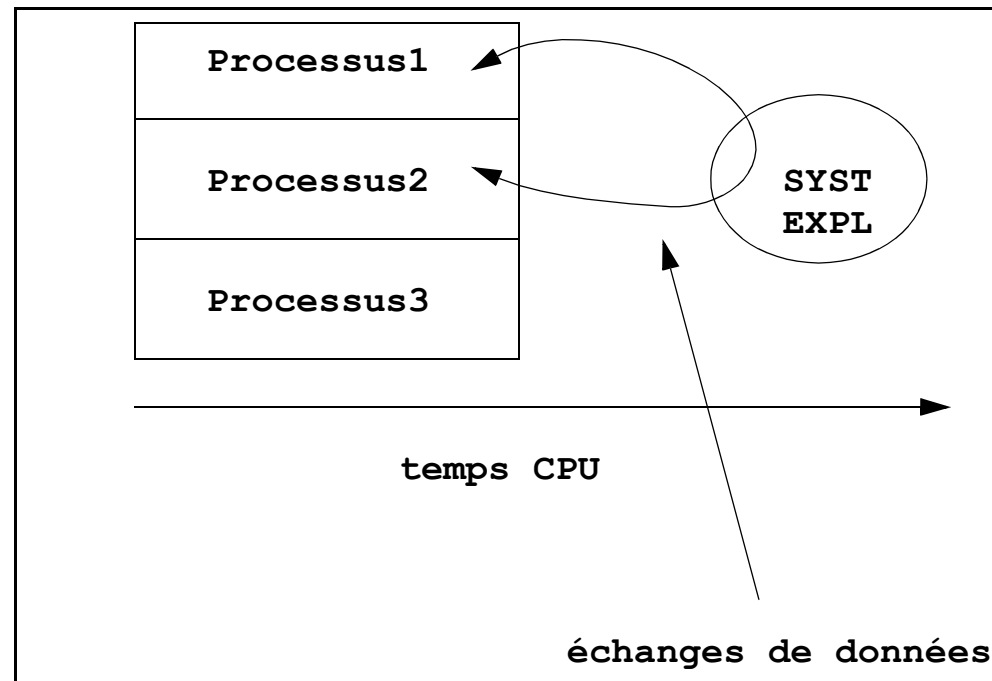
Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Programmation parallèle

Processus = unité fonctionnelle



Exécution



Programmation parallèle

Processus

- Processus = une UNITE d'exécution qui possède son code, son espace mémoire et un accès à des ressources gérées par le système d'exploitation.
- La vie d'un processus est gérée par le système d'exploitation de la machine sur laquelle il s'exécute.
- La communication inter-processus s'effectue par des mécanismes offerts par le système d'exploitation (*sockets, pipes, ...*).

Programmation parallèle

Difficultés fréquentes

- Un processus "embarque" son code et ses données, ce qui en fait un objet souvent plus coûteux en espace mémoire que nécessaire.
- Le partage des données entre processus est difficile à réaliser.
- Le lancement d'un processus est relativement lent à cause de :
 - sa taille,
 - la charge éventuelle du système d'exploitation.
- Le partage de certaines ressources systèmes est difficile, parfois impossible.
 - la sortie standard,
 - un fichier,
 - ...

Programmation parallèle

Notes

Thread

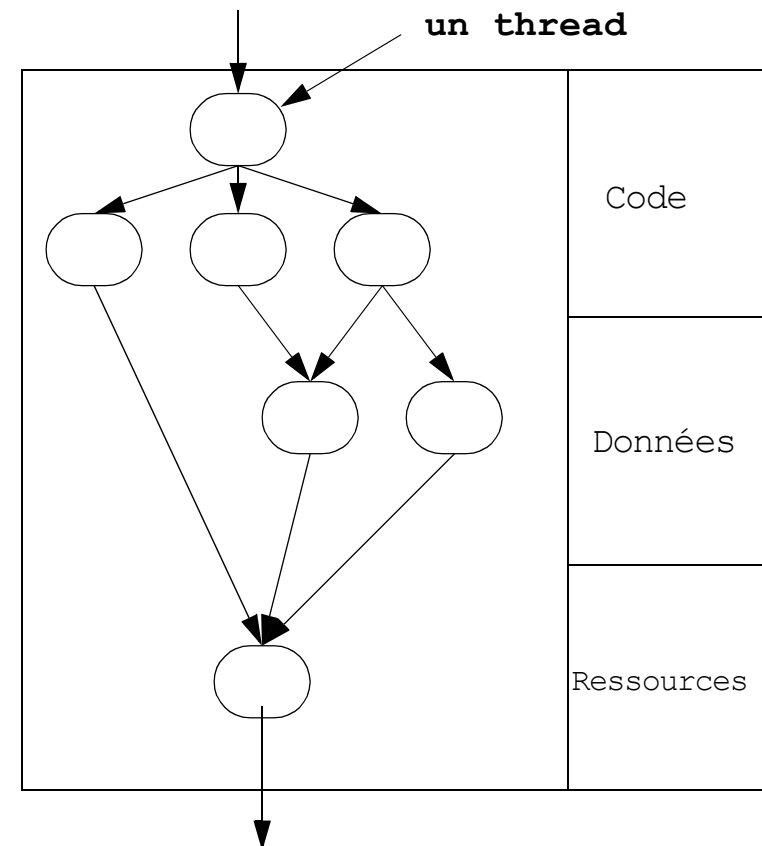
Définition

- "processus INTERNE" à un processus,
- partage les données, le code et les ressources de son processus,
- peut disposer de ses données propres.
- parfois appelé "lightweight process"

Avantages

- Légèreté
- Partage de données
- Partage de ressources systèmes

**Durée
de vie**



Thread

Définition

- Un processus comporte un ou plusieurs threads.
- Les threads d'un processus partagent tous son code, ses données et ses ressources.
- Les threads d'un processus peuvent s'exécuter en parallèle.
- Un thread peut disposer de ses données propres.

Avantages

- Légèreté grâce au partage du code et des données. Cela entraîne de meilleures performances :
 - au lancement,
 - en exécution.
- Partage des ressources systèmes du processus. Cette fonctionnalité est extrêmement pratique, ne serait-ce qu'au niveau des entrées-sorties.

Utilité du multithreading

Puissance

- Puissance de modélisation : le monde réel est massivement parallèle.
- Puissance d'exécution grâce au parallélisme.
 - indispensable côté serveur pour traiter plusieurs requêtes simultanément
 - certaines machines virtuelles peuvent exécuter les threads sur différents processeurs, ou sur les différents coeurs d'un processeur multi-coeurs

Abord simplifié

- Un thread est un OBJET comme un autre.
- La répartition et l'ordonnancement des threads sont pris en charge par la Machine Virtuelle

Mais

- La conception d'applications multi-threadée peut-être complexe
- La mise au point est complexe.

Utilité du multithreading

Puissance

- De nombreuses applications sont composées de tâches qui s'exécutent en parallèle. Disposer de capacités de programmation parallèle permet de modéliser ces applications de façon fidèle.
- Certaines implémentations de machines virtuelles permettent d'exploiter les ressources des architectures multi-processeurs SMP.

Simplicité

- Un thread est un objet et en tant que tel dispose de toutes les possibilités liées à la programmation Objet :
 - attributs,
 - méthodes,
 - encapsulation,
 - réutilisation ...

Utilité du multithreading

Exemple

- Deux vérificateurs d'une usine doivent contrôler le contenu de sacs d'objets.

```
public class Thread1 {  
    public static void main(String args[]) {  
        Sac plus = new Sac (10, '+');  
        Sac moins= new Sac (10, '-');  
        Verificateur Gaston = new Verificateur ("Gaston", plus);  
        Verificateur Robert = new Verificateur ("Robert", moins);  
        // début du "contrôle"  
        Gaston.run ();  
        Robert.run ();  
    }  
}
```

Le résultat est le suivant :

```
Z:\supports\Java v2.0\threads>java Usine  
+++++++  
Gaston a termine  
-----  
Robert a termine  
Z:\supports\Java v2.0\threads>_
```

Conclusion

- Sans thread, des activités réelles parallèles deviennent séquentielles en programmation classique.

Utilité du multithreading

Exécution séquentielle

Il faut paralléliser les opérations effectuées par les vérificateurs.

Classes Sac et Verificateur

```
class Sac {
    char contenu;
    int taille;
    public Sac (int taille, char contenu) {
        this.taille = taille;
        this.contenu = contenu;
    }
}
class Verificateur {
    private Sac sac;
    private String nom;
    public Verificateur (String nom, Sac sac) {
        this.nom = nom;
        this.sac = sac;
    }
    public void run () {
        for (int i = 0; i < sac.taille; i++) {
            System.out.print (sac.contenu);
        }
        System.out.println ("\n" + nom + " a terminé");
    }
}
```

Classe Thread

Infinité (théorique) de processeurs virtuels

- Pour définir un thread, il faut:
 - du code = une méthode, définie au sein d'une classe
 - des données manipulées par ce code = un objet
 - un processeur virtuel support d'exécution = une instance de Thread

Première approche

- `java.lang.Thread` est la classe d'implémentation d'un thread.
- Le code d'une instance de Thread s'exécute "parallèlement" au code des threads déjà existants.

Utilisations

- Deux types d'utilisations sont possibles :
 - associer un thread à un objet
 - sous-classer Thread

Classe Thread

Notes

La programmation Objet définit deux approches pour concevoir l'utilisation de concepts existants :

- sous-classer permet de spécialiser une classe et d'enrichir son comportement,
- définir une aggrégation ou association entre la classe à utiliser et la classe

Il en est de même pour le multithreading. Thread est la classe que nous souhaitons réutiliser. Nous disposons donc de deux manières de procéder pour définir de nouvelles classes d'objets bénéficiant du multithreading :

1. sous-classer Thread, car toute instance d'une sous-classe de Thread est un Thread.
2. associer Thread à une classe utilisateur afin de pouvoir associer une instance de Thread à un objet utilisateur pour que ce dernier puisse s'exécuter en parallèle.

Classe Thread

`java.lang.Thread` **dispose de l'ensemble des services de gestion d'un thread**

- création et destruction
- gestion de l'état
- gestion de la priorité
- synchronisation

Complétée par

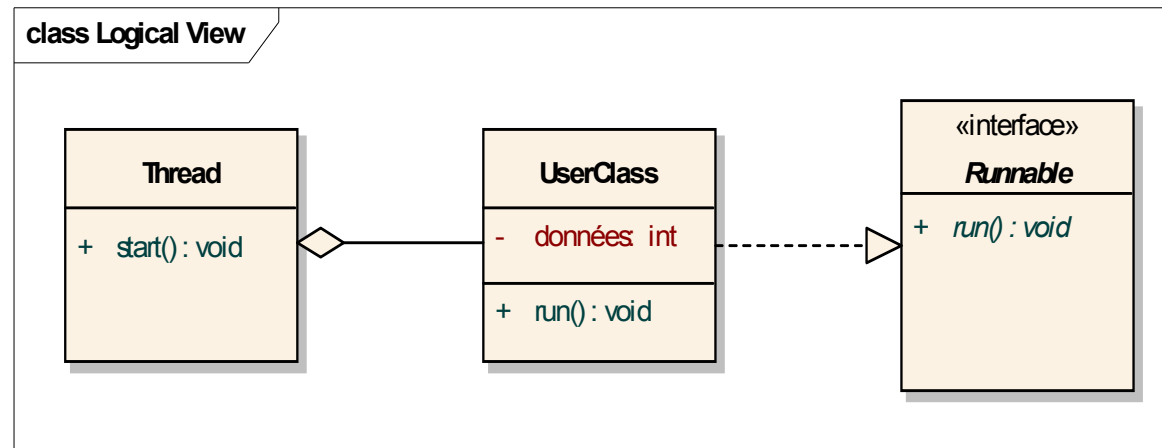
- ThreadGroup : ensembles de threads
 - permet des opérations conjointes sur un ensemble de threads
- Executor : modalités de gestion avancées
 - pool de threads

Classe Thread

Notes

Solution 1

Association Thread - objet



- Code = code de la méthode `run()` de `UserClass`
- Données = instance de la classe `UserClass`
- Thread = instance de la classe `Thread`

`UserClass` : classe définie par l'utilisateur

Solution 1

Notes

Cette solution est illustrée par l'exemple qui suit.

Solution 1

Un thread utilise un objet comme support d'exécution.

- Cet objet doit implémenter l'interface Runnable.

```
public static void main(String args[]) throws InterruptedException {
    Verificateur Gaston = new Verificateur ("Gaston", new Sac (10, '+'));
    Verificateur Robert = new Verificateur ("Robert", new Sac (10, '-'));
    // on construit un thread avec l'objet sur lequel on veut
    // exécuter du code en parallèle
    Thread t1 = new Thread (Gaston);
    Thread t2 = new Thread (Robert);
    t1.start ();
    t2.start ();
}
```

Interface Runnable

```
// héritage libre
// interface imposée
class Verificateur implements Runnable {
    private Sac sac;
    private String nom;
    public Verificateur (String nom, Sac sac) {
        this.nom = nom;
        this.sac = sac;
    }
    // implémentation de Runnable
    public void run () {
        for (int i = 0; i < sac.taille; i++)
            System.out.print (sac.contenu);
        System.out.println ("\n" + nom + " a terminé");
    }
}
```

Solution 1

Notes

Solution 1

Interface Runnable

- Si l'on désire implémenter un thread qui hérite d'une autre classe que `Thread`, on utilisera le constructeur de `Thread` qui prend en paramètre l'objet dont on veut paralléliser le code.

```
public Thread ( Runnable target );
```

- Lorsque le thread est démarré par appel à `start()`, la méthode `run()` de l'objet "embarqué" dans le thread est exécutée. Cette exécution se déroule alors que le code qui suit l'appel à `start()` se déroule également.

```
Z:\supports\Java v2.0\threads>java Thread2
+--+--+--+--+--+--+--+
Robert a termine

Gaston a termine

Z:\supports\Java v2.0\threads>
```

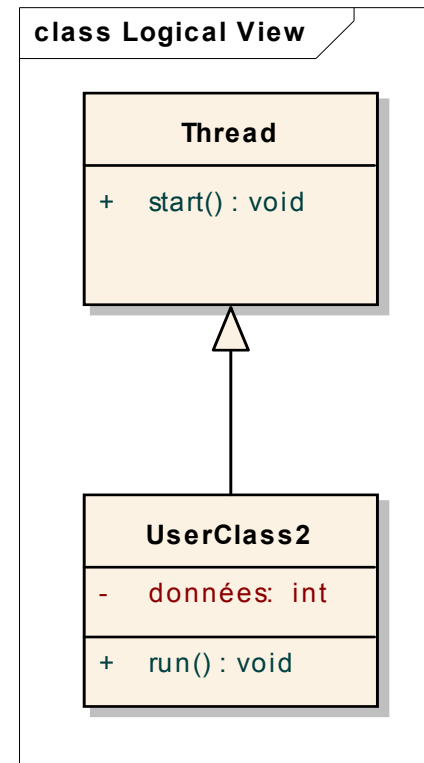
Solution 1

Notes

Solution 2

Sous-classer Thread

- Code = code de la méthode run() de UserClass2
- Données = instance de la classe UserClass2
- Thread = instance de la classe UseClass2



UserClass2 : classe définie par l'utilisateur

Solution 2

Notes

Cette solution est illustré par un exemple dans les pages suivantes.

Solution 2

Sous- classer Thread

```
class Verificateur extends Thread {
    private Sac sac;
    private String nom;
    public Verificateur (String nom, Sac sac) {
        this.nom = nom;
        this.sac = sac;
    }
    public void run () {
        for (int i = 0; i < sac.taille; i++) {
            System.out.print (sac.contenu);
            try {
                sleep (100);
            } catch (Exception e) {}
        }
        System.out.println ("\n" + nom + " a terminé");
    }
}
```

Utilisation

```
public static void main(String args[]) {
    Sac plus = new Sac (10, '+');
    Sac moins= new Sac (10, '-');
    Verificateur Gaston = new Verificateur ("Gaston", plus);
    Verificateur Robert = new Verificateur ("Robert", moins);
    // début du "contrôle"
    Gaston.start (); // appelle run() de Verificateur
    Robert.start (); // appelle run() de Verificateur
}
```

Solution 2

Notes

Solution 2

Sous-classer Thread

Ce code possède un point d'entrée qui est la méthode `run ()`. Dans notre exemple, les méthodes `run ()` des deux objets Robert et Gaston se déroulent en parallèle :

```
Z:\supports\Java v2.0\threads>java Thread2
+---+---+---+---+
Robert a termine

Gaston a termine

Z:\supports\Java v2.0\threads>
```

Utilisation

- `public void start ()` permet de commencer l'exécution d'un thread. La méthode `run ()` de ce thread est alors appelée.
- `start ()` rends la main immédiatement après l'appel.

Solution 2

Notes

Choix d'implémentation d'un thread

Solution recommandée: utiliser `Runnable`

- Permet de bien identifier les rôles
 - Thread = processeur virtuel
 - classe implémentant `Runnable` = code
 - instance de cette classe = données
- Pratique lorsque la super-classe est conceptuellement imposée (exemple : Applet).

```
public class MaClasse extends Base implements Runnable {  
    ...  
    public void run () {  
        // redéfinition de run()  
    }  
    ...  
}
```

Sous-classer `Thread`

- Lorsque l'on veut modifier le fonctionnement d'un thread

Choix d'implémentation d'un thread

Solution recommandée: utiliser Runnable

L'utilisation de Runnable est toujours possible et permet d'identifier parfaitement les rôles. C'est la solution à préconiser.

Sous-classer Thread

Cette solution est plutôt à réserver aux cas où l'on souhaiterait spécialiser la définition d'un thread.

Choix d'implémentation d'un thread

Utiliser une classe anonyme

- Permet de bénéficier des avantages des classes anonymes en matière de partage de données

```
public class Truc {  
    private Type donnée;  
  
    public void uneMéthode() {  
        Runnable travail = new Runnable () {  
            public void run () {  
                // traitements effectués par le thread  
                Truc.this.donnée.uneMéthode ();  
            }  
        };  
        new Thread (travail).start(); // lancement du thread  
    }  
}
```

Solution pratique lorsque la réutilisation est limitée.

Choix d'implémentation d'un thread

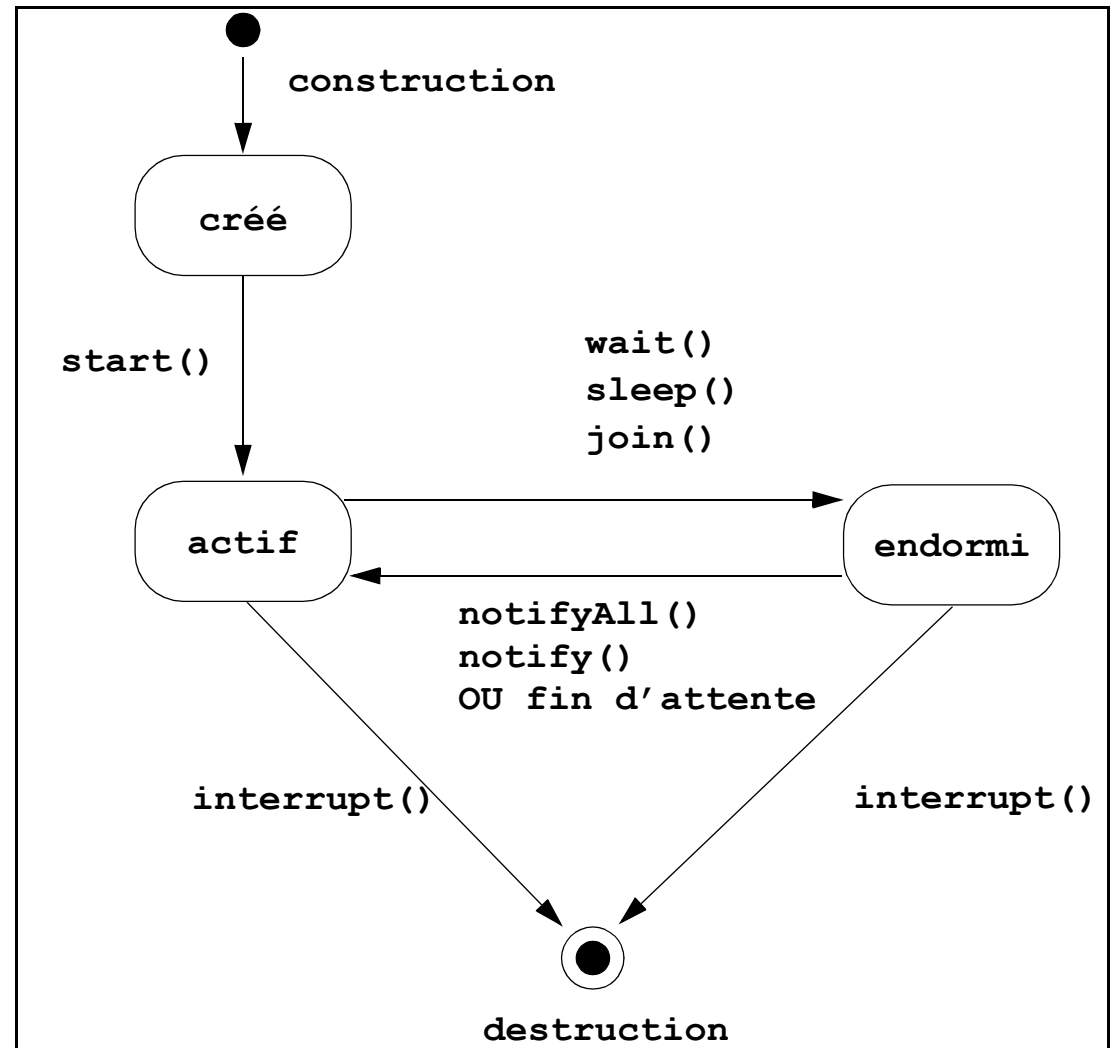
Notes

Les états d'un thread

Transitions

isAlive ()

- à état créé = false
- à l'état actif = true
- à l'état endormi = true
- à l'état mort = false



`stop()`, `suspend()` et `resume()` sont dépréciées.

Les états d'un thread

Créé

Un thread est créé comme n'importe quel objet Java, par appel à un constructeur. Lors de cette étape, aucune autre opération que la réservation mémoire n'est effectuée.

Actif

Le thread est dans cet état lorsqu'après la création, il est activé par `start()`, qui lance la méthode `run()`. Il est alors ajouté à la liste des threads actifs qui sont exécutés en temps partagé.

Endormi

Un thread peut être endormi et réactivé de plusieurs manières :

- Méthodes `wait()` ou `sleep(long milliseconds)`
- Méthode `join()` : attend que le thread termine son exécution.
- une entrée-sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un thread.

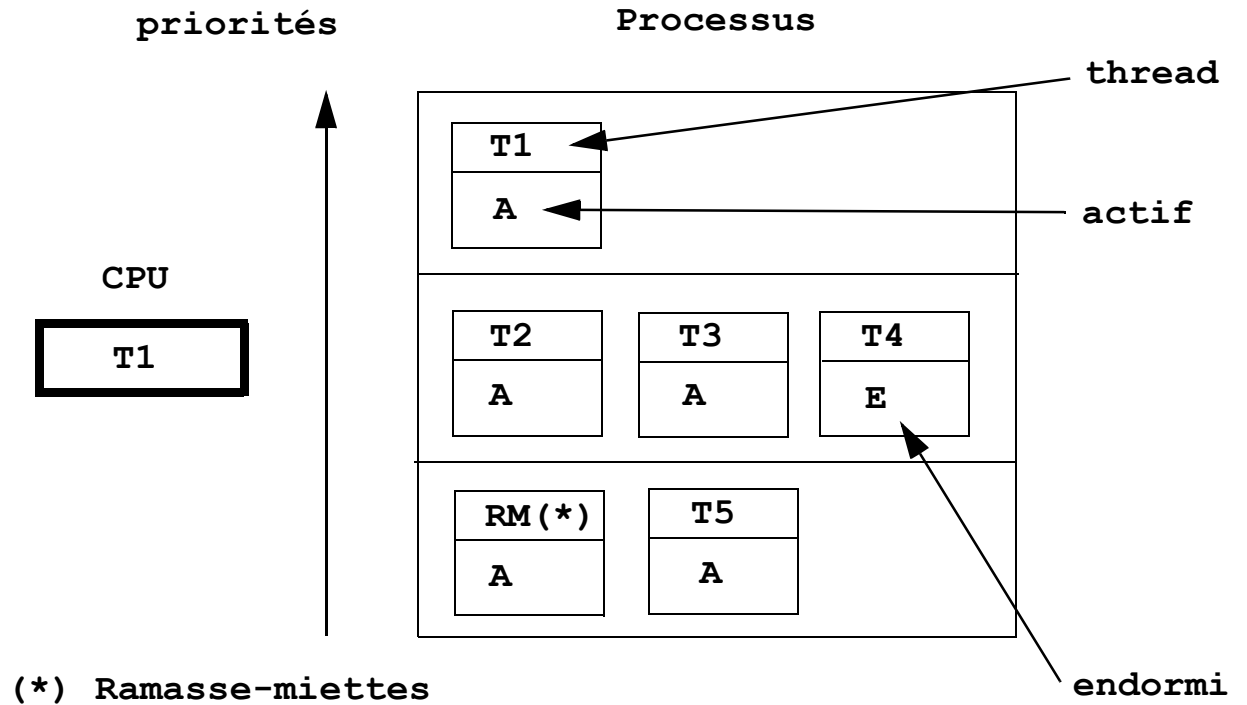
Mort

- Quand la méthode `run()` a terminé son exécution.
- Par un appel explicite à `interrupt()`.

Priorités

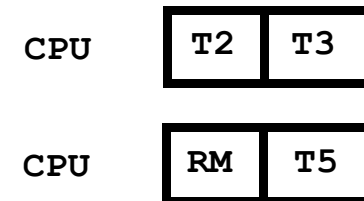
MIN_PRIORITY à MAX_PRIORITY

- Indique une "préférence" d'exécution
- Le comportement n'est pas spécifié
- Peut être ignorée par la JVM



Partage CPU dépendant de la JVM et de l'OS, exemple

- T1 : état Actif --> Endormi
- T2 et T3 : état Actif --> Endormi



Priorités

Notes

<http://java.sun.com/j2se/1.5.0/docs/guide/vm/thread-priorities.html>

Méthodes (classe Thread)

- `void setPriority (int)` modifie la priorité du receveur en fonction du paramètre. Cette priorité ne peut sortir de l'intervalle [`MIN_PRIORITY`, `MAX_PRIORITY`]. Dans le cas contraire, l'exception `IllegalArgumentException` est levée.
- `int getPriority()` permet de connaître la priorité d'un thread.
- Le niveau de priorité "normal" est donné par la constante `NORM_PRIORITY`.

Partage de la ressource CPU

Dépendant de la JVM et du système d'exploitation

- L'ordonnancement des threads et la gestion des priorités ne sont pas spécifiés
- Ils dépendent de
 - la version de la JVM
 - ses choix d'implémentation des threads : threads natifs, "green" threads, lightweight processes, ...

Un thread peut "passer la main"

- Sur action de l'ordonnanceur
- Sur E/S bloquante
- Par appel à Thread.yield()

```
public static void yield ();
```

- D'autres possibilités existent
 - pause avec Thread.sleep()
 - synchronisation avec wait(), notify()

Partage de la ressource CPU

Notes

Le partage de temps CPU entre threads de même priorité est totalement dépendant de la machine virtuelle et du système d'exploitation utilisé.

- Windows95/NT : si le système répartit ce temps équitablement entre threads de même priorité, ces derniers s'exécutent en parallèle sans qu'il y ait besoin de leur demander de partager la ressource CPU.
- Solaris : dans le cas contraire, le premier thread du groupe des threads à priorités égales monopolise la ressource CPU. Il faut qu'il cède cette ressource, soit involontairement en effectuant une entrée-sortie, soit volontairement pour que les autres threads puissent s'exécuter.

Le thread courant cède la ressource CPU à un autre thread de même priorité dans les cas suivants :

- involontairement sur entrée-sortie,
- implicitement sur passage à l'état endormi par `wait()`, `sleep()` ou `suspend()`.
- volontairement par appel à la méthode statique `yield()`. Cette méthode ne permet pas à un thread de priorité inférieure de s'exécuter.

Démons

Caractéristiques

La machine virtuelle peut quitter la méthode `main()` sans attendre la fin des threads de type démon.

- Threads de services :
 - ramasse-miettes,
 - afficheur d'images,
 - ...
- Faible priorité.
- Boucle infinie.

Démons

Principe

- Ce sont principalement des threads de service qui tournent durant toute l'exécution d'un programme Java et qui effectuent des tâches lorsque plus aucun thread plus prioritaire n'est actif.
- Lorsque le programme est terminé, la machine virtuelle Java n'attend pas la mort des démons pour stopper l'exécution. L'arrêt explicite d'un démon n'est donc pas indispensable.

Méthodes

- `setDaemon ()` permet de déclarer un thread comme démon,
- `isDaemon ()` permet de savoir si un thread est un démon.

Groupe de threads

- Groupe de threads par défaut : main.
- Arborescence de Threads et ThreadGroups.
- Opérations récursives.

```
public static void main(String args[]) throws InterruptedException {
    Verificateur Gaston = new Verificateur ("Gaston", new Sac (10, '+'));
    Verificateur Robert = new Verificateur ("Robert", new Sac (10, '-'));
    // création du groupe des "vérificateurs"
    ThreadGroup lesVerificateurs = new ThreadGroup ("Verificateurs");
    Thread t1 = new Thread (lesVerificateurs, Gaston);
    Thread t2 = new Thread (lesVerificateurs, Robert);

    // lancement individuel
    t1.start ();
    t2.start ();

    // opérations globales
    lesVerificateurs.interrupt ();

}
```

Groupes de threads

ThreadGroup

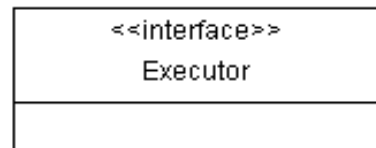
- permet de constituer une arborescence de threads et de groupes de threads,
- offre des méthodes de manipulations récursives d'un ensemble de threads telles que `interrupt()`, `destroy()` ...
- comporte des méthodes qui agissent seulement sur les threads créés ensuite dans un groupe : `setDaemon()`, `setMaxPriority()`.

Fonctionnement

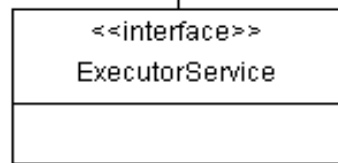
- La machine virtuelle crée au minimum un groupe de threads appelé `main`.
- Par défaut, tout thread appartient au même groupe que son père. Ce sera donc le groupe `main` si l'on n'a jamais précisé de groupe à la création des threads.
- Il est possible de spécifier un `ThreadGroup` à la construction d'un `Thread`.
- Tout thread est capable de connaître le groupe auquel il appartient par l'emploi de `getThreadGroup()`.

Executor

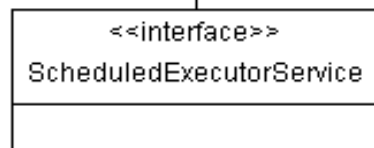
- L'interface `Executor` permet de s'affranchir de l'utilisation de la classe `Thread` pour lancer l'exécution de threads



Permet l'exécution de `Runnable`
`void execute(Runnable command);`



Permet l'exécution de `Runnable` et de `Callable` (équivalent à des tâches)



Pour l'exécution de tâches périodiques (qui se répètent dans le temps) et différées (la tâche doit commencer dans 60 secondes par exemple)

- La classe `Executors` contient plusieurs "fabriques" qui ne renvoient que des executors mono-thread :
 - `Executors.newSingleThreadExecutor()` : `Executor` mono-thread classique
 - `Executors.newSingleThreadScheduledExecutor()` : `Executor` mono-thread pour les tâches périodiques

Executor

Notes

Ensemble d'interfaces / classes / classes abstraites du package `java.util.concurrent` introduite avec la version de 5 de Java.

Executor

```
// Exécution l'aide d'Executors
public class ExempleExecutor {
    public static void main(String[] args) {
        Executor executor = Executors.newSingleThreadExecutor();
        executor.execute(new MonRunnable("Async 1.));
        executor.execute(new MonRunnable("Async 2.));

        //Execute MonRunnable toutes les secondes
        ScheduledExecutorService scheduler = Executors.newSingleThreadScheduledExecutor();
        final ScheduledFuture<> handle = scheduler.scheduleAtFixedRate(new MonRunnable("Scheduled 1")
            , 0 //délai pour la 1ière exécution
            , 1 // périodicité
            , TimeUnit.SECONDS); // unité de temps

        // Arrêt de l'exécution au bout de 10 secondes
        //Création et excécution du ScheduledFutur disponible après un délai
        scheduler.schedule(new Runnable() {
            public void run() { handle.cancel(true); }} // tâche à exécuter
            , 10 //délai de mise à disposition
            , TimeUnit.SECONDS); // unité de temps
    }
}
```

Executor

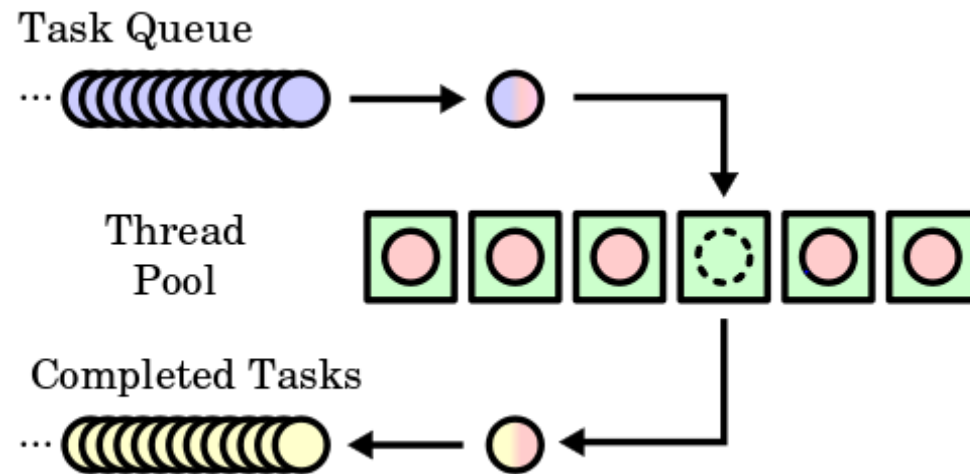
Notes

La classe MonRunnable:

```
// Tâche à exécuter
class MonRunnable implements Runnable{
    String name = "";
    int count = 0;
    MonRunnable(String aName){
        name = aName;
    }
    @Override
    public void run() {
        count++;
        System.out.printf("%1$d - Execution de [ %2$s ] %n", count, name );
    }
}
```

Pool de threads

- Ensemble de threads prêts à être utilisés pour exécuter des tâches en parallèle.
- Evite l'instanciation de threads à chaque fois qu'une tâche doit être exécutée (économie de mémoire et temps)



- Ces pools sont créés via la classe Executors qui définit des fabriques de création de pool de threads:
 - `newFixedThreadPool()` : création d'un pool de taille fixe,
 - `newScheduledThreadPool()` : pool de threads permettant d'exécuter des tâches périodiquement.

Pool de threads

Notes

Schéma provenant du site Wikipedia: "Thread Pool pattern".

Pool de threads

```
public class ExempleExecutorPool {  
    public static void main(String[] args) {  
        List<Runnable> listeRunnable = new ArrayList<Runnable>();  
        listeRunnable.add(new MonRunnable("Async 1."));  
        listeRunnable.add(new MonRunnable("Async 2."));  
        listeRunnable.add(new MonRunnable("Async 3."));  
        listeRunnable.add(new MonRunnable("Async 4."));  
        listeRunnable.add(new MonRunnable("Async 5."));  
  
        ExecutorService executor = Executors.newFixedThreadPool(3);  
        for(Runnable r : listeRunnable){  
            executor.execute(r);  
        }  
        //On ferme l'executor une fois les tâches finies  
        //En effet shutdown va attendre la fin d'exécution des tâches  
        executor.shutdown();  
    }  
}
```

```
1 - Execution de [ Async 1. ][ pool-1-thread-1 ]  
1 - Execution de [ Async 2. ][ pool-1-thread-2 ]  
1 - Execution de [ Async 3. ][ pool-1-thread-3 ]  
1 - Execution de [ Async 4. ][ pool-1-thread-2 ]  
1 - Execution de [ Async 5. ][ pool-1-thread-1 ]
```

Pool de threads

Notes

Chacun des "Runnable" est exécuté par l'un des trois threads du pool.

Java, Programmation avancée

Synchronisation de threads

Version 2.2

- Maîtriser les différentes possibilités de synchronisation et de communication entre threads
- Connaître les classes d'aide à la mise en oeuvre de la synchronisation: Lock, Semaphore, Collections synchronisées

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Synchronisation

wait()

```
Object verrou = new Object ();
try {
    verrou.wait() ; // bloque Thread.currentThread()
} catch (InterruptedException e) {
}
```

- Permet de mettre en attente le thread qui exécute le code contenant l'appel à la méthode wait().
- L'objet receveur du message wait() joue le rôle d'un verrou

Comment a-t-on la garantie qu'un seul thread sera bloqué ?

notify()

- Permet de libérer le premier thread bloqué par wait() sur le même objet receveur

notifyAll()

- Permet de libérer tous les threads bloqués par wait() sur le même verrou

```
try {
    verrou.notifyAll() ;
} catch (InterruptedException e) {
}
```

Synchronisation

Notes

Synchronisation

Modificateur synchronized

- Méthode 1

```
//  
// la méthode est une section critique  
// pour le receveur  
//  
public synchronized void uneMethode() {  
    // corps  
}
```

- Méthode 2

```
public void uneMethode() {  
    // ..  
    // acquisition d'un verrou sur l'objet  
    // début  
    synchronized (objet) {  
        objet.methode();  
    }  
    // fin  
}
```


Synchronisation

Modificateur `synchronized`

La programmation par threads impose souvent de disposer de mécanismes permettant de contrôler les concurrences d'accès à des données partagées.

En Java, deux syntaxes permettent de mettre en oeuvre un mécanisme de gestion de verrous:

- Définir une méthode comme étant `synchronized`:
 - lorsqu'un thread déroule cette méthode sur un objet, un autre thread ne peut pas exécuter toute autre méthode `synchronized` pour le même objet.
 - par contre, il peut exécuter la méthode en parallèle pour un autre objet.
- Restreindre l'utilisation d'un objet pour un thread à la fois par `synchronized (objet) { ... }`:
 - l'accès à l'objet en paramètre est réservé à un thread et un seul. Tout autre thread tentant d'acquérir un verrou sur le même objet est mis en attente.

Pour éviter une dégradation de performances, il faut que les sections critiques soient courtes et utilisées à bon escient.

La seconde méthode d'acquisition de verrou permet une délimitation plus fine de la section critique.

Synchronisation

Modificateur synchronized

```
public synchronized void uneMethode() {  
    // corps  
}
```

- est équivalent à

```
public void uneMethode() {  
    synchronized (this) {  
        // corps  
    }  
}
```

Synchronisation

Notes

Synchronisation

verrou.wait() impose synchronized (verrou)

```
Object verrou = new Object ();
synchronized (verrou) {
    try {
        verrou.wait() ; // bloque l'unique thread qui est entré
                        // dans la section critique
    } catch (InterruptedException e) {
    }
}
```

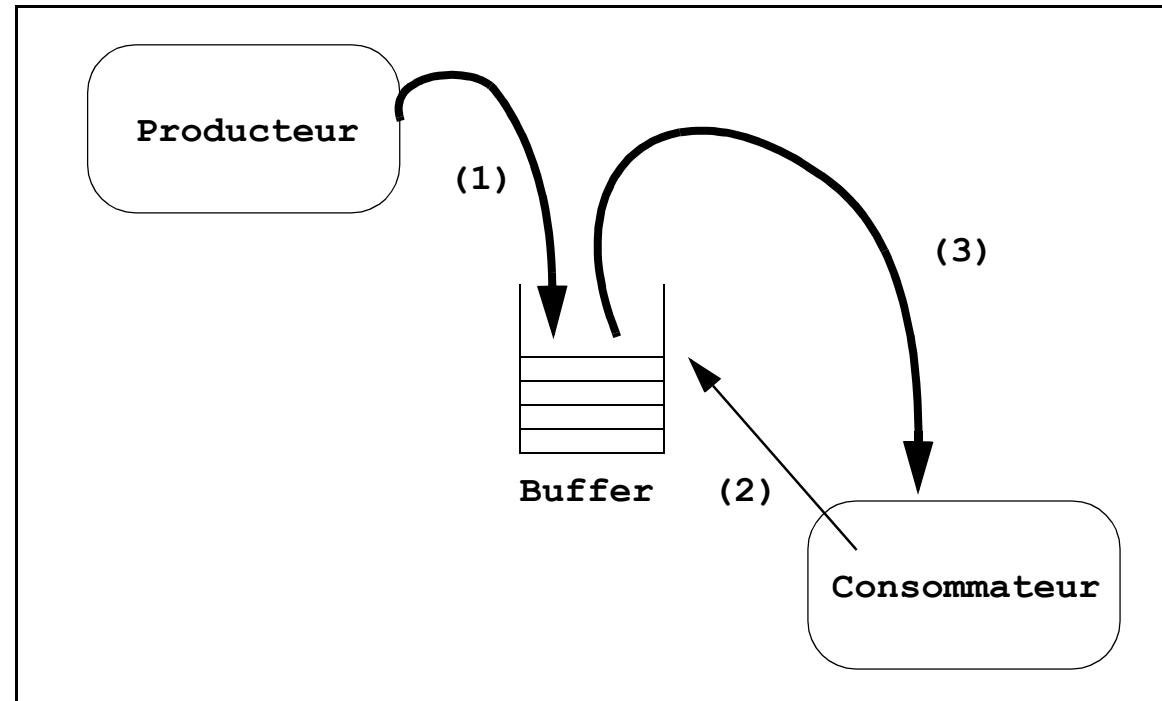
Attention aux blocages et inter-blocages.

Synchronisation

Notes

Rendez-vous

Exemple Producteur - Consommateur



1. le Producteur empile des données dans le Buffer,
2. le Consommateur demande au Buffer de lui donner une information,
3. lorsqu'une information est disponible, le Consommateur la prend.

Rendez-vous

Exemple Producteur - Consommateur

- le producteur et le consommateur sont deux threads qui s'exécutent en parallèle,
- ils possèdent chacun leur propre rythme de production ou de consommation,
- si le buffer est vide, le consommateur est mis en attente jusqu'à ce que le producteur y dépose quelque chose.

Rendez-vous

Producteur

```
class Producteur extends Thread {
    private Buffer buffer;
    private String donnee;

    public Producteur (Buffer buffer, String donnee) {
        this.buffer = buffer;
        this.donnee = donnee;
    }
    // redéfinition de run()
    public void run () {
        for (int i = 0; i < donnee.length (); i++) {
            // dépôt des données
            buffer.poser (donnee.charAt (i));
            try {
                // rythme de production aléatoire
                sleep ((int) (Math.random() * 25));
            } catch (Exception e) {}
        }
        System.out.println ("\nProduction terminée");
    }
}
```


Rendez-vous

Producteur

- La méthode `run()` du producteur dépose chaque caractère de la chaîne donnée dans le buffer en activant la méthode `poser()` sur celui-ci.
- Si le buffer est plein (2 éléments dans cet exemple), la méthode `poser()` :
 - met le thread appelant en attente avec `wait()`,
 - le libère avec `notify()` quand un espace est disponible, c'est-à-dire quand le consommateur est venu prendre un élément.
- Le buffer est une sous-classe de `java.lang.Stack`, qui implémente la notion de pile `First In Last Out`

Rendez-vous

poser()

```
class Buffer extends java.util.Stack {  
    public synchronized void poser (char donnee) {  
        // attente tant que le buffer est plein  
        while (full()) {  
            try {  
                wait (); // mise en attente  
            } catch (Exception e) {}  
        }  
        // il y a une place libre  
        push (new Character (donnee));  
        notify (); // fin de mise en attente  
    }  
}
```

Rendez-vous

Notes

Rendez-vous

Consommateur

```
class Consommateur extends Thread {
    private Buffer buffer;
    private int nb;

    public Consommateur (Buffer buffer, int nb) {
        this.buffer = buffer;
        this.nb = nb;
    }
    public void run () {
        // boucle de lecture des données
        for (int i = 0; i < nb; i++) {
            char car = buffer.prendre ();
            System.out.print (car);
            try { // rythme de consommation aléatoire
                sleep ((int)(Math.random() * 500));
            } catch (Exception e) {}
        }
        System.out.println ("\nConsommation terminée");
    }
}
```

Rendez-vous

Consommateur

- La méthode `run()` du consommateur effectue une itération afin de prendre dans le buffer tous les caractères. Pour cela, il invoque la méthode `prendre()` sur le buffer.
- Si le buffer est vide, le buffer met le consommateur (thread appelant) en attente jusqu'à ce que le producteur pose un nouvel élément. Il le libère alors par `notify()`.

Rendez-vous

prendre ()

```
class Buffer extends java.util.Stack {  
    // ...  
    public synchronized char prendre () {  
        // attente tant que le buffer est vide  
        while (empty()) {  
            try { // mise en attente  
                wait ();  
            } catch (Exception e) {}  
        }  
        notify (); // libération  
        // retour de la donnée  
        return ((Character)pop()).charValue();  
    }  
}
```

Rendez-vous

Notes

Rendez-vous

Main

```
public static void main (String argv[])
{
    String donnee = "Rendez-vous avec Java";
    Buffer buffer = new Buffer ();
    Producteur producteur = new Producteur (buffer, donnee);
    Consommateur conso = new Consommateur (buffer, donnee.length());

    producteur.start ();          // début du thread producteur
    conso.start ();              // début du thread consommateur
}
```

Exécution



le n de "Rendez" a été produit avant que le e soit consommé

Rendez-vous

Buffer

```
class Buffer extends java.util.Stack {
    public synchronized void poser (char donnee) {
        // attente tant que le buffer est plein
        while (full()) {
            try {
                wait (); // mise en attente
            } catch (Exception e) {}
        }
        // il y a une place libre
        push (new Character (donnee));
        notify (); // fin de mise en attente
    }
    public synchronized char prendre () {
        // attente tant que le buffer est vide
        while (empty()) {
            try { // mise en attente
                wait ();
            } catch (Exception e) {}
        }
        notify (); // libération
        // retour de la donnée
        return ((Character)pop()).charValue();
    }
    public boolean full() {
        return (size() == 2);
    }
}
```

Buffer est une sous-classe de Stack, ce qui explique le comportement mis en évidence à l'exécution.

Interface Lock

- Met à disposition d'objets "haut-niveau" de gestion des verrous depuis la version 5 de Java
- Permet de gérer des conditions différentes (interface Condition) de libération en utilisant un même verrou
- Utilisation de l'interface Lock

```
Lock l = ...t
l.lock(); // demande de verrou - attende si non possible
try {
    // accès à la ressource protégée par le verrou
} finally {
    l.unlock(); //libère le verrou
}
```

Classe ReentrantLock

- Implémentation de l'interface Lock
- Se comporte comme un verrou standard : si un thread a acquis un verrou sur un objet, il a accès toutes les sections critiques (synchronized)

```
//....
public synchronized outer() {
    inner();
    //other things
}
public synchronized inner() {
    //other things
}
}
```

Interface Lock

Notes

Interfaces / classes du package `java.util.concurrent.locks`

Exemple d'utilisation d'un Lock

```
class BufferLock extends java.util.Stack<Character> {
    final Lock lock = new ReentrantLock();
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();
    public boolean full() { return (size() == 2); }
    public void poser(char donnee) throws InterruptedException {
        lock.lock(); //demande verrou
        try {
            while (full()) notFull.await(); // attente tant que le buffer est plein
            push(new Character(donnee)); // il y a une place libre
            notEmpty.signal(); // réveille du thread en attente sur le condition notEmpty
        } finally {
            lock.unlock(); //libération du verrou
        }
    }
    public char prendre() throws InterruptedException {
        lock.lock();
        try{
            while (empty()) notEmpty.await(); // attente tant que le buffer est vide
            char c = pop().charValue();
            notFull.signal();// réveille du thread en attente sur le condition notFull
            return (c); // retour de la donnee
        } finally { lock.unlock();}
    } }
}
```

Exemple d'utilisation d'un Lock

Notes

Les sémaphores

- Définit un nombre fixé d'autorisations d'accès à une ressource en simultané (nombre entier)
- Un thread qui souhaite accéder à cette ressource doit demander une autorisation à ce sémaphore, qui peut accorder l'accès ou pas,

Méthodes

- `acquire()` et `acquire(int permits)` : demande une ou plusieurs autorisations. Méthodes bloquantes, ne rendent la main que lorsque le nombre d'autorisations demandé est disponible.
- `tryAcquire()` et `tryAcquire(int permits)` : demande une ou plusieurs autorisations. Ces deux méthodes rendent immédiatement la main, et retournent `true` ou `false`, suivant que le bon nombre de permis ait été octroyé ou non.
- `release()` et `release(int permits)` : rend une autorisation, ou le nombre d'autorisations passées en paramètre.

Les sémaphores

Notes

Exemple d'utilisation d'un sémaphore

```
class CollectionTailleMax<T> {  
    // création de la collection et d'un sémaphore  
    private Collection<T> collection ;  
    private Semaphore semaphore ;  
  
    public CollectionTailleMax(int tailleMax) {  
        // initialisation d'un collection synchronisée  
        this.collection = Collections.synchronizedCollection(new ArrayList<T>(tailleMax)) ;  
        this.semaphore = new Semaphore(tailleMax) ;  
    }  
    public boolean add(T t) throws InterruptedException {  
        // ajout d'un objet, demande d'une autorisation auprès du sémaphore  
        // cette méthode est bloquante, s'il n'y a pas d'autorisation disponible on attend...  
        semaphore.acquire() ;  
        collection.add(t) ;  
        return true ;  
    }  
    public boolean remove(Object o) throws InterruptedException {  
        // on commence par vérifier que l'objet passé est bien retiré de la collection  
        boolean removed = collection.remove(o) ;  
        if (removed) { // dans ce cas on rend une autorisation au sémaphore  
            semaphore.release() ;  
        }  
        return removed ;  
    }  
}
```


Exemple d'utilisation d'un sémaphore

Notes

Exemple de limitation de la taille d'une collection à l'aide d'un sémaphore

Les collections synchronisées

Problématique

```
public class ConcurrentAccessCollection {  
    // création d'une liste classique  
    private List<String> liste = new ArrayList<String>() ;  
  
    public String getLastString() {  
        // on détermine l'index du dernier élément  
        int lastIndex = liste.size() - 1 ;  
        return liste.get(lastIndex) ;  
    }  
    public boolean remove(String str) {  
        return liste.remove(str) ;  
    }  
}
```

Que se passe-t'il si 2 threads accèdent à la méthode getLastString() en même temps?

Les collections synchronisées

Notes

Les collections synchronisées

Solutions

- Utilisation de collections dites "thread-safe": `Vector` et `Hashtable` par exemple
- Transformation d'une Collection non "thread-safe" en collection "thread-safe" à l'aide de la classes `Collections` et ses méthodes `Set<T> synchronizedSet(Set<T> set)` ou `Map<K, V> synchronizedMap(Map<K, V> map)`
- Utilisation des collections concurrentes `CopyOnWriteArrayList` et `CopyOnWriteArraySet`
 - Opérations de modifications réalisent des copies de l'ensemble du tableau
 - Opérations de lecture se font sur un unique tableau, en lecture seule.
- Utilisation de tables de hashage concurrentes implémentant l'interface `ConcurrentMap<K, V>`
 - Les lectures sont non-bloquantes mais les opérations de modification sont contrôlées
- Implémentations synchronisées des interfaces `SortedMap` et `SortedSet`: `ConcurrentSkipListSet<K, V>` et `ConcurrentSkipListMap<K, V>`.

Les collections synchronisées

Notes

Attention les Set, List, Map retournées par les méthodes synchronizedXXX(XXX arg) : l'utilisation des méthodes de manipulation de la collection n'ont plus besoin d'être dans une section critique sauf lors du parcours de l'ensemble, il doit être dans un bloc synchronized.

Les files d'attentes

- Interface `BlockingQueue<E>` pour la mise en oeuvre de files d'attente à accès concurrents
- En plus des méthodes de l'interface `Queue<E>` elle offre les fonctionnalités suivantes:
 - ajout d'un élément - méthode `put(E e)`, ou retrait d'un élément - méthode `take()` : bloquent si la file d'attente est saturée, ou si elle est vide. Ces deux méthodes attendent en fait la fin de la saturation, ou l'arrivée d'un nouvel élément pour rendre la main.
 - `offer(E e, long time, TimeUnit unit)` et `poll(long time, TimeUnit unit)` qui ont le même fonctionnement, sauf qu'elles ne peuvent attendre plus longtemps que le temps passé en paramètre. Quand ce temps est écoulé, elles retournent `false` ou `null` respectivement.
- Implémentations:
 - `ArrayBlockingQueue` et `LinkedBlockingQueue`: FIFO qui peuvent définir une capacité maximale.
 - `PriorityBlockingQueue` conserve ses éléments ordonnés
 - `SynchronousQueue` : sa capacité est de 0. On ne peut donc pas l'interroger sur son contenu, ni itérer sur ses éléments, tout simplement parce qu'elle n'en possède pas. En fait, chaque insertion d'une nouvelle donnée dans une telle file, doit attendre qu'une demande de retrait soit faite pour rendre la main. Réciproquement, chaque demande de retrait ne rend la main que lorsqu'une demande d'insertion est faite. Cette classe permet à deux threads de se synchroniser sur des points de rendez-vous particuliers.

Les files d'attentes

Notes

Les files d'attentes

```
public class Buffer {  
    // file d'attente de 2 cases  
    private BlockingQueue<Character> queue = new ArrayBlockingQueue<Character>(2) ;  
  
    // on peut y dposer du pain, mais le boulanger n'est pas patient  
    // pose le caractère q'il y a de la place sinon attend qu'une place se libère  
    public void poser(char donnee) throws InterruptedException {  
        queue.put(donnee) ;  
    }  
    // récupération d'un caractère s'il en existe sinon attend  
    public char prendre() throws InterruptedException {  
        return queue.take() ;  
    }  
}
```

- Les méthodes poser et prendre n'ont plus besoin de définir de section critiques (synchronized).

Les files d'attente

Notes

Java, Programmation avancée

Mécanismes avancés des threads Java

Version 2.2

- Java Memory Model

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Threads et variables

Concepts

- Variable = variable de classe ou d'instance, résidant en mémoire partagée par tous les threads
- Copie de travail = copie d'une variable résidant dans la mémoire de travail d'un thread
- Lors de son exécution, un thread utilise la copie de travail d'une variable
- Il faut des règles de synchronisation
 - mémoire principale partagée -> copie de travail
 - copie de travail -> mémoire principale partagée

Spécifications JVM, chapitre 8.

Java Memory Model, règles principales

- Les échanges entre mémoire principale partagée et mémoire de travail d'un thread sont décomposés en actions atomiques : use, read, load, store, lock, ...
- Deux threads échangent via la mémoire principale seulement
- Les actions atomiques sont ordonnées
- Deux actions atomiques identiques ne peuvent se suivre

Threads et variables

Notes

Threads et variables

Actions à l'initiative d'un thread

- use : transfert de la valeur de la copie de travail d'une variable en zone d'exécution
- assign : transfert d'une valeur en zone d'exécution vers une copie de travail
- load : mise-à-jour d'une copie à partir d'une valeur transmise depuis la mémoire centrale, consécutive à un "read"
- store : mise à disposition de la mémoire centrale d'une valeur de copie de travail en vue d'un "write"

Actions à l'initiative de la mémoire principale

- read : transfert de la valeur principale d'une variable en mémoire de travail pour un "load"
- write : affectation de la valeur principale d'une variable en fonction de la valeur transmise par "store"

Actions

- lock : acquisition d'un verrou
- unlock : restitution d'un verrou

Threads et variables

Notes

Java, Programmation avancée

Introduction à JMX

Version 2.2

- Présentation de JMX
- Architecture
- Outils
- Agents JMX
- ...

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Java Management Extensions

Apparue dans la version 5.0 de JavaSE

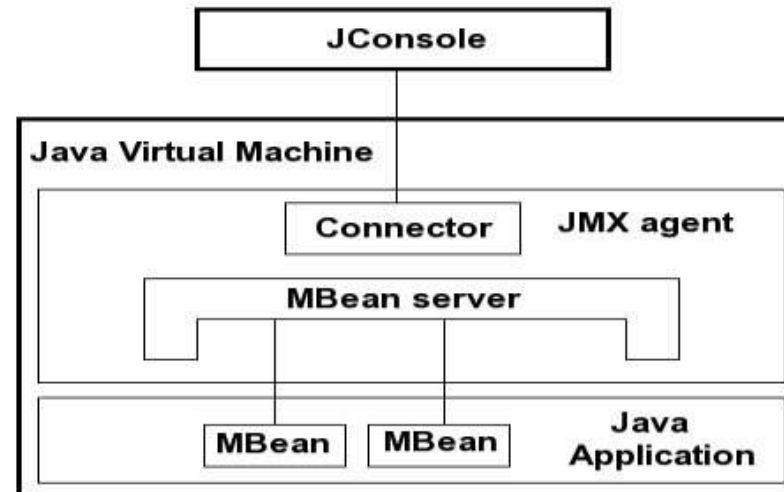
- Moyen simple et standard de gérer des ressources :
 - gestion d'applications, de matériel, de services
 - monitoring et controle à chaud et en temps réel
 - gestion locale ou à distance.
- JMX est liée à deux autres spécifications :
 - Java Management Extensions Instrumentisation and Agent Specification (JSR 3)
 - Java Management Extensions Remote API (JSR 160)

Java Management Extensions

Notes

Java Management Extensions

- Contenu de la spécification :
 - Architecture
 - Design patterns
 - APIs
 - Services
- Permet de monitorer des ressources via des objets Java nommés **Managed beans**, ou **MBean**.
- Les MBeans sont enregistrés, dans la JVM, au sein d'un **serveur de MBeans**.
- Des connecteurs permettent à des outils de contrôle (console) d'accéder au serveur de MBeans et de communiquer avec les MBeans.



Java Management Extensions

Notes

Java Management Extensions

Avantages de JMX

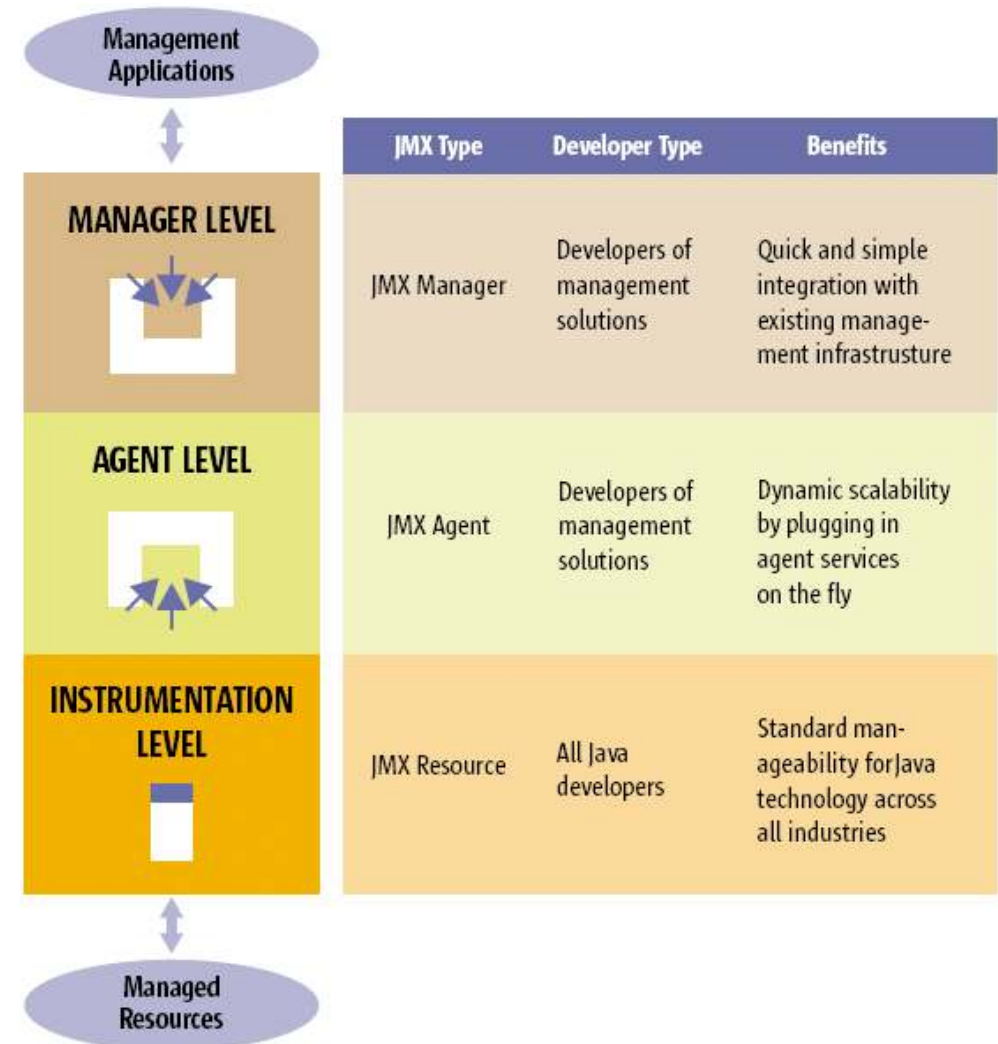
- Contrôle et monitoring d'applications Java sans outils tiers
- Technologie standard Java
- Gestion distante d'une JVM
- Architecture scalable
- Basée sur les technologies Java existantes
- Intégration aisée avec des logiciels tiers

Java Management Extensions

Notes

Architecture

- L'architecture JMX est découpée en 3 niveaux :
 - **Instrumentation** : MBeans et ressources associées
 - **Agents** : exposent les MBeans
 - **Gestion distante** : connection au serveur JMS pour monitoring / controle.



Architecture

Notes

Le niveau instrumentation

- Toute ressources contrôlée par JMX doit être disponible sous la forme d'un objet Java.
- Un objet Java représentant une ressource est appelé **MBean** et doit être développé suivant la spécification JMX (JSR 3).
- Il existe deux types de MBeans :
 - **MBean statique** : développé suivant la spécification JavaBeans
 - **MBean dynamique** : conforme à une interface spécifique, offrant une plus grande flexibilité d'exécution.
- Les MBeans sont rendus accessibles par des **agents**, mais les MBeans n'ont pas besoin de connaître les agents qui les exposeront.

Le niveau instrumentation

Notes

Agents JMX

Caractéristiques

- Un **agent** est une entité existant au sein d'une JVM et permettant la liaison entre des MBeans et une application de gestion.
- Le composant essentiel de l'agent JMX est le **serveur de MBean**. Il fait office d'annuaire ; tout MBean enregistré auprès du serveur devient visible (via son interface) aux outils de gestion.
- Les MBeans peuvent être instanciés et enregistrés auprès d'un serveur :
 - par un autre MBean,
 - par l'agent,
 - par une application de gestion distante.
- Tout MBean enregistré auprès de l'annuaire se voit attribuer un nom unique. Une application de gestion utilise ce nom pour retrouver une référence sur le MBean. Les opérations que l'on peut effectuer sur des MBeans sont
 - récupération des interfaces de gestion
 - lecture / écriture des valeurs des attributs
 - invocation d'opérations
 - réception de notifications
 - Requêtage

Agents JMX

Notes

Agents JMX

Services d'agent

- Les agents de service sont des objets effectuant des opérations de gestion sur les MBeans enregistrés auprès du serveur de MBean.
- Les services d'agent sont souvent eux mêmes disponibles sous forme de MBeans.
- La spécification JMX définit les services d'agent suivants :
 - le **service de chargement de classes dynamique**, à travers l'applet de gestion (m-let) permet de récupérer et d'instancier de nouvelles classes et des bibliothèques natives téléchargement dynamiquement à partir du réseau.
 - les **monitors** surveillent la valeur de certains attributs de MBeans et peuvent notifier d'autres objets en cas de changement de valeur.
 - les **Timers** fournissent un mécanisme d'ordonnancement et peuvent emettre des notification à intervalles réguliers.
 - Le **service de relation** définit des associations entre MBeans et maintient la cohérence de la relation.

Agents JMX

Notes

Adapteurs et connecteurs

- Les **adaptateurs** et **connecteurs** permettent de rendre un agent accessible à distance pour les applications de gestion.
- Tout agent JMX doit inclure au moins un adaptateur ou un connecteur pour être utilisable.
- La plateforme JavaSE inclut le connecteur RMI.

Adaptateurs

- Un adaptateur permet de rendre un agent JMX visible à travers un protocole (ex SNMP).
- Le modèle des MBean et du serveur de MBean n'est pas forcément respecté, il est adapté au protocole.

Connecteurs

- Les connecteurs permettent à un client JMX de se connecter à un serveur JMX.
- Un connecteur est spécifique à un certain protocole, mais l'application de gestion peut utiliser n'importe quel connecteur de façon transparente car ils sont tous conformes à la même interface.

Adapteurs et connecteurs

Notes

Monitoring d'une JVM avec JMX

Agent JMX d'une JVM

- Depuis JavaSE 5.0, toute JVM inclut un agent JMX, couplé à un connecteur RMI, ainsi qu'un certain nombres de MBeans standards.
- En JavaSE 5.0, l'agent JMX doit être explicitement activé :
 - Ajouter la propriété `com.sun.management.jmxremote` valorisée à `true` lors du lancement de la JVM. Il est possible de spécifier d'autres propriétés :

Nom de la propriété	Valeurs / valeur par défaut
<code>com.sun.management.jmxremote</code>	<code>true / false</code>
<code>com.sun.management.jmxremote.port</code>	
<code>com.sun.management.jmxremote.ssl</code>	<code>true / false</code> ; <code>true</code> par défaut
<code>com.sun.management.jmxremote.ssl.enabled.protocols</code>	par défaut : <code>SSL/TLS</code>
<code>com.sun.management.jmxremote.ssl.enabled.cipher.suite</code>	par défaut : <code>ciphers SSL/TLS</code>
<code>com.sun.management.jmxremote.ssl.need.client.auth</code>	<code>true / false</code> ; <code>false</code> par défaut
<code>com.sun.management.jmxremote.authenticate</code>	<code>true / false</code> ; <code>true</code> par défaut
<code>com.sun.management.jmxremote.password.file</code>	<code>JRE_HOME/lib/management/jmxremote.password</code>
<code>com.sun.management.jmxremote.access.file</code>	<code>JRE_HOME/lib/management/jmxremote.access</code>
<code>com.sun.management.jmxremote.login.config</code>	Configuration JAAS

- En JavaSE 6.0, l'agent JMX est activé par défaut.

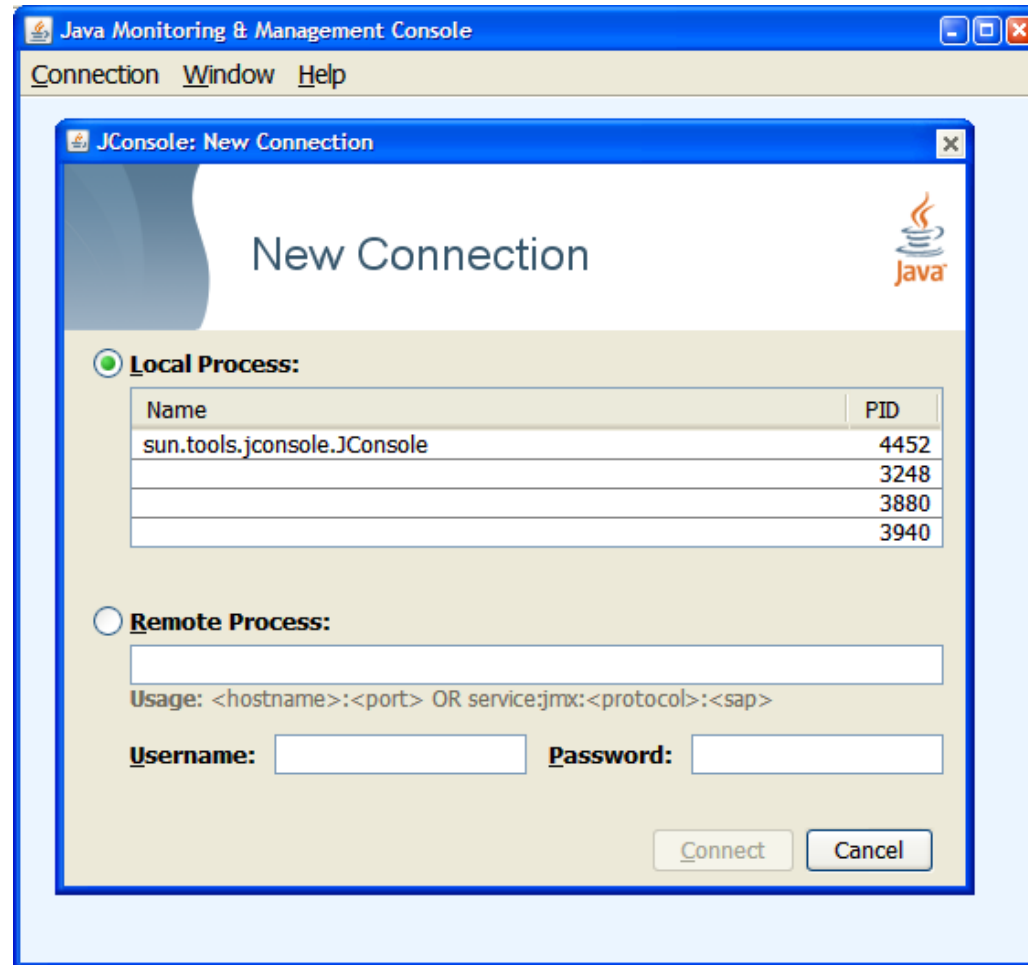
Monitoring d'une JVM avec JMX

Notes

Monitoring d'une JVM avec JMX

JConsole

- Le JDK fournit une application, la JConsole, permettant de se connecter à un agent JMX local ou distant.



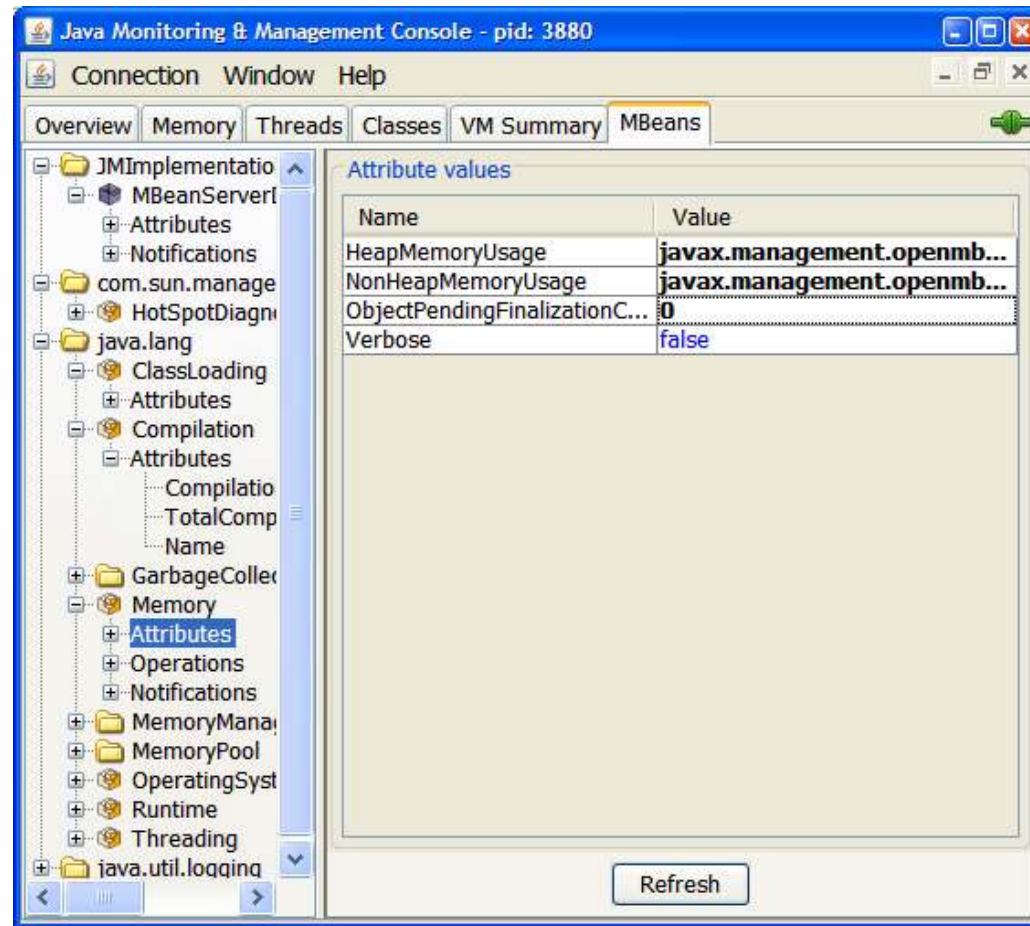
Monitoring d'une JVM avec JMX

Notes

Monitoring d'une JVM avec JMX

JConsole

- La JConsole inclut un navigateur de MBeans à partir duquel il est possible de lire / écrire les attributs des MBeans, invoquer leurs méthodes, récupérer leur notifications.



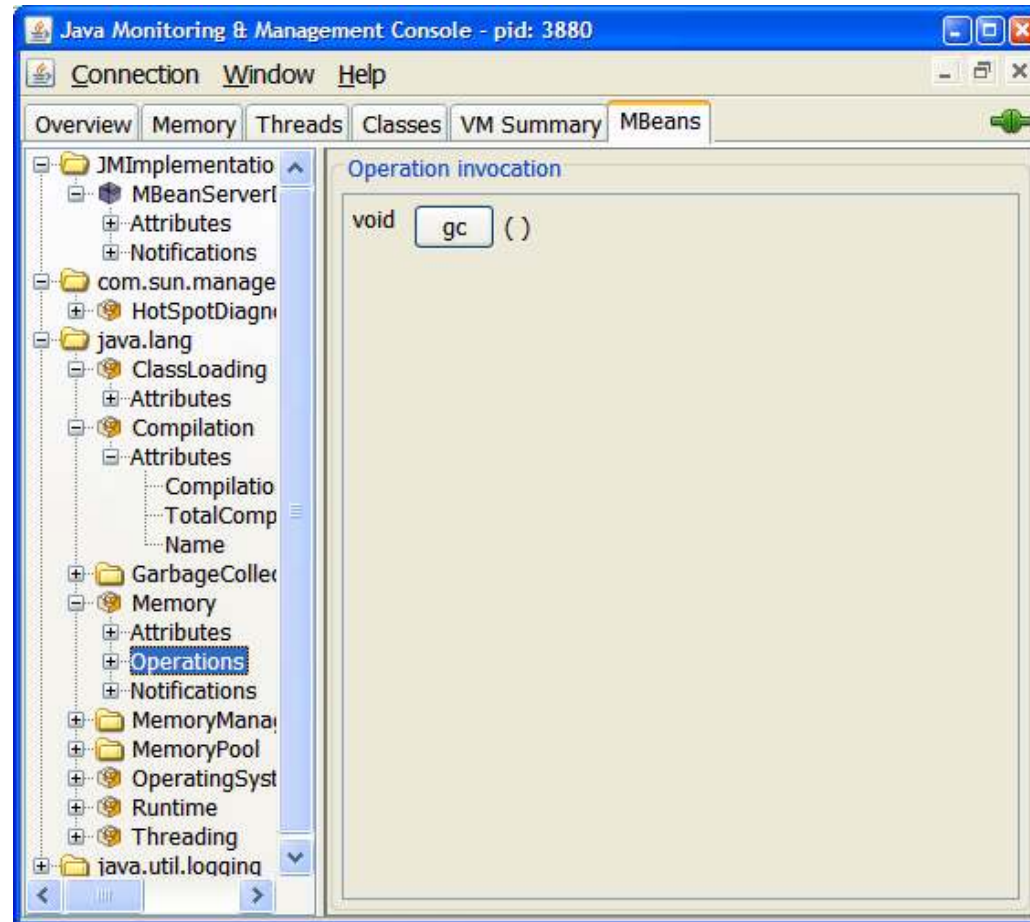
Monitoring d'une JVM avec JMX

Notes

Monitoring d'une JVM avec JMX

JConsole

- Exemple d'invocation de méthode d'un MBean :



Monitoring d'une JVM avec JMX

Notes

Java, Programmation avancée

L'essentiel de JMX

Version 2.2

- Développer des MBeans
- Envoyer des notifications

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Management Beans

- Un MBean est similaire à un Java Bean standard.
- Un MBean peut représenter un périphérique, une application ou toute ressource nécessitant d'être surveillée.
- Un MBean expose une interface de gestion : un ensemble de propriétés, accessibles en lecture et/ou écriture, ainsi qu'un certain nombre de méthodes invocables.
- Les MBeans peuvent également déclencher des notifications lors de divers événements.
- La spécification JMX définit 4 types de MBeans :
 - standard MBeans
 - dynamic MBeans
 - open MBeans
 - model MBeans

Management Beans

Notes

Standard MBeans

- Un Standard MBean est défini par :
 - une interface Java, nommée `QuelquechoseMBean`
 - une classe Java, nommée `Quelquechose` qui implémente cette interface.
- Toute méthode de l'interface définit soit un attribut, soit une opération du MBean.
- Par défaut, chaque méthode correspond à une opération.
- Attributs et opérations sont des méthodes dont la signature respecte un certain modèle.

Standard MBeans

Notes

Standard MBeans

Interface du MBean

- Par convention, l'interface du MBean prend le nom de son implémentation, suffixé par MBean.
- L'interface d'un MBean est composée :
 - d'attributs en lecture et/ou écriture, représentés par des getters / setters
 - de méthodes, qui seront celles invocables à distance.

```
public interface HelloMBean {  
  
    public void sayHello();  
    public int add(int x, int y);  
  
    public String getName();  
  
    public int getCacheSize();  
    public void setCacheSize(int size);  
  
}
```


Standard MBeans

Notes

Standard MBeans

Implémentation du MBean

```
public class Hello implements HelloMBean {
    private final String name = "Leuville Objects";
    private int cacheSize = 200;

    public void sayHello() {
        System.out.println("hello, world");
    }
    public int add(int x, int y) {
        return x + y;
    }
    public String getName() {
        return this.name;
    }
    public int getCacheSize() {
        return this.cacheSize;
    }
    public synchronized void setCacheSize(int size) {
        this.cacheSize = size;
    }
}
```

Standard MBeans

Notes

Enregistrement d'un MBean

- Pour être utilisé, un MBean doit être géré par un agent JMX.
- Le MBean doit être enregistré auprès du serveur JMX de l'agent.
- Un serveur JMX est représenté par l'interface `javax.management.MBeanServer`.

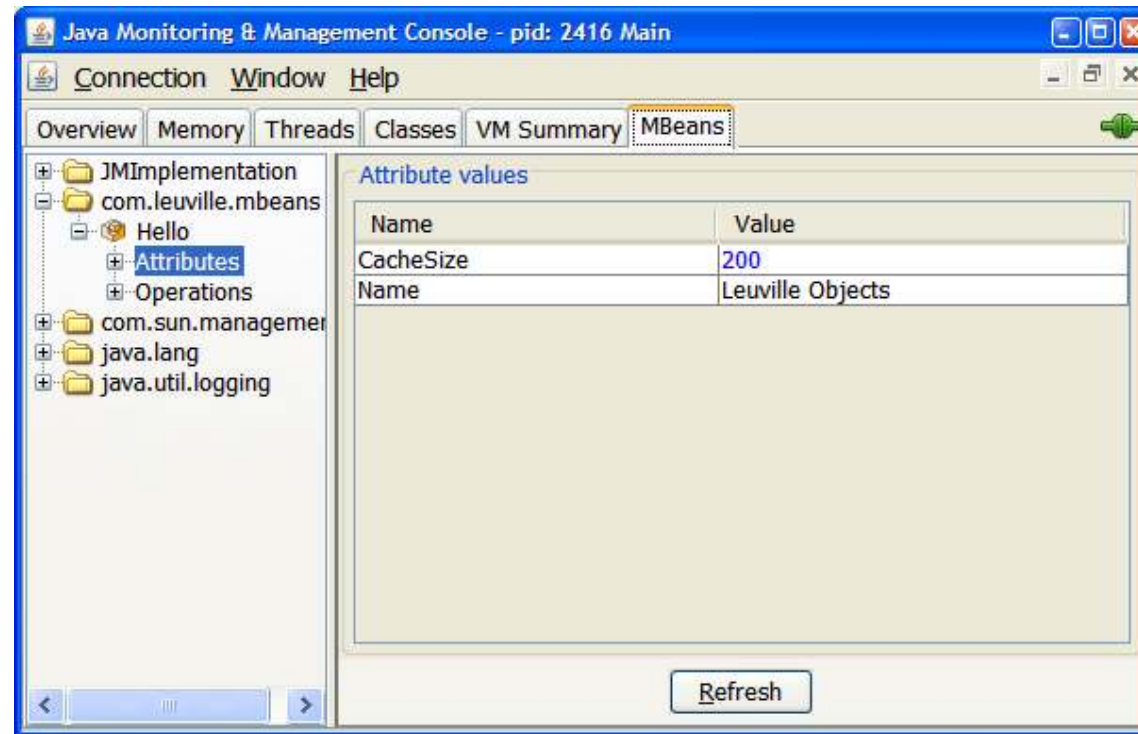
```
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
ObjectName name = new ObjectName("com.leuville.mbeans:type=Hello");
Hello mbean = new Hello();
mbs.registerMBean(mbean, name);
```

- La classe `java.lang.management.ManagementFactory` permet de gérer les serveurs JMX.
 - La méthode `getPlatformMBeanServer` permet de récupérer un serveur JMX disponible dans l'environnement.
 - Si aucun serveur existe, la méthode en crée un automatiquement.

Enregistrement d'un MBean

Notes

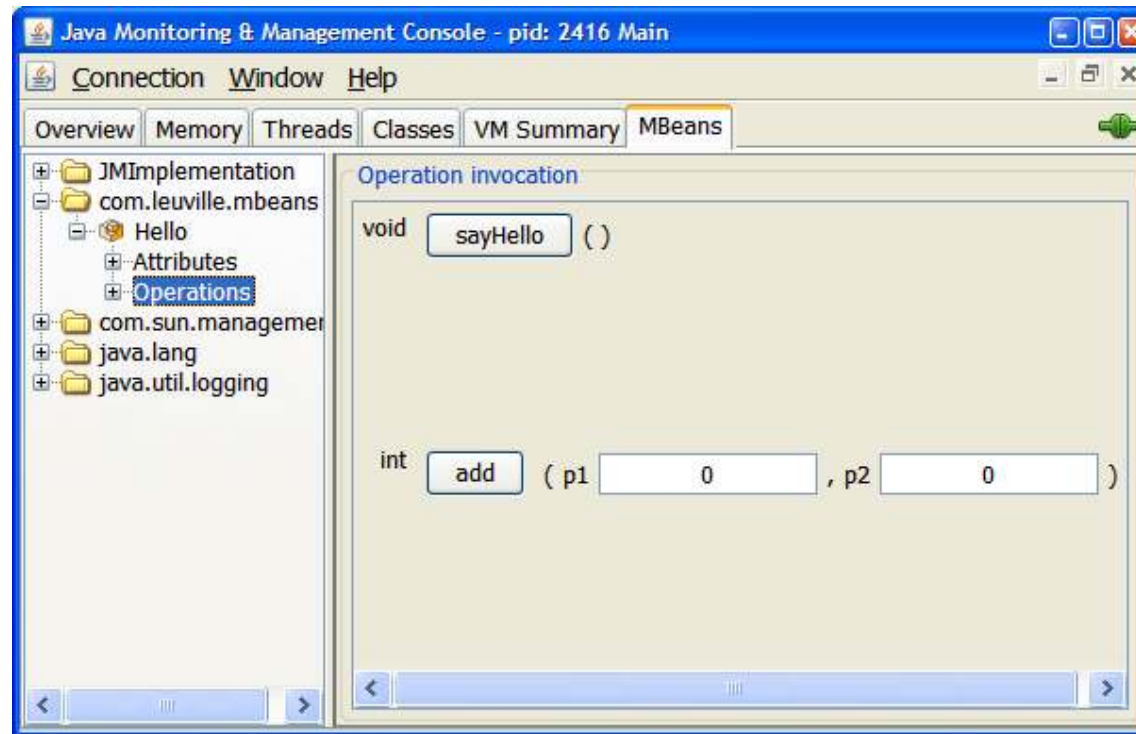
Enregistrement d'un MBean



Enregistrement d'un MBean

Notes

Enregistrement d'un MBean

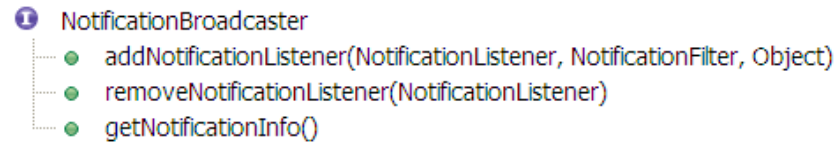


Enregistrement d'un MBean

Notes

Envoi de notifications

- Afin de pouvoir envoyer des notifications, un MBean doit implémenter l'interface `javax.management.NotificationBroadcaster`.



- La classe `javax.management.NotificationBroadcasterSupport` est une implémentation standard de cette interface et peut être utilisée comme classe mère des MBeans.

```
public class Hello extends NotificationBroadcasterSupport implements HelloMBean {

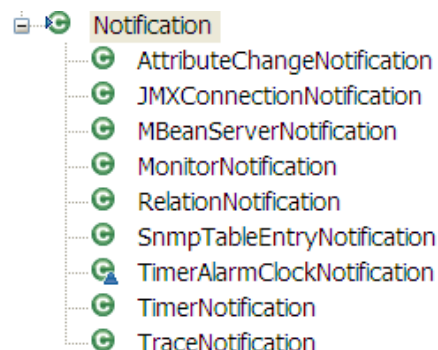
    private final String name = "Leuville Objects";
    private int cacheSize = 200;
    private long sequenceNumber = 1;
    ...
    public synchronized void setCacheSize(int size) {
        int oldSize = this.cacheSize;
        this.cacheSize = size;
        Notification n = new AttributeChangeNotification(
            this, sequenceNumber++, System.currentTimeMillis(),
            "Cache size changed", "CacheSize", "int",
            oldSize, this.cacheSize);
        sendNotification(n);
    }
}
```

Envoi de notifications

Notes

Envoi de notifications

- Pour émettre une notification, le MBean doit construire une instance de `javax.management.Notification`.



- La méthode `sendNotification(Notification)` de `NotificationBroadcasterSupport` permet d'envoyer la notification à tous les systèmes qui ont souscrit au MBean.
- Le MBean doit également proposer une méthode renvoyant les différentes notifications qu'il peut émettre :

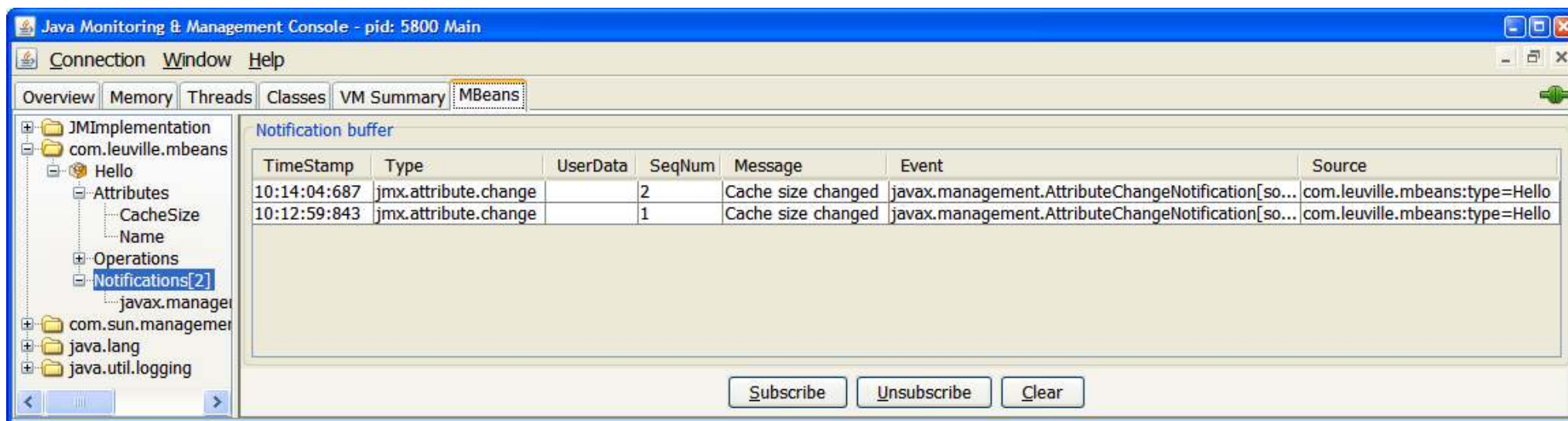
```
@Override
public MBeanNotificationInfo[] getNotificationInfo() {
    String[] types = new String[] { AttributeChangeNotification.ATTRIBUTE_CHANGE };
    String name = AttributeChangeNotification.class.getName();
    String description = "An attribute of this MBean has changed";
    MBeanNotificationInfo info = new MBeanNotificationInfo(types, name, description);
    return new MBeanNotificationInfo[] {info};
}
```

Envoi de notifications

Notes

Envoi de notifications

Souscription et notification avec la JConsole



Java Monitoring & Management Console - pid: 5800 Main

Connection Window Help

Overview Memory Threads Classes VM Summary MBeans

Notification buffer

TimeStamp	Type	UserData	SeqNum	Message	Event	Source
10:14:04:687	jmx.attribute.change		2	Cache size changed	javax.management.AttributeChangeNotification[so...	com.leuville.mbeans:type=Hello
10:12:59:843	jmx.attribute.change		1	Cache size changed	javax.management.AttributeChangeNotification[so...	com.leuville.mbeans:type=Hello

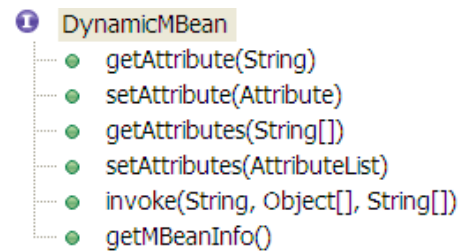
Subscribe Unsubscribe Clear

Envoi de notifications

Notes

Dynamic MBean

- Les Dynamic MBeans sont des MBeans dont la structure est découverte dynamiquement.
- Un Dynamic MBean est une classe Java implémentant l'interface `javax.management.DynamicMBean`.



Dynamic MBean

Notes

Dynamic MBean

Exemple : un gestionnaire de propriétés

```
public class PropertiesManager implements DynamicMBean {
    private Properties properties = new Properties();

    public PropertiesManager() {
        reset(); }

    public String getProperty(String key) {
        return properties.getProperty(key); }

    public void clear() {
        properties.clear(); }

    public void reset() {
        try {
            InputStream in = PropertiesManager.class
                .getClassLoader().getResourceAsStream("dynamic/config.xml");
            properties.loadFromXML(in);
        } catch (IOException ioe) {
            ioe.printStackTrace(); }
    }
    ...
}
```

Dynamic MBean

Notes

Dynamic MBean

Exemple: : définition des méthodes `getAttribute` et `getAttributes`

```
@Override
public Object getAttribute(String attribute)
    throws AttributeNotFoundException, MBeanException,
    ReflectionException {
    return properties.getProperty(attribute);
}

@Override
public AttributeList getAttributes(String[] attributes) {
    AttributeList list = new AttributeList(properties.size());

    for (String att : attributes) {
        String value = getProperty(att);
        list.add(new Attribute(att, value));
    }

    return list;
}
```

Dynamic MBean

Notes

Dynamic MBean

Exemple : définition des méthode setAttribute et setAttributes

```
@Override
public void setAttribute(Attribute attribute) throws AttributeNotFoundException,
    InvalidAttributeValueException, MBeanException, ReflectionException {
    String key = attribute.getName();
    String value = (String) attribute.getValue();
    properties.put(key, value);
}

@Override
public AttributeList setAttributes(AttributeList attributes) {
    for (Iterator it = attributes.iterator(); it.hasNext();) {
        Attribute att = (Attribute) it.next();
        try {
            setAttribute(att);
        } catch (AttributeNotFoundException e) { e.printStackTrace(); }
        } catch (InvalidAttributeValueException e) { e.printStackTrace(); }
        } catch (MBeanException e) { e.printStackTrace(); }
        } catch (ReflectionException e) { e.printStackTrace(); }
    }
    return attributes;
}
```

Dynamic MBean

Notes

Dynamic MBean

Exemple : définition de la méthode invoke

```
@Override
    public Object invoke(String actionName, Object[] params, String[] signature)
        throws MBeanException, ReflectionException {

        if ("reset".equals(actionName)) {
            reset();
        } else if ("clear".equals(actionName)) {
            clear();
        }

        return null;
    }
```


Dynamic MBean

Notes

Dynamic MBean

Exemple : Définition de la méthode getMBeanInfo

```
@Override
public MBeanInfo getMBeanInfo() {
    MBeanAttributeInfo[] mbAttInfos = new MBeanAttributeInfo[properties.size()];
    int index = 0;
    for (Enumeration e = properties.keys(); e.hasMoreElements();) {
        String key = (String) e.nextElement();
        mbAttInfos[index++] = new MBeanAttributeInfo(
            key, "String", "Propriété n°" + index, false, false, false);
    }

    MBeanConstructorInfo[] mbConsInfos = null;
    try {
        mbConsInfos = new MBeanConstructorInfo[] {
            new MBeanConstructorInfo("Default Constructor", getClass().getConstructor()) };
    } catch (SecurityException e) { e.printStackTrace(); }
    } catch (NoSuchMethodException e) { e.printStackTrace(); }
    }
    ...
}
```

Dynamic MBean

Notes

Dynamic MBean

Exemple : Définition de la méthode getMBeanInfo

```
...
MBeanOperationInfo[] mbOpInfos = null;
try {
    mbOpInfos = new MBeanOperationInfo[] {
        new MBeanOperationInfo("clear", getClass().getMethod("clear")),
        new MBeanOperationInfo("reset", getClass().getMethod("reset"))
    };
} catch (SecurityException e) { e.printStackTrace(); }
} catch (NoSuchMethodException e) { e.printStackTrace(); }
}

MBeanNotificationInfo[] nbNotifInfos = new MBeanNotificationInfo[0];

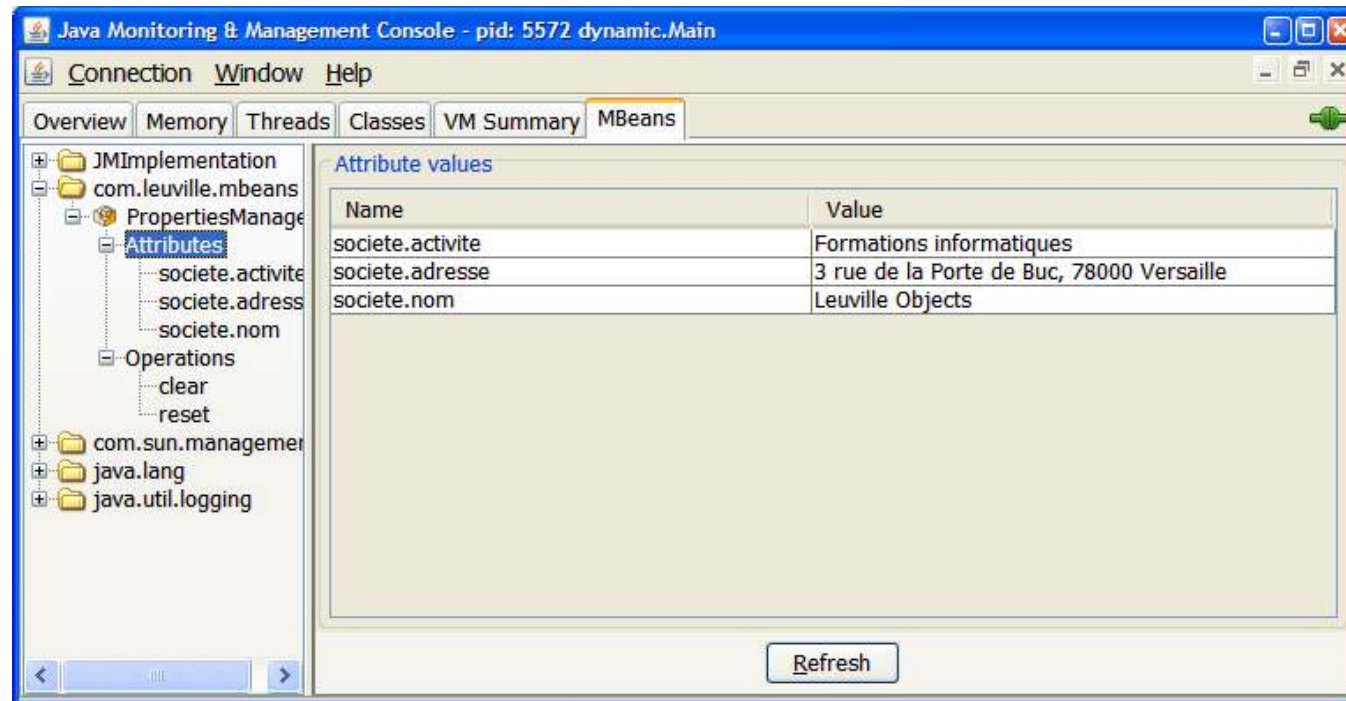
MBeanInfo info = new MBeanInfo(getClass().getName(),
    "Gestionnaire de propriétés",
    mbAttInfos, mbConsInfos, mbOpInfos, nbNotifInfos);

return info;
}
```

Dynamic MBean

Notes

Dynamic MBean



Dynamic MBean

Notes

Model MBean

- MBean configurable et générique permettant à des applications de gérer des ressources dynamiquement.
- Son interface peut être définie à la volée, permettant ainsi à un administrateur de créer et d'instancier des MBeans à chaud.
- Un model MBean est défini à partir d'un `ModelMBeanInfo`, et manipulé à travers une instance de `RequiredModelMBean`.

Etapes pour créer un Model MBean :

- Décrire le MBean dans un `ModelMBeanInfo`.
- Créer une instance de `RequiredModelMbean`.
- Associer l'objet cible au MBean.
- Associer le MBean au serveur JMX.

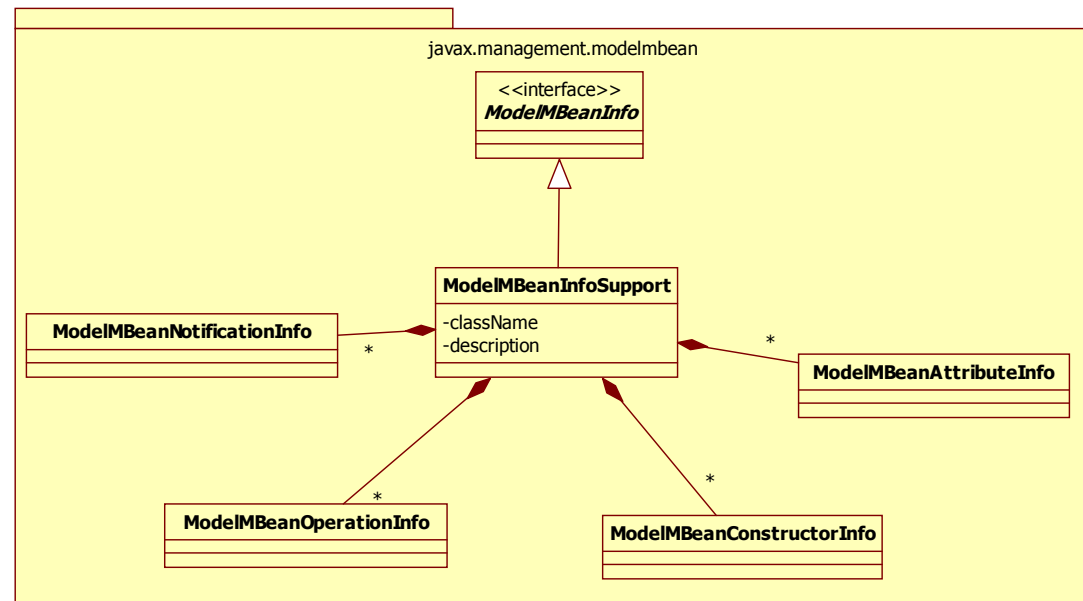
Model MBean

Notes

Model MBean

Description du MBean

- La description d'un Model MBean est fait dans une instance de `javax.management.modelmbean.ModelMBeanInfo`.



Model MBean

Notes

Model MBean

Description du MBean : exemple

```
Class printerClass = Printer.class;

ModelMBeanAttributeInfo[] mbAttInfos = new ModelMBeanAttributeInfo[2];
try {
    Method ipAdressGetter = printerClass.getMethod("getIpAddress");
    ModelMBeanAttributeInfo att1 = new ModelMBeanAttributeInfo(
        "ipAddress", "Printer IP Address", ipAdressGetter, null);
    mbAttInfos[0] = att1;
    Method statusGetter = printerClass.getMethod("getStatus");
    ModelMBeanAttributeInfo att2 = new ModelMBeanAttributeInfo(
        "status", "Printer status", statusGetter, null);
    mbAttInfos[1] = att2;
} catch (Throwable t) {
    t.printStackTrace();
}
```

Model MBean

Notes

Model MBean

Description du MBean : exemple

```
ModelMBeanConstructorInfo[] mbConsInfos = new ModelMBeanConstructorInfo[2];
try {
    mbConsInfos[0] = new ModelMBeanConstructorInfo(
        "Constructor", printerClass.getConstructor());
    mbConsInfos[1] = new ModelMBeanConstructorInfo(
        "Constructor", printerClass.getConstructor(String.class));
} catch (Throwable t) {
    t.printStackTrace();
}

ModelMBeanOperationInfo[] mbOpInfos = new ModelMBeanOperationInfo[1];
try {
    Method printMethod = printerClass.getMethod("print");
    mbOpInfos[0] = new ModelMBeanOperationInfo("Print", printMethod);
} catch (Throwable t) {
    t.printStackTrace();
}

ModelMBeanNotificationInfo[] nbNotifInfos = new ModelMBeanNotificationInfo[0];
```

Model MBean

Notes

Model MBean

Description du MBean : exemple

```
ModelMBeanInfo info = new ModelMBeanInfoSupport(  
    Printer.class.getName(), "Network printer",  
    mbAttInfos, mbConsInfos, mbOpInfos, nbNotifInfos);
```

RequiredModelMBean

- Encapsule la description du MBean ainsi que la ressource gérée.

```
RequiredModelMBean rmmb = new RequiredModelMBean(info);  
rmmb.setManagedResource(new Printer("192.168.0.15"), "ObjectReference");
```

- Le deuxième paramètre de la méthode setManagedResource indique le type de ressource :
 - ObjectReference
 - Handle
 - IOR
 - EJBHandle
 - RMIRreference

ModelMBean

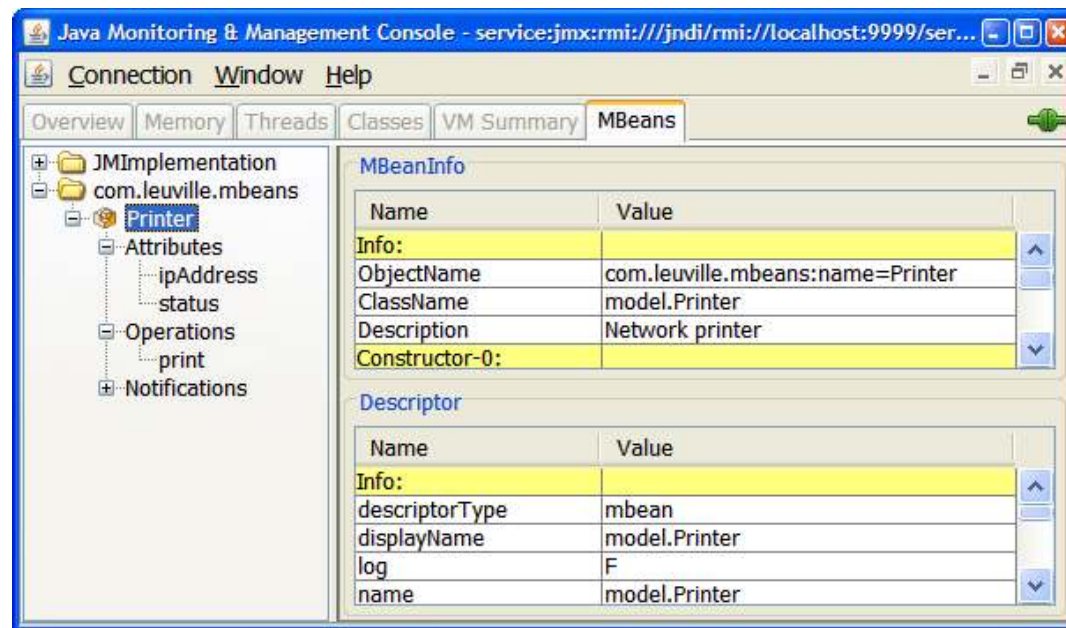
Notes

Model MBean

Associer un Model MBean à un serveur JMX

- Le principe est le même que pour les autres types de MBeans.

```
server.registerMBean(rmmb, new ObjectName("com.leuville.mbeans:name=Printer"));
```



Model MBean

Notes

Java, Programmation avancée

Connecteurs JMX

Version 2.2

- Utiliser le connecteur JMX RMI
- Ajouter un connecteur RMI à un serveur JMX
- Développer un client utilisant un connecteur RMI

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Lancement d'un connecteur RMI

- On peut choisir de rendre disponible un serveur JMX via un connecteur RMI.
- Le service RMI utilisé peut alors être indépendant de la JVM hébergeant le serveur JMX.
- Pour cela, il faut :
 - Lancer un service RMI Registry,
 - Créer un serveur JMX en invoquant la méthode `createMBeanServer` de `MBeanServerFactory`,
 - Créer un connecteur RMI et l'associer au serveur JMX,
 - Démarrer le connecteur.

Lancement d'un connecteur RMI

Notes

Lancement d'un connecteur RMI

Lancer le service RMI Registry

- Utiliser l'exécutable `rmiregistry` fourni avec le JDK.



Lancement d'un connecteur RMI

Notes

Lancement d'un connecteur RMI

Créer le serveur JMX

```
MBeanServer mbs = MBeanServerFactory.createMBeanServer();

ObjectName name = new ObjectName("com.leuville.mbeans:type=Hello");
Hello mbean = new Hello();

mbs.registerMBean(mbean, name);
```

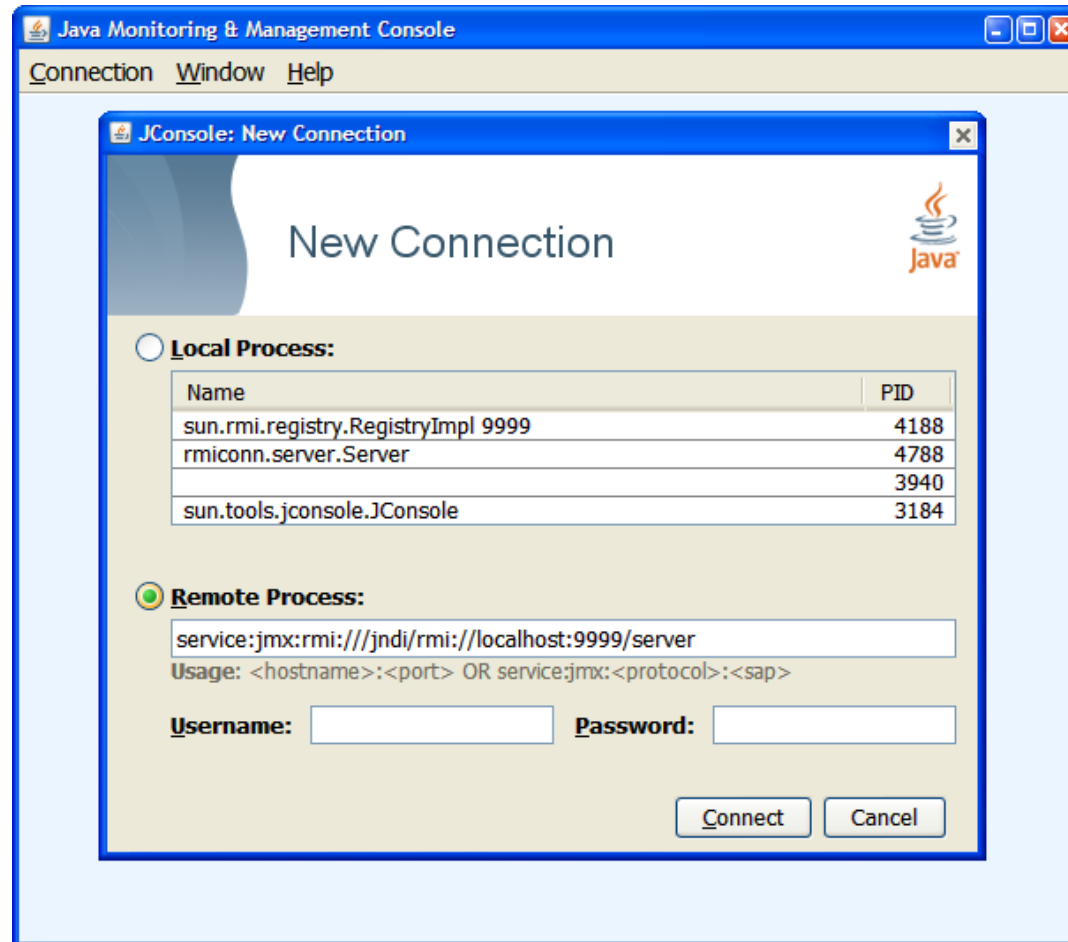
Créer le connecteur, l'associer au serveur et le démarrer

```
JMXServiceURL url = new JMXServiceURL(
    "service:jmx:rmi:///jndi/rmi://localhost:9999/server");
JMXConnectorServer cs = JMXConnectorServerFactory
    .newJMXConnectorServer(url, null, mbs);
cs.start();
```

Lancement d'un connecteur RMI

Notes

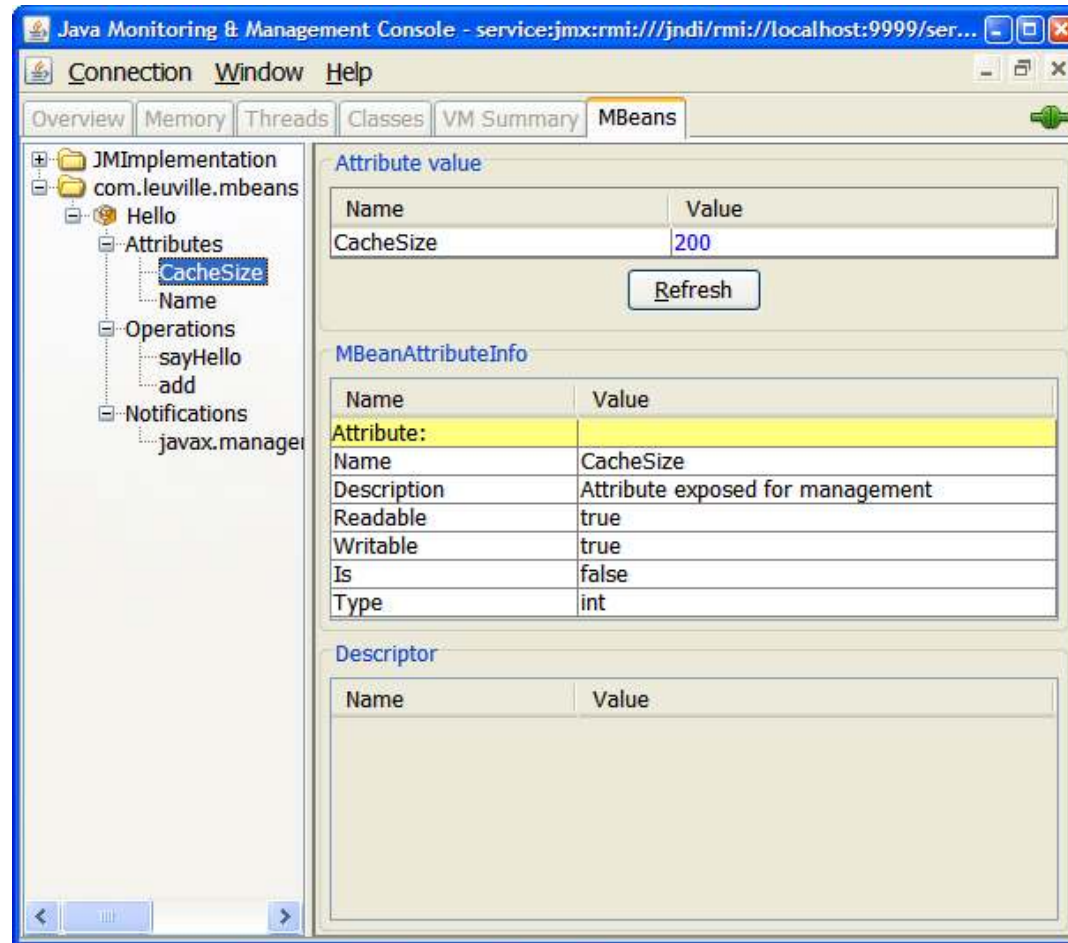
Utiliser JConsole via le connecteur RMI



Utiliser JConsole avec le connecteur RMI

Notes

Utiliser JConsole avec le connecteur RMI



Utiliser JConsole avec le connecteur RMI

Notes

Accès au serveur JMX via le connecteur RMI

- Coté client, on accède au serveur JMX via la classe `JMXConnector`.
- Une fois connecté, on obtient une instance de `MBeanServerConnection` grâce à laquelle on peut lire/écrire des valeurs d'attribut et invoquer des méthodes sur les MBeans.

Connexion au serveur JMX

```
JMXServiceURL url = new JMXServiceURL(
    "service:jmx:rmi:///jndi/rmi://localhost:9999/server");
JMXConnector jmxc = JMXConnectorFactory.connect(url, null);
MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
```


Accès au serveur JMX via le connecteur RMI

Notes

Accès au serveur JMX via le connecteur RMI

Manipuler les attributs d'un MBean

```
// Récupération de la valeur d'un attribut d'un MBean
int cacheSize = (Integer) mbsc.getAttribute(
    new ObjectName("com.leuville.mbeans:type=Hello"), "CacheSize");
System.out.println(cacheSize);

// Ecriture de la valeur d'un attribut
Attribute a = new Attribute("CacheSize", 350);
mbsc.setAttribute(new ObjectName("com.leuville.mbeans:type=Hello"), a);
```

Invoquer une méthode d'un MBean

```
// Appel d'une méthode
int somme = (Integer) mbsc.invoke(
    new ObjectName("com.leuville.mbeans:type=Hello"), "add",
    new Object[]{3,5}, new String[]{"int", "int"});
System.out.println(somme);
```

Accès au serveur JMX via le connecteur RMI

Notes

Java, Programmation avancée

Sécurité et permissions en Java

Version 2.2

- Les gestionnaires de sécurité en Java
- Les permissions
- ...

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Le gestionnaire de sécurité en Java

Appelé après le chargeur de classe et la vérification du bytecode

Vérifie entre autres :

- Si le thread courant peut créer un nouveau chargeur de classe
- Si le thread courant peut arrêter la machine virtuelle
- Si une classe peut accéder à un membre d'une autre classe
- Si un thread local peut accéder à un fichier local
- Si un thread local peut ouvrir une connexion socket vers un hôte externe
- Si une classe peut démarrer un travail d'impression
- etc.

Par défaut, aucun gestionnaire de sécurité installé pour les applications

Ce n'est pas le cas pour les applets (politique de sécurité très restrictive)

Utilisation de la classe `SecurityManager`

Le gestionnaire de sécurité en Java

Notes :

La sécurité de la plate-forme Java 2

Un modèle qui a évolué depuis le JDK 1.0

- JDK 1.0 : Les classes locales avaient tous les droits et les classes distantes étaient confinées au sandbox
 - Le code distant ne pouvait que dessiner que sur l'écran et interagir avec l'utilisateur
 - Aucun accès aux ressources locales pour les applets
- JDK 1.1 : Ajout de la signature par une entité fiable pour le code distant
 - Permettait d'attribuer les mêmes permissions que les classes locales
- JDK 1.2 : Ajout de règles de sécurité (security policies)
 - Mise en correspondance du code source et des permissions

Une source de code = deux propriétés

- L'emplacement du code (code location)
- Les certificats (certificates)

La sécurité de la plate-forme Java 2

Notes :

La sécurité de la plate-forme Java 2

Une permission = une propriété vérifiée par un gestionnaire de sécurité

- Depuis le JDK 1.2 de nombreuses classes de permission sont gérées

```
FilePermission perm = new FilePermission("/TestPermissions.java", "read");
```

Dans le JDK 1.2, implémentation par défaut de la classe Policy

- Lecture des permissions dans un fichier de permission fournit par défaut avec le JRE

```
permission java.util.FilePermission "/TestPermissions.java", "read"
```

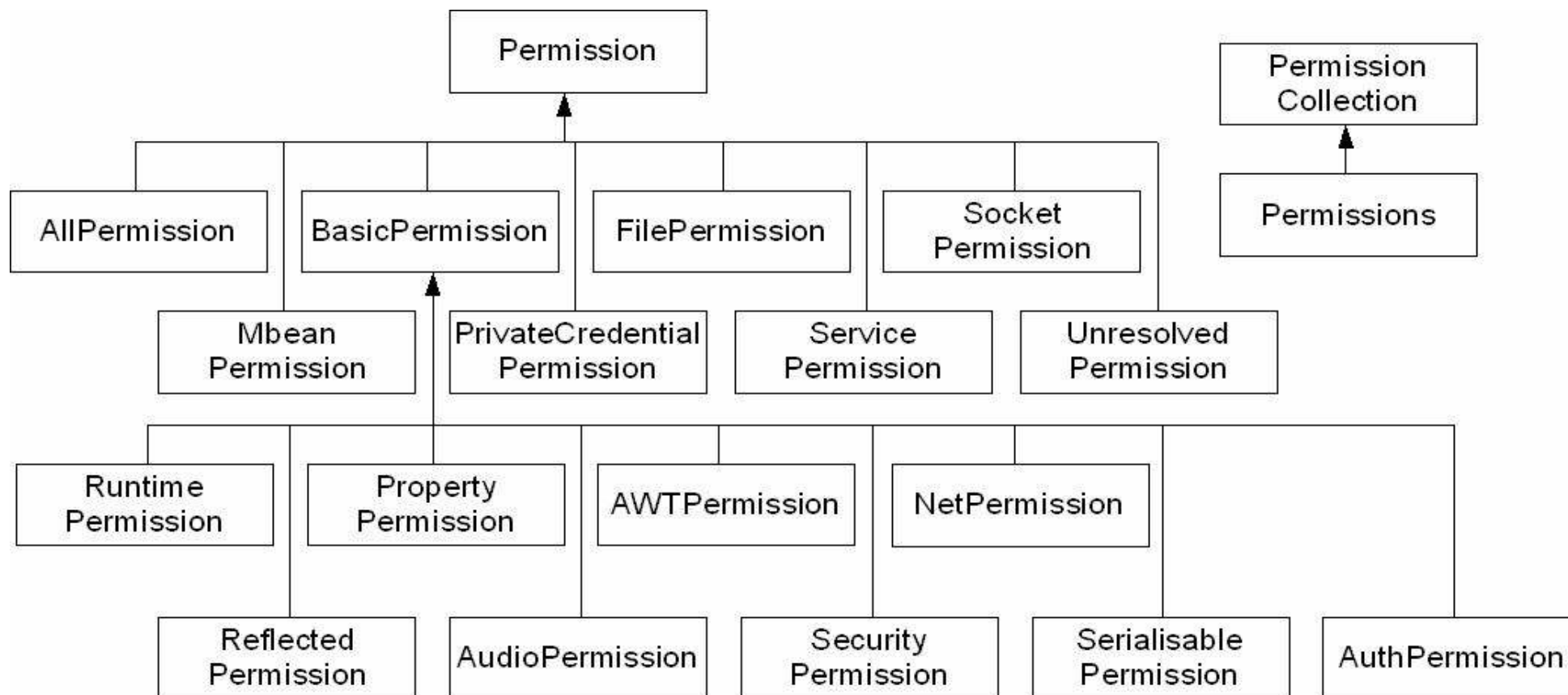
Se trouve dans le répertoire JRE_PATH/lib/security

Les nouvelles versions de JDK ont apporté de nouvelles classes de permission

La sécurité de la plate-forme Java 2

Notes :

La sécurité de la plate-forme Java 2



Les classes de permission de Java

Chaque classe a un domaine de protection

- Un objet qui encapsule la source de code et la collection de permissions de la classe

La sécurité de la plate-forme Java 2

Notes :

Les fichiers de règles de sécurité

Accordent les permissions d'accès aux ressources

- Possibiliter d'installer des fichiers de règles aux emplacement standard
 - le fichier `java.policy` du répertoire home de la plate-forme Java
 - le fichier `.java.policy` le répertoire personnel de l'utilisateur
- Il est possible de modifier les chemins standard
 - Modification du fichier de configuration `java.security`

```
policy.url.1=file:${java.home}/lib/security/java.policy  
policy.url.2=file:${user.home}/.java.policy
```

Le fichier `java.security` se trouve à l'emplacement `JRE_PATH/lib/security`

Pour utiliser le fichier `policy` il faut le spécifier dans la ligne de commande

- Utilisation des règles indiquées en plus des règles standards

```
java -Djava.security.policy=MonAppli.policy MonAppli
```
- Utilisation des nouvelles règles uniquement

```
java -Djava.security.policy==MonAppli.policy MonAppli
```

Les fichiers de règles de sécurité

Notes :

Les fichiers de règles de sécurité

Il est nécessaire de mettre en place un gestionnaire de sécurité

```
System.setSecurityManager(new SecurityManager());
```

Code a placer dans la méthode main

Rappel : Par défaut, aucun gestionnaire de sécurité installé en Java

- Implique de spécifier dans la ligne de commande l'utilisation du gestionnaire de sécurité
`java -Djava.security.manager -Djava.security.policy=MonAppli.policy MonAppli`

Un fichier de règles contient un séquence d'entrées 'grant'

```
grant [sourcecode]
{
    permission1;
    permission2;
    ...
};
```

- sourcecode qui est optionnel indique le chemin d'accès au code (sous forme d'URL)

```
grant codeBase "file:///c:/repertoire/nomfichier.ext"
{
    ...
};
```


Les fichiers de règles de sécurité

Notes :

Les fichiers de règles de sécurité

- Plusieurs écritures pour définir l'URL de type fichier

file:c:/repertoire/nomfichier.ext
file:/c:/repertoire/nomfichier.ext
file://c:/repertoire/nomfichier.ext
file:///c:/repertoire/nomfichier.ext

La structure d'une permission peut contenir jusqu'à 4 parties

permission NomClasse NomCible, ListeActions

- NomClasse est le nom complet de la classe : package.NomClasse
- NomCible valeur spécifique de permission
- ListeActions est spécifique à la permission (droits en lecture, écriture, etc.)

Une permission est une autorisation bien définie

- La permission `java.io.FilePermission` impose de spécifier la cible
- La permission `java.net.SocketPermission` impose de spécifier l'hôte et le numéro de port
- la permission `java.util.PropertyPermission` impose de spécifier la propriété à laquelle elle s'applique

Les fichiers de règles de sécurité

Notes :

Les fichiers de règles de sécurité

Utilisation de permissions sur des fichiers

- Les cibles de permission de fichiers peuvent prendre plusieurs formes
 - *fichier* : un nom de fichier
 - *répertoire/* : un répertoire
 - *répertoire/** : tous les fichiers d'un répertoire
 - *** : tous les fichiers du répertoire courant
 - *répertoire/-* : tous les fichiers du répertoire ou de ses sous-répertoires
 - *-* : tous les fichiers du répertoire courant ou de ses sous-répertoires
 - *<<ALL FILES>>* : tous les fichiers du système de fichiers
- La définition de permissions pour des fichiers se fait de la façon suivante
permission java.io.FilePermission "/myapp/-", "read,write";

Les fichiers de règles de sécurité

Notes :

Les fichiers de règles de sécurité

Utilisation de permissions sur des sockets

- Les spécifications d'hôte sont de la forme
 - *NomHôte* ou *AdresseIp* : un hôte unique
 - *localhost* ou *chaîne vide* : l'hôte local
 - **.SuffixeDomaine* : tout hôte dont le domaine se termine par le suffixe donné
 - *** : tous les hôtes
- Les numéros de port sont facultatifs et sont de la forme
 - *:n* : un port unique
 - *:n-* : tous les ports numérotés de n et supérieur
 - *:-n* : tous les ports numérotés jusqu'à n
 - *:n1-n2* : tous les ports de n1 à n2
- La permission peut se déclarer de la façon suivante
permission java.net.SocketPermission "*.host.com:80:120", "connect";

Les fichiers de règles de sécurité

Notes :

Les fichiers de règles de sécurité

Utilisation de permission sur les propriétés

- Deux formes possibles
 - *propriété* : une propriété spécifique (java.home, etc.)
 - *PréfixePropriété.** : toutes les propriétés avec le préfixe (java.vm.*, etc.)
- Exemple d'utilisation

```
permission java.util.PropertyPermission "java.vm.*", "read";
```

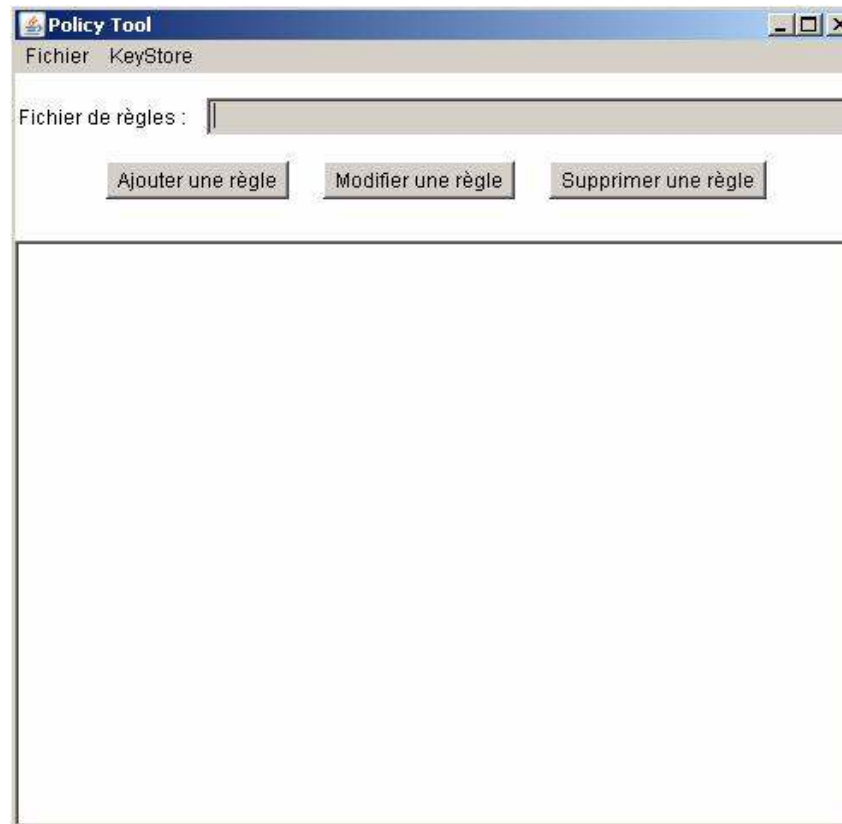

Les fichiers de règles de sécurité

Notes :

L'outil PolicyTool

La mise au point de fichier de permissions peut-être compliqué

- Syntaxe fastidieuse
- Connaissance de l'ensemble des cibles peut évier
- L'utilisation d'un outil pour cette création peut-être avantageuse



L'outil PolicyTool

Notes :

Java, Programmation avancée

JAAS : Java Authentication and Authorization System

Version 2.2

- L'utilité de JAAS
- L'authentification
- La gestion des autorisations
- Les modules d'identification
- Le mécanisme des Callbacks

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Objectifs de JAAS

Framework de sécurité Java

- Objectifs : standardiser la gestion des authentifications et l'attribution d'autorisations avec Java
- Intégré à Java depuis la version 1.4, mais pas utilisé par défaut
- Indispensable avec la montée en puissance de Java sur le client-serveur et les applications multi-niveaux
- Gestion des authentifications
 - reconnaître les utilisateurs de façon fiable
- Gestion des autorisations
 - s'assurer que les opérations sensibles sont exécutées par des utilisateurs ayant les permissions adéquates

Caractéristiques principales

- Architecture modulaire basée sur un ensemble de classes abstraites et d'interfaces
- Extensible par "branchement" de modules compatibles
- Utilisable de deux façons
 - par programmation
 - de façon déclarative

Objectifs de JAAS

Framework de sécurité Java

JAAS a été initialement proposé en tant qu'extension à la version 1.2 de Java.

Bien que JAAS soit totalement intégré à Java depuis la version J2SE 1.4, les applications Java doivent être explicitement programmées pour l'utiliser.

JAAS permet d'affranchir une application Java des spécificités de la technologies utilisée pour effectuer les opérations telles que l'authentification.

Possibilités de JAAS

Authentication

- Authentication déléguée à un système UNIX
- Authentication déléguée à un système NT
- Authentication KERBEROS
- Authentication fondée sur des certificats
- Authentication réalisée à l'aide d'un annuaire à travers JNDI
- Authentication réalisée avec un module personnalisé

Permissions

- Attacher un ensemble de permissions à un utilisateur authentifié
 - fichier .policy
 - API dédiée

Possibilités de JAAS

Notes

Terminologie JAAS

Entité / Subject

- Personne ou système qui interagit avec une application Java, avec une ou plusieurs identités

Identité / Principal

- Résultat de l'authentification d'une entité ou subject

LoginModule ou module d'identification

- Représente une technique d'authentification d'un utilisateur
- Configurable à l'aide d'un fichier

LoginContext ou contexte d'identification

- Responsable de l'authentification d'un subject dans une application

CallbackHandler

- Permet de transférer des informations d'authentification de l'application vers les modules d'identification (LoginModule)

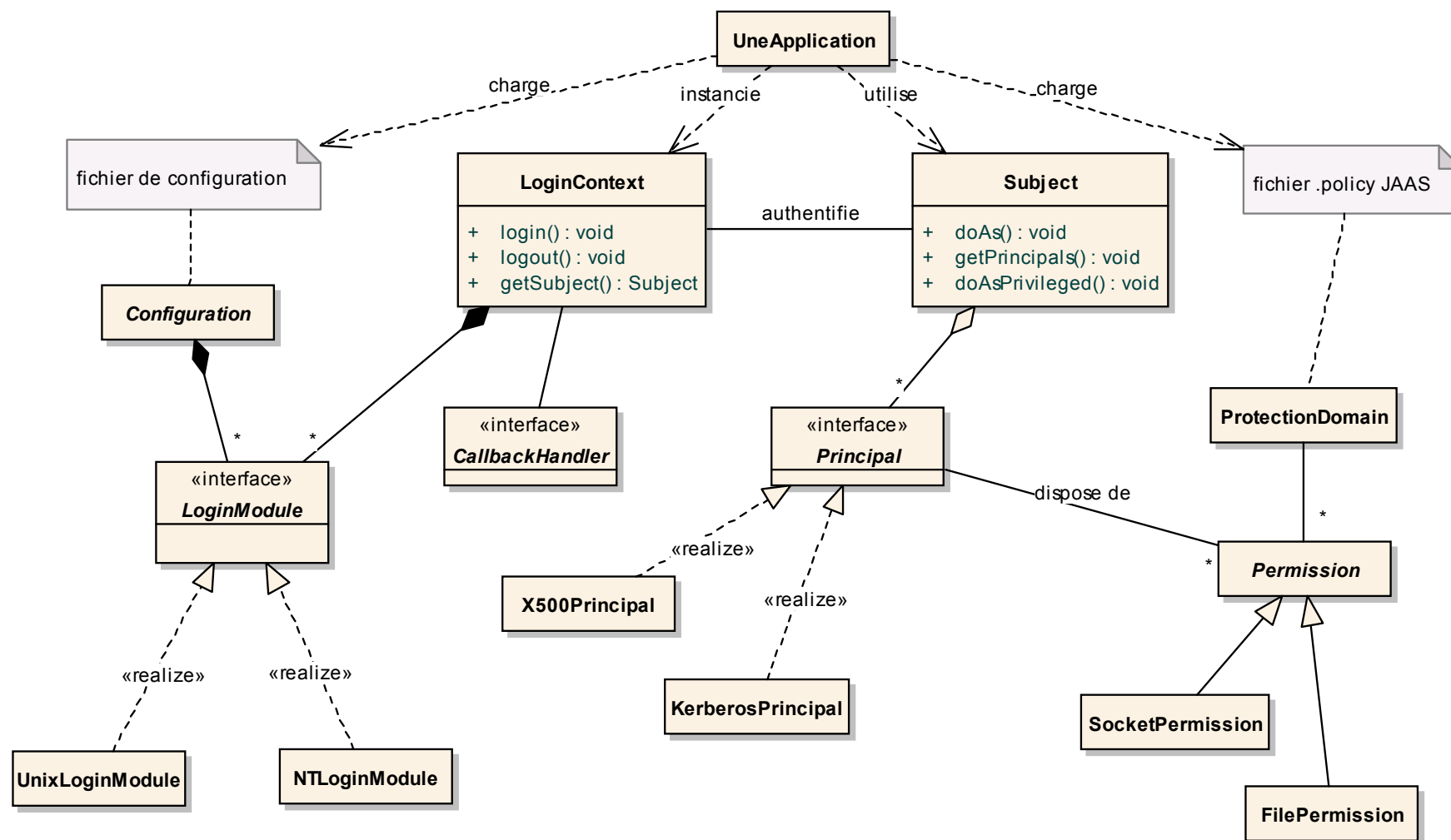
Terminologie JAAS

Notes

Une personne peut interagir avec un système selon plusieurs identités. C'est notamment le cas lorsque ce système est constitué de différents modules possédant chacun leurs règles de sécurité propres.

Cette association entre une entité et plusieurs identités permet de rendre possible le Single-Sign-On (SSO).

Vue d'ensemble



Vue d'ensemble

Notes

Certaines classes ne sont pas représentées ici pour favoriser la lisibilité du diagramme.

A quoi ressemble une application qui utilise JAAS

Les grandes étapes

- Instanciation d'un LoginContext
 - sélection d'une configuration JAAS
 - définition d'un CallbackHandler permettant de récupérer les informations de type login / mot de passe
- Identification de l'utilisateur
 - Invocation de la méthode login() permettant de réaliser l'authentification auprès des différents LoginModule
 - Défaut d'identification signalé par la levée d'une exception LoginException
- Exécution du code protégé par appel à Subject.doAs() ou doAsPrivileged()
 - Permet d'exécuter un code sensible avec une identité particulière
 - Le code exécuté est défini au sein d'une classe implémentant l'interface PrivilegedAction, ou PrivilegedExceptionAction

A quoi ressemble une application qui utilise JAAS

Notes

A quoi ressemble une application qui utilise JAAS

Code Java

- ExempleJAAS : nom d'une application définie dans un fichier de configuration JAAS
- TextCallbackHandler est une classe qui lit le login et le mot de passe à partir de l'entrée standard
- Le code "sensible" est défini au sein de la méthode run() d'une classe qui implémente PrivilegedAction
- Cette classe doit implémenter PrivilegedExceptionAction si le traitement lève une exception que l'on souhaite traiter

```
import javax.security.auth.login.*;
import com.sun.security.auth.callback.TextCallbackHandler;

...

LoginContext lc = null;
try {
    lc = new LoginContext("ExempleJAAS", new TextCallbackHandler());
    lc.login();
    Object résultat = Subject.doAs(lc.getSubject(), new PrivilegedAction() {
        public Object run () {
            Object x = null;
            // x = traitement soumis à autorisation
            return x;
        }
    });
    System.out.println (résultat);
    lc.logout();
} catch (LoginException e) {
    e.printStackTrace();
} catch (Exception e) {
    e.printStackTrace();
}
```


A quoi ressemble une application qui utilise JAAS

Code Java

Le nom spécifié à la construction d'une instance de `LoginContext` représente une application définie dans un fichier de configuration JAAS. Cette application contient une suite de modules d'identification (`LoginModule`).

`TextCallbackHandler` est fourni par Sun à titre d'exemple et ne devrait pas être utilisée telle quelle dans un véritable projet. Cette classe est membre du paquetage `com.sun.security.auth.callback`.

A quoi ressemble une application qui utilise JAAS

Exemple de configuration associée

- fichier de configuration : par exemple **jaas.conf**

```
ExempleJAAS {  
    com.sun.security.auth.module.Krb5LoginModule required;  
};
```

- Ici le LoginModule choisi est Kerberos
- choisi à l'exécution par l'emploi d'une propriété

```
-Djava.security.auth.login.config=jaas.conf
```

Les autres possibilités de configuration seront explicitées ultérieurement.

A quoi ressemble une application qui utilise JAAS

Notes

JAAS a été conçue de façon à reporter un grand nombre d'opérations de configuration sur des fichiers de paramètres plutôt que sur du code, notamment pour tout ce qui relève de technologies spécifiques.

Le fichier de configuration peut contenir plusieurs définitions d'applications.

A quoi ressemble une application qui utilise JAAS

Les permissions nécessaires à l'exécution

- Le code qui instancie LoginContext et invoque doAs() ou doAsPrivileged()
 - `permission javax.security.auth.AuthPermission "createLoginContext";`
 - `permission javax.security.auth.AuthPermission "doAs"`
 - `permission javax.security.auth.AuthPermission "doAsPrivileged"`
- Le module d'identification
 - `permission javax.security.auth.AuthPermission "modifyPrincipals"`
 - parfois d'autres permissions suivant le type de module : ouverture de sockets, accès à des fichiers, ...

A quoi ressemble une application qui utilise JAAS

Notes

Les modules d'identification doivent disposer de permissions qui varient suivant la façon dont ils réalisent cette identification.

Si ces modules sont installés en tant qu'extensions standards de la Machine Virtuelle, ils bénéficieront de toutes les permissions accordées par le fichier Policy par défaut.

Un rappel sur ce sujet est effectué dans la suite de ce document.

A quoi ressemble une application qui utilise JAAS

Etapas et recommandations

- Séparer le code de configuration de JAAS et le code d'exécution des actions privilégiées
- Créer le fichier de configuration des modules d'identification
- Créer le fichier de définition de la politique de sécurité JAAS accordant les permissions spécifiques du ou des utilisateurs
- Créer un fichier .policy standard accordant les droits nécessaires à l'exécution de JAAS
- Exécuter le code en spécifiant les options et propriétés suivantes
 - -classpath
 - -Djava.security.manager pour activer le contrôle d'accès
 - -Djava.security.policy=<fichier policy standard>
 - -Djava.security.auth.policy=<fichier policy JAAS>
 - -Djava.security.auth.login.config=<fichier de configuration des modules d'identification JAAS>

A quoi ressemble une application qui utilise JAAS

Notes

LoginContext

Méthode login()

- Instancie un objet de type `javax.security.auth.Subject`
- Construit un `LoginModule` en fonction des choix définis dans la configuration JAAS
- Initialise ce `LoginModule` avec :
 - l'instance de `Subject` créée au départ
 - une instance de `CallbackHandler` optionnelle
- Exécute l'authentification
 - l'instance de `CallbackHandler` est utilisée afin de récupérer les informations d'identification
 - ces informations sont ensuite utilisées par le `LoginModule` pour déterminer s'il y a échec ou succès de l'authentification
 - en cas de succès, un nouveau `Principal` est ajouté au `Subject`

Méthode logout()

- Invoque la méthode `logout()` de chaque module d'identification rattaché à ce `LoginContext`
- Détruit toutes les identités attachées à l'entité déconnectée

LoginContext

Notes

Les modules d'identification ou LoginModule

Exemples

```
Login1 {  
    com.sun.security.auth.module.Krb5LoginModule required;  
    com.mycompany.auth.module.RetinaScanModule sufficient;  
};  
  
Login2 {  
    com.sun.security.auth.module.UnixLoginModule required;  
};
```

Module

- un type + une étiquette parmi required, sufficient, requisite, optional

Types de modules définis dans `com.sun.security.auth.module`

UnixLoginModule	Associe au subject courant une identité Unix
NTLoginModule	Associé au subject courant une ou plusieurs identités NT
Krb5LoginModule	Identification utilisant les protocoles de Kerberos
JndiLoginModule	Demande un nom d'utilisateur et un mot de passe et effectue le contrôle par rapport à un annuaire accessible via JNDI
KeyStoreLoginModule	Prend en entrée un alias de keystore et associe au subject courant l'identité et les droits liés à cet alias

Les modules d'identification ou LoginModule

Notes

Les modules sont définis de façon externe à l'application, grâce à un fichier de configuration qui sera spécifié au lancement de l'application.

La valeur de l'étiquette associée à chaque module d'identification conditionne la façon dont se déroulera l'identification.

Ces valeurs sont étudiées ultérieurement.

Les modules d'identification ou LoginModule

Règles d'identification

ETIQUETTE	COMPORTEMENT
Required	Le module est indispensable au succès de l'authentification. Le processus d'authentification continue au module suivant quel que soit le résultat.
Requisite	Le module est indispensable au succès de l'authentification. Le processus d'authentification est interrompu en cas d'échec et le contrôle retourne à l'application.
Sufficient	Le module n'est pas indispensable au succès de l'authentification. Le processus d'authentification est interrompu en cas de succès et le contrôle retourne à l'application. En cas d'échec, le processus continue avec le module suivant.
Optional	Le module n'est pas indispensable au succès de l'authentification. Le processus d'authentification continue au module suivant quel que soit le résultat.

- l'authentification est un succès si TOUS les modules Required et Requisite réussissent
- si un module Sufficient est défini, tous les modules Required et Requisite rencontrés avant dans la liste des modules doivent réussir
- si aucun module Required ou Requisite n'est défini, l'authentification est un succès si au moins un module Sufficient ou Optional réussit

Les modules d'identification ou LoginModule

Notes

Les méthodes doAs() et doAsPrivileged()

```
static Object doAs(Subject subject, PrivilegedAction action)
static Object doAs(Subject subject, PrivilegedExceptionAction action)

static Object doAsPrivileged(Subject subject, PrivilegedAction action, AccessControlContext acc)
static Object doAsPrivileged(Subject subject, PrivilegedExceptionAction action,
                             AccessControlContext acc)
```

Caractéristiques communes

- Exécutent une action en appelant la méthode run() d'un objet de type PrivilegedAction ou PrivilegedExceptionAction
- Cette action retourne un résultat à l'appelant sous la forme d'un objet
- PrivilegedExceptionAction est à utiliser si l'action déclenche des exceptions que l'appelant souhaite traiter

Différences

- doAs() utilise le contexte courant (instance de AccessControlContext)
- doAsPrivileged() utilise un nouveau contexte

<http://java.sun.com/j2se/1.5.0/docs/guide/security/jaas/JAASRefGuide.html#doAsComp>

Les méthodes `doAs()` et `doAsPrivileged()`

Notes

Subject et Principal

Subject

- Entité concrète ou abstraite pouvant posséder plusieurs identités

Principal

- Identité d'un sujet
- Propriétés
 - Nom d'utilisateur
 - ID de groupe
 - un rôle

Subject et Principal

Notes

La gestion des permissions avec JAAS

Extension du modèle de sécurité introduit avec Java 2

- Gestionnaire de sécurité qui peut vérifier l'exécution d'opérations sensibles, notamment:
 - le chargement des classes
 - l'arrêt de la machine virtuelle
 - l'accès au système de fichier de la machine locale
 - l'ouverture de connexions réseau vers l'extérieur
 - l'accès au système graphique
 - l'accès aux files d'impression
- Droits accordés en fonction
 - de l'origine du code (URL)
 - de la fourniture ou non de certificats numériques
- Contrôle possible
 - avec une API dédiée et classes de permissions : `FilePermission`, `RuntimePermission`, etc ...
 - par la configuration de fichiers `.policy`

La gestion des permissions avec JAAS

Notes

Le modèle de sécurité de Java 2 permet d'accorder des permissions spécifiques à un code en fonction de son origine ou des certificats qu'il porte.

Il peut être activé selon les types d'applications:

- non activé par défaut pour une application autonome
- activé par défaut dans le cas d'une applet.

La gestion des permissions avec JAAS

Les points indispensables

- L'utilisateur doit être identifié à l'aide de JAAS
- Les fichiers de configuration de la politique de sécurité doivent être modifiés de façon à intégrer des règles basées sur l'identité
- Le sujet issu de l'identification doit invoquer un bloc de code privilégié en s'associant avec un `AccessControlContext` ou contexte de contrôle d'accès
- Le code privilégié est défini au sein de la méthode `run()` d'une instance de type
 - `PrivilegedAction`
 - `PrivilegedExceptionAction`

La gestion des permissions avec JAAS

Notes

AccessControlContext est abordé dans la suite de ce document.

AccessControlContext

Contexte de contrôle d'accès

- Objet associé à un thread d'exécution
- Contient des informations sur l'ensemble du code exécuté par ce thread
 - origine et localisation du code
 - permissions associées définies par les politiques de sécurité configurées à l'aide des fichiers .policy
- Propose une méthode checkPermission (Permission p)
 - la méthode ne retourne rien si l'accès demandé est autorisé
 - sinon, une exception AccessControlException est levée
- Peut être obtenu de la façon suivante

```
AccessControlContext acc = AccessController.getContext() ;
```

AccessControlContext

Notes

La méthode `AccessController.getContext()` permet d'obtenir une "photo" du contexte de contrôle d'accès courant.

La classe Policy

- Permet de spécifier des règles de sécurité à l'aide de fichiers de configuration
- Activée par défaut

Extrait du fichier \$JAVA_HOME/jre/lib/security/java.security

```
#  
# Class to instantiate as the system Policy. This is the name of the class  
# that will be used as the Policy object.  
#  
policy.provider=sun.security.provider.PolicyFile  
  
# The default is to have a single system-wide policy file,  
# and a policy file in the user's home directory.  
policy.url.1=file:${java.home}/lib/security/java.policy  
policy.url.2=file:${user.home}/.java.policy
```

- Il est possible de fournir une classe qui remplace la classe Policy définie par défaut mais il est beaucoup plus fréquent de la conserver
- Permet d'utiliser différents fichiers de configurations
 - au niveau de la machine virtuelle
 - par utilisateur
 - par application

La classe Policy

Notes

Les permissions définies par les différents fichiers (JVM, utilisateur, application) sont combinées.

`${user.home}` a pour valeur:

- `$HOME` sous Unix,
- le chemin d'accès au profil de l'utilisateur sous Windows.

Utiliser un fichier de règles spécifique

Ajouter des règles à celles définies par défaut

- Application

```
java -Djava.security.policy=MesRegles.policy MonApplication
```

- Applet

```
appletviewer -J-Djava.security.policy=MesRegles.policy MonApplet.html
```

Pour utiliser seulement les règles définies par un fichier

```
java -Djava.security.policy==MesRegles.policy MonApplication
```

Utiliser un gestionnaire de sécurité

- Activé par défaut avec certains types d'applications : applet, RMI dans certains cas.
- A activer avec les applications autonomes

- soit

```
System.setSecurityManager (new SecurityManager());
```

- soit

```
java -Djava.security.manager -Djava.security.policy=MyApp.policy MyApp
```

Utiliser un fichier de règles spécifique

Notes

Structure d'un fichier de règles

grant

```
grant sourcecode
{
    permission1;
    permission2;
    ...
}
```

sourcecode

```
codeBase "url"
```

- omis si les permissions sont accordées à tout code quelle que soit son origine
- url peut être distante (http) ou locale (file)

permission

```
permission ClassePermission Valeur , ListActions
```

- Valeur = valeur spécifique de permission : nom de fichier, numéro de port, ...
- ListActions = actions spécifiques à une permission : read, connect, execute, delete, ...

Structure d'un fichier de règles

Notes

Structure d'un fichier de règles

Exemple (fichier livré avec le JDK)

```
// Standard extensions get all permissions by default

grant codeBase "file:${java.ext.dirs}/*" {
    permission java.security.AllPermission;
};

grant {
    permission java.lang.RuntimePermission "stopThread";
    permission java.net.SocketPermission "localhost:1024-", "listen";
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version", "read";
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator", "read";
    permission java.util.PropertyPermission "path.separator", "read";
    permission java.util.PropertyPermission "line.separator", "read";

    ...
}
```

Structure d'un fichier de règles

Notes

Attribuer des permissions à des utilisateurs identifiés

Extension JAAS

- Accorder des permissions à des utilisateurs identifiés

Exemple

```
grant principal com.sun.security.auth.UnixPrincipal "gaston"
{
    permission java.util.PropertyPermission "user.*", "read" ;
};

grant principal com.sun.security.auth.NTUserPrincipal "robert"
{
    permission java.util.PropertyPermission "user.*", "read" ;
};
```

- Accorde à l'utilisateur Unix "gaston" et NT "robert" les permissions suivantes:
 - accès en lecture aux propriétés "user.*"

Attribuer des permissions à des utilisateurs identifiés

Notes

Ce mécanisme se base sur l'identité de l'utilisateur du programme telle qu'elle est connue par le système d'exploitation.

Les différents types de Principal

Java 5.0

<code>javax.security.auth.kerberos.KerberosPrincipal</code>	Encapsulation d'une identité Kerberos
<code>javax.security.auth.x500.X500Principal</code>	Identité annuaire X500
<code>javax.management.remote.JMXPrincipal</code>	Utilisateur distant à travers l'API JMX
<code>com.sun.security.auth.NTUserPrincipal</code>	Utilisateur sur un système Windows NT et dérivés
<code>com.sun.security.auth.UnixPrincipal</code>	Utilisateur sur un système de type Unix

Les différents types de Principal

Notes

Les callbacks JAAS

Utilité

- Certains modules d'identification obtiennent leurs informations directement auprès du système d'exploitation
 - UnixLoginModule
 - NTLoginModule
- D'autres ont besoin d'informations fournies interactivement par l'utilisateur

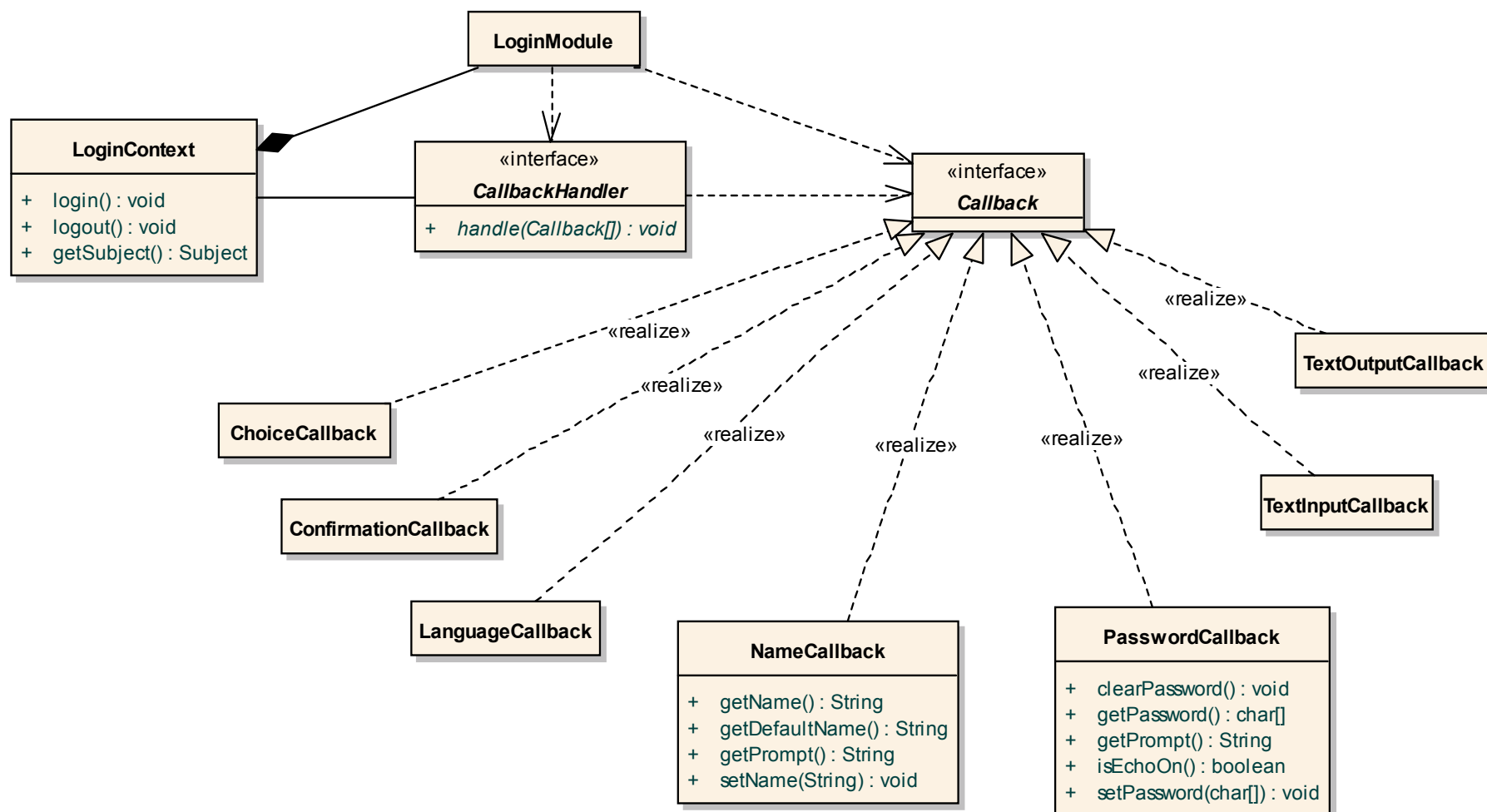
Mécanisme

- Une instance de CallbackHandler est passée en paramètre à la construction d'un LoginContext
- Cet objet est communiqué à tous les modules d'identification
- En cas de besoin, ces modules extraient des informations d'objets de type Callback renseignés par le CallbackHandler

Les callbacks JAAS

Notes

javax.security.auth.callback



Toutes les méthodes ne sont pas montrées sur ce schéma.

javax.security.auth.callback

Notes

L'instance de CallbackHandler est transmise aux modules d'identification.

Ceux-ci peuvent alors invoquer la méthode handle(Callback[]) en fournissant un tableau de Callbacks que le handler pourra renseigner.

Les exigences en termes de Callbacks varient suivant le LoginModule utilisé.

Par exemple, le module d'identification JNDI exige (comme de nombreux autres types de modules d'identification) la prise en charge par le CallbackHandler des callbacks suivantes:

- NameCallback
- PasswordCallback

javax.security.auth.callback

Les différents types de Callbacks

NameCallback	Callback la plus utilisée, permet de communiquer un nom d'utilisateur aux modules d'identification.
PasswordCallback	Egalement très utilisée, permet de communiquer un mot de passe aux modules d'identification.
TextInputCallback	Permet à un LoginModule de récupérer un texte fourni par l'utilisateur.
TextOutputCallback	Permet à un LoginModule de communiquer de l'information à l'utilisateur.
ChoiceCallback	Présente un ensemble d'alternatives à l'utilisateur et lui permet d'en choisir une ou plusieurs.
ConfirmationCallback	Présente à l'utilisateur une demande de confirmation attendant une réponse de type "oui/non", "oui/non/annuler" ou "oui/annuler".
LanguageCallback	Utilisée lorsqu'un module d'identification doit déterminer selon quelle localisation l'utilisateur doit être identifié.

- Le CallbackHandler doit prévoir la gestion de tous ces cas
- Ou lever une UnsupportedOperationException

javax.security.auth.callback

Notes

Plusieurs types sont inclus par défaut dans le JDK.

Il est possible de définir des types de callbacks supplémentaires en implémentant l'interface Callback.

Un CallbackHandler

Exemple

```
public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
        if (callbacks[i] instanceof NameCallback) {
            // prompt the user for a username
            NameCallback nc = (NameCallback)callbacks[i];
            // ignore the provided defaultName
            System.err.print(nc.getPrompt());
            System.err.flush();
            nc.setName((new BufferedReader (new InputStreamReader(System.in))).readLine());
        } else if (callbacks[i] instanceof PasswordCallback) {
            // prompt the user for sensitive information
            PasswordCallback pc = (PasswordCallback)callbacks[i];
            System.err.print(pc.getPrompt());
            System.err.flush();
            pc.setPassword(readPassword(System.in));
        } else {
            throw new UnsupportedCallbackException (callbacks[i], "Unrecognized Callback");
        }
    }
}
```

Un CallbackHandler

Notes

Ecrire un module d'identification

Besoin

- Utiliser un système d'identification non supporté par les modules fournis en standard

Etapes

- Implémenter l'interface `javax.security.auth.spi.LoginModule`

<code>public void initialize</code> (Subject, CallbackHandler, Map sharedState, Map options)	Initialise le module. Le sujet doit être sauvé. options contient les options du fichier de configuration sharedState est un cache de résultats intermédiaires d'identifications.
<code>public boolean login()</code>	Authentifie l'utilisateur et retourne true en cas de succès, false sinon
<code>public boolean commit()</code>	Méthode appelée quand l'utilisateur est authentifié par tous les modules. Un Principal approprié doit être associé au sujet et true est retourné. Si le stockage est impossible, false est retourné.
<code>public boolean abort()</code>	Méthode appelée quand l'utilisateur ne peut pas être authentifié.
<code>public boolean logout()</code>	Les instances de Principal associées au sujet par ce module doivent être supprimées.

- Sélectionner ou concevoir une classe de Principal adaptée

Ecrire un module d'identification

Notes

Ce besoin survient dès qu'il est nécessaire d'utiliser un système d'authentification spécifique:

- autre application,
- base de données,
- ...

Ecrire un module d'identification

Une classe de Principal

```
import java.io.Serializable;
import java.security.Principal;

public class SimplePrincipal implements Principal, Serializable {
    private String name;

    public SimplePrincipal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public boolean equals (Object o) {
        if (o instanceof Principal) {
            return (((Principal)o).getName().equals(name));
        } else return false;
    }
}
```

Ecrire un module d'identification

Notes

Ecrire un module d'identification

SimpleLoginModule 1/2

```
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

public class SimpleLoginModule implements LoginModule {
    private Subject subject;
    private CallbackHandler callbackHandler;
    private SimplePrincipal principal;
    private String username;
    private boolean succeeded = false;
    private boolean commitSucceeded = false;

    public void initialize (Subject s, CallbackHandler cb, Map sharedMap, Map options) {
        subject = s;
        callbackHandler = cb;
    }

    public boolean login() throws LoginException {
        username = "toto";
        succeeded = true;
        return true;
    }
}
```


Ecrire un module d'identification

Notes

Ecrire un module d'identification

SimpleLoginModule 2/2

```
public boolean commit() throws LoginException {
    if (!succeeded) {
        username = null;
        return false;
    }
    principal = new SimplePrincipal(username);
    if (!subject.getPrincipals().contains(principal)) {
        subject.getPrincipals().add(principal);
    }
    username = null;
    commitSucceeded = true;
    return true;
}

public boolean abort() throws LoginException {
    if (!succeeded)
        return false;
    else if (commitSucceeded)
        logout();
    else {
        succeeded = false;
    }
    return true;
}

public boolean logout() throws LoginException {
    subject.getPrincipals().remove(principal);
    principal = null;
    username = null;
    succeeded = commitSucceeded = false;
    return true;
}
```

Ecrire un module d'identification

Notes

Java, Programmation avancée

Le ClassLoader Java

Version 2.2

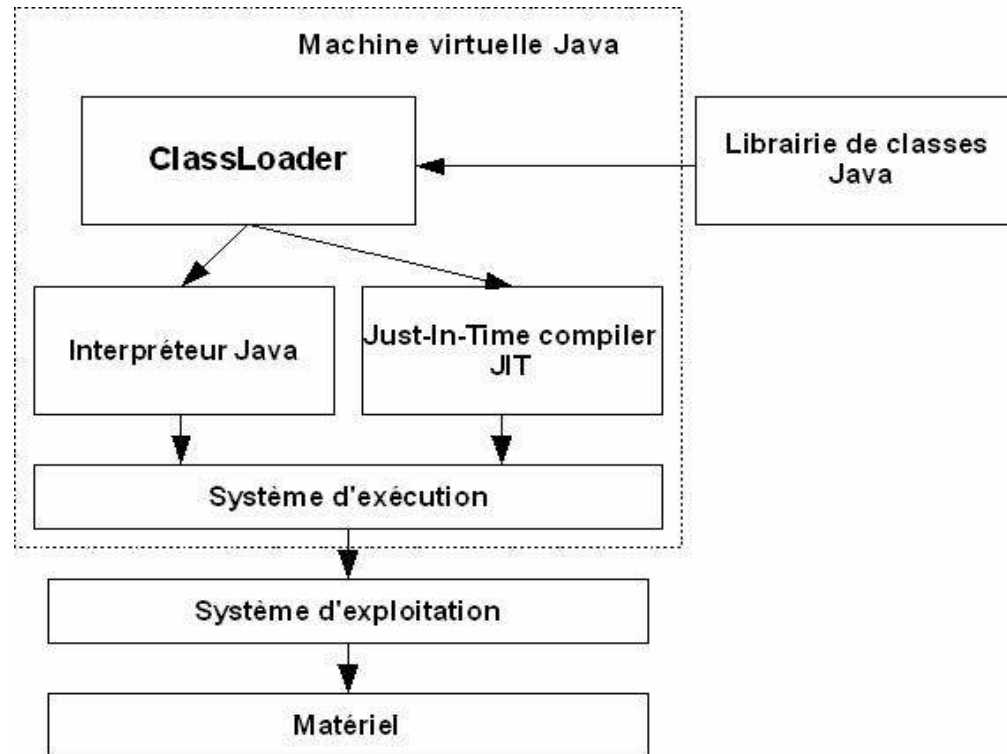
- Mécanisme de chargement des classes
- Rôle du ClassLoader
- Créer un Classloader
- ...

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Le chargeur de classes (classloader)

Fait partie de la machine virtuelle JVM



Rappel de l'architecture de la JVM

- A pour but de charger les classes
- S'assure que les classes ne sont chargées qu'une fois

Le chargeur de classes (classloader)

Notes :

Mécanisme de chargement des classes

La machine virtuelle (VM) ne charge que les fichiers de classes nécessaires

- La VM possède un mécanisme de chargement des fichiers de classe
 - Ressources locales
 - Ressources Web
- Chargement de toutes les classes dont dépend la classe chargée (résolution de classe)
- La VM exécute la méthode main
- Chargement des classes supplémentaires nécessaires à la méthode main ou aux méthodes appelées par main

Le chargement de classes utilise plusieurs chargeurs de classe (au moins trois)

- Le chargeur de classe d'amorce
 - Charge les classes systèmes
 - Fait partie de VM
 - Implémenté en C
 - Pas d'objet `ClassLoader` correspondant au chargeur de classe d'amorce

```
System.out.println(String.class.getClassLoader());  retourne null
```

```
System.out.println(Class.class.getClassLoader());   retourne sun.misc.Launcher$AppClassLoader@xxxxx
```


Mécanisme de chargement des classes

Notes :

Mécanisme de chargement des classes

- Le chargeur de classe d'extension
 - Charge les extensions standard (qui se trouvent dans le répertoire jre/lib/ext)
 - Ecrit en Java dans l'implémentation de Sun
- Le chargeur de classe système (parfois appelé chargeur de classe d'application
 - Charge les classes de l'application
 - Effectue la recherche des classes en se basant sur le \$CLASSPATH
 - Ecrit en Java dans l'implémentation de Sun

Les chargeurs de classes autres que le chargeur d'amorce ont une relation parent/enfant

- Chaque chargeur de classe possède un chargeur de classe parent
- Il donne à son parent la possibilité de charger toutes les classes indiquées
 - En cas d'échec du parent, c'est lui qui s'occupe du chargement

Mécanisme de chargement des classes

Notes :

L'absence de classe comme atout

La recherche des .class dans les répertoires et archives prend du temps

- Le classloader doit ouvrir toutes les archives et consulter les index
- L'absence de classe est constatée
 - après ouverture de l'ensemble des archives
 - après le parcours de l'ensemble des répertoires

Certaines API Java utilisent l'absence de classe comme information pertinente

- Le mécanisme de gestion des ressources en plusieurs langues par exemple

```
ResourceBundle.getResourceBundle("MaRessource", lang1)
```

- Recherche de la ressource en commençant par la langue demandée

```
"MaRessource_" + lang1 + "_" + pays1 + "_" + variante1  
"MaRessource_" + lang1 + "_" + pays1 + "_" + variante1 + ".properties"  
"MaRessource_" + lang1 + "_" + pays1  
"MaRessource_" + lang1 + "_" + pays1 + ".properties"  
"MaRessource_" + lang1  
"MaRessource_" + lang1 + ".properties"
```

L'absence de classe comme atout

Notes :

L'absence de classe comme atout

- Parcoure la suite des fichiers de langue

```
"MaRessource_" + lang2 + "_" + pays1 + "_" + variante2  
"MaRessource_" + lang2 + "_" + pays1 + "_" + variante2 + ".properties"  
"MaRessource_" + lang2 + "_" + pays1  
"MaRessource_" + lang2 + "_" + pays1 + ".properties"  
"MaRessource_" + lang2  
"MaRessource_" + lang2 + ".properties"
```

- Recherche ensuite la ressource avec la langue par défaut

```
"MaRessource"  
"MaRessource.properties"
```

L'absence de classe comme atout

Notes :

L'utilisation de chargeurs de classe comme espace de nom

Deux classes de la même machine virtuelle :

- peuvent avoir les mêmes nom de classe
- peuvent avoir les mêmes nom de package

Une classe se détermine par son nom complet et par son chargeur de classes

Exemple d'utilisation

- Un navigateur utilise plusieurs instances du chargeur de classe d'applet pour chaque page Web
- Permet de séparer des classes de différentes pages Web
- Quel que soit leur nom

<http://java.sun.com/developer/TechTips/2000/tt1027.html>

L'utilisation de chargeurs de classe comme espace de nom

Notes :

Création d'un classloader

Deux étapes

- Etendre la classe ClassLoader
- Surcharger la méthode findClass(String name)

```
protected Class<?> findClass(String name) throws ClassNotFoundException {  
  
    byte[] b = loadClassData(name);  
    return defineClass(name, b, 0, b.length);  
}  
  
private byte[] loadClassData(String name)  
    throws ClassNotFoundException {  
    try {  
        String filename = name.replace('.', '/') + ".class";  
        InputStream in = new FileInputStream(filename);  
        byte[] data = new byte[in.available()];  
        in.read(data);  
        in.close();  
  
        return data;  
    } catch (FileNotFoundException x) {  
        throw new ClassNotFoundException(name, x);  
    } catch (IOException x) {  
        throw new ClassNotFoundException(name, x);  
    }  
}
```

Chargement des classes

- La méthode loadClass de ClassLoader prend soin de la délégation au parent
- N'appelle findClass que si la classe n'a pas été chargée et que le parent n'a pas réussi à la charger

Création d'un classloader

Notes :

Création d'un classloader

Exemple d'utilisation d'un chargeur de classe personnalisé

```
package com.leuville.test;

public class ClassToLoad {
    public ClassToLoad() {
        System.out.println("Constructeur de ClassToLoad");
    }
}

public class TestLoader {

    public static void main(String[] args) {
        ClassLoader loader = new MyLoader();
        try {
            Class<?> c = loader.loadClass("com.leuville.test.ClassToLoad");
            c.newInstance();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
    }
}
```

Création d'un classloader

Notes :

Java, Programmation avancée

Références

Version 2.2

- Bibliographie
- Sites Internet

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Concepts Objet

Site	URL
Object Management Group	www.omg.org
Cetus-links : collection de références sur l'objet	www.cetus-links.org
La Boite à Objets	www.stm.tj/objet/
Lieu d'Informations sur le Génie Logiciel Orienté Objet	www.laas.fr/~delatour/Igloo/index_image_fr.html
Cours sur l'Objet	www.laas.fr/~delatour/Igloo/coursOO_image_fr.html
The Object Oriented Page	www.well.com/user/ritchie/oo.html
Informations sur l'objet	iamwww.unibe.ch/~scg/OOinfo/
Merise et extensions Objet	perso.wanadoo.fr/guezelou/mcd/mcd.htm
SGBDOO	pages.infinet.net/map/present.html
Forums de discussion	fr.comp.lang.objet
Liste de diffusion francophone	www.eGroups.com/list/objet www.francopholistes.com

Notes

UML

Site	URL
Standard UML de l'OMG	www.omg.org/uml
Centre de ressources UML Rational Software	www.rational.com/uml/index.jtмл
UML en français	uml.free.fr
Introduction à UML	www.commentcamache.net/uml/umlintro.php3
Cours Université de Nantes	iae.univ-nantes.fr/insertio/pages_etudiant/uml/info_uml/documl.htm

Notes

Langages

Site	URL
C++	www.cetus-links.org/oo_c_plus_plus.html www.cs.wustl.edu/~schmidt/C++/index.html www.icce.rug.nl/docs/cplusplus/cplusplus.html www.sgi.com/Technology/STL/
Forums de discussion C++	fr.comp.lang.c++ comp.lang.c++
JAVA	www.javasoft.com www.ibm.com/java/ www.developer.com www.javaworld.com
Forums de discussion JAVA	fr.comp.lang.java comp.lang.java.* (14 groupes)
Listes de diffusion JAVA	pause-java.u-strasbg.fr (JAVA) www.leuville.com (EJB & J2EE) www.javasoft.com (listes anglophones)
VisualBasic	msdn.microsoft.com/vbasic/ www.vbwm.com/
Forums de discussion VisualBasic	microsoft.public.vb.* (27 groupes)
XML - XSL	www.w3schools.com

Notes

Architectures réparties

Site	URL
CORBA	www.omg.org/corba/ www.corba.org
DCOM	www.microsoft.com/com/default.asp
RMI	www.javasoft.com/api/rmi

Notes

Objets métier et composants

Site	URL
Enterprise JavaBeans	www.javasoft.com/products/ejb/index.html www.javasoft.com/products/ejb/ejbvscom.html www.jguru.com www.ejbportal.com www.ejbnow.com
Distributed interNet Architecture	www.microsoft.com/dna/tech.asp www.microsoft.com/com/wpaper/mts-ejb.asp
CORBA Component Model	www.omg.org
.NET	msdn.microsoft.com/net
OSGi	www.osgi.org www.vogella.com/articles/OSGiServices/article.html

Notes

Web Services

Site	URL
Spécifications SOAP	http://www.w3.org/TR/soap/
Spécifications WSDL	http://www.w3.org/TR/wsdl
Spécifications UDDI	http://www.oasis-open.org/specs/
Spécifications XML	http://www.w3.org/TR/REC-xml/

Notes

SOA

Site	URL
Définitions Wikipedia d'une architecture SOA	http://fr.wikipedia.org/wiki/Architecture_orient%C3%A9e_services
Définitions Wikipedia d'un bus ESB	http://fr.wikipedia.org/wiki/Enterprise_Service_Bus
WS-BPEL	http://www.ibm.com/developerworks/library/specification/ws-bpel/
WS-Security	http://msdn.microsoft.com/en-us/library/ms951273.aspx
WS-Coordination	http://www.ibm.com/developerworks/library/specification/ws-tx/
Blog de retours d'expériences sur les bus ESB	http://enterpriseservicebus.blogspot.com/index.html

Notes

Bibliographie

Sources

- Le Monde en Tique
www.lmet.fr
- Amazon
www.amazon.fr

Notes

Objet, UML, design patterns

- Modélisation objet avec UML
Pierre-Alain Muller
Eyrolles
- UML - La notation unifiée de modélisation objet - Applications en JAVA
Michel Lai
InterEditions
- Le Processus Unifié de développement logiciel
Ivar Jacobson
Grady Booch
James Rumbaugh
Eyrolles
- Intégrer UML dans vos projets
Nathalie Lopez, Jorge Migueis, Emmanuel Pichon
Eyrolles
- Design Patterns - Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Addison-Wesley
- Patterns in Java
Mark Grand
Wiley

Notes

Objet, UML, design patterns

- The Unified Modeling Language User Guide
Booch, Rumbaugh, Jacobson
Addison-Wesley
- Objects, Components, and Frameworks with UML
D'Souza, Wills
Addison-Wesley
- Applying UML and PATTERNS
Larman
Prentice Hall

Notes

CORBA

- Instant CORBA
Orfali, Harkey, Edwards
Wiley
- Client/server programming with Java and CORBA
Orfali, Harkey
Wiley

Notes

Langages

- Pour mieux développer avec C++
Aurélien Geron, Fatmé Tawbi
InterEditions
- Le Développeur Java 2 - Mise en oeuvre et solutions - Les Applets
Auteur(s) : A . Mirecourt
OEM - Eyrolles, 9/1999
- Dictionnaire officiel Java 2
Eyrolles, 5/1999
- Les langages objets Smalltalk - Java - C++
Auteur(s) : P. Prados
Eyrolles , 4 / 1998
- Java Embarqué
Auteur(s) : Yves Bossu, Cédric Nicolas, Albert Proust et Jean-Bernard Blanchet
Eyrolles, 2/1999
- Java client-serveur
Auteur(s) : C. Avare, C. Nicolas, F. Najman
Eyrolles , 11 / 1997

Notes

Langages

- Le Développeur Java 2
Edition 1999, version finale du JDK 1.2 incluse
Auteur(s) : Mirecourt
OEM / Osman Eyrolles Multimédia
- Corba, ActiveX et Java Beans
Auteur(s) : J. Chauvet
Cmp , 4 / 1997
- Java Beans
Auteur(s) : R. Englander
O'Reilly France , 12 / 1997
- Java par la pratique
Auteur(s) : J. Peck, P. Niemeyer
O'Reilly France , 1 / 1997
- Programmation réseau avec Java
Auteur(s) : E. Harold
O'Reilly France , 11 / 1997
- Java in a Nutshell
Auteur(s) : D. Flanagan
O'Reilly France , 1 / 1997

Notes

Langages

- Au coeur de Java - Volume 1 - Notions fondamentales
Auteur(s) : Cay S. Horstmann et Gary Cornell
Person Education, 04/2008
- Real-Time Java Programming With Java RTS
Auteur(s) : Eric J. Bruno et Greg Bolleta
Prentice Hall Education 05/2009

Notes

Objets métier et composants

- Objets Métier
Thierry Ando, Jean-Marie Chauvet
Eyrolles
- Enterprise JavaBeans
Richard Monson-Haefel
O'Reilly
- OSGi - Conception d'applications modulaires en Java
Jérôme Molière
Eyrolles

Notes

Web Services

- Les Web Services - Techniques, démarches et outils XML WSDL SOAP Rosetta UML
Hubert Kadima, Valérie Monfort
Dunod

Notes

SOA

- Service-Oriented Architecture - Concepts, Technology and Design
Thomas Erl
Prentice Hall
- Enterprise Service Bus
David A. Chappell
O'Reilly

SOA

Notes

Embarqué

- L'art du développement Android 2ème édition
Mark Murphy
PEARSON

Embarqué

Notes

Java, Programmation avancée

Annexes

Version 2.2

- Introspection

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Java, Programmation avancée

L'API d'inspection

Version 2.2

- Comprendre le mécanisme d'inspection et les classes mises en oeuvre
- Utiliser l'inspection pour manipuler des objets

(c) Leuville Objects. Tous droits de traduction, d'adaptation et de reproduction par tous procédés, réservés pour tous pays.

Toute reproduction ou représentation intégrale ou partielle, par quelque procédé que ce soit des pages publiées dans le présent ouvrage, faite sans l'autorisation de Leuville Objects, est illicite et constitue une contrefaçon (loi du 11 mars 1957 et code de la propriété intellectuelle du 1er juillet 1992, articles L 122-4, L 122-5 et L 335-2).

Utilité de l'introspection

- Permet d'examiner ou de modifier le comportement d'une application tournant au sein d'une JVM.
- Il s'agit d'une fonctionnalité avancée du langage Java.

Extensibilité

- L'introspection permet d'utiliser des classes développées par un utilisateur, simplement en indiquant le nom de la classe (principe des frameworks).

Recherche de classes / Environnement de développement

- Les environnements de développement Java sont capables d'indiquer de façon graphique le contenu (membres) d'une classe. Ceci est possible grâce à l'introspection.

Debuggers, outils de tests

- Lors de tests ou de débogage, il peut être nécessaire de passer outre les limitations de visibilité des membres d'une classe.

Utilité de l'inspection

Notes

Les inconvénients de l'introspection

- L'introspection est puissante mais n'a pas que des avantages.
- Si une opération est possible sans l'usage de l'introspection, il est préférable d'éviter de l'utiliser.

Diminution des performances

- La résolution dynamique des types empêche certains mécanismes d'optimisation d'être effectué.
- Un traitement utilisant l'introspection est toujours moins rapide que son équivalent sans introspection.

Resctrictions de sécurité

- L'usage de l'introspection doit être autorisée par le SecurityManager pour fonctionner.

Rupture des règles de visibilité

- L'API d'introspection permet d'effectuer des opérations illégales en tant normal.
- Il est ainsi possible d'accéder à des membres privés d'une classe (modification de la valeur d'un attribut sans passer par un accesseur visible).
- Dans ces conditions, les effets de bord de l'usage de l'introspection sont non négligeables et peuvent aller jusqu'à un dysfonctionnement de l'application.

Les inconvénients de l'introspection

Notes

Manipulation des classes

- En Java, tout objet est soit une référence, soit un type primitif.
- Les classes, énumérations, tableaux et interfaces sont des références et héritent de la classe `java.lang.Object`.
- Pour chaque type primitif, il existe une classe d'encapsulation (wrapper).

Toute classe est un objet

- Pour chaque type, la JVM crée une instance immuable de la class `java.lang.Class`.
- Cette classe fournit les méthodes nécessaires à la découverte des membres de la classe qu'elle représente.
- Cette classe est le point de départ de l'API d'introspection.

Manipulation des classes

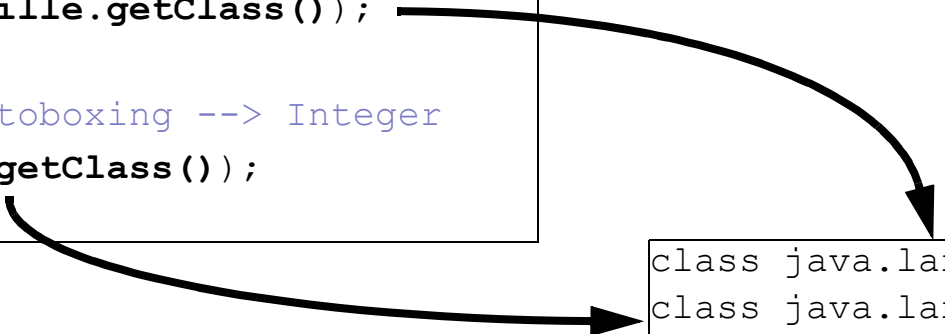
Notes

Manipulation des classes

Récupération d'une instance de Class à partir d'un objet

- Pour récupérer la classe dont un objet est type, il suffit d'invoquer la méthode `getClass()` sur cet objet.
- L'existence de cette méthode est garantie car elle est définie dans la classe `java.lang.Object`.

```
public static void main(String[] args) {  
    String leuville = "Leuville Objects";  
    System.out.println(leuville.getClass());  
  
    Object val = 123; // Autoboxing --> Integer  
    System.out.println(val.getClass());  
}
```



```
class java.lang.String  
class java.lang.Integer
```

Manipulation des classes

Notes

Manipulation des classes

Récupération d'une instance de Class par référence à la classe

- Une autre façon de récupérer une instance de `Class` est d'utiliser la syntaxe `.class`.
- Pour cela, il suffit d'ajouter `.class` après le nom de la classe.

```
public class TestDotClass {  
  
    public static void main(String[] args) {  
        System.out.println(String.class);  
        System.out.println(Integer.class);  
        System.out.println(WebService.class);  
        System.out.println(TestDotClass.class);  
    }  
}
```

```
class java.lang.String  
class java.lang.Integer  
interface javax.jws.WebService  
class TestDotClass
```


Manipulation des classes

Notes

Manipulation des classes

Récupération d'une instance de Class à partir du nom de la classe

- La méthode statique `forName(String)` de la classe `Class` permet d'obtenir une instance de la classe `Class` en spécifiant le nom de la classe souhaitée.

```
public static void main(String[] args) {  
    try {  
        System.out.println(Class.forName("java.lang.String"));  
        System.out.println(Class.forName("[I"]); // Tableau de int  
        System.out.println(Class.forName("com.leuville.notAClass"));  
    } catch (ClassNotFoundException e) {  
        e.printStackTrace(System.out);  
    }  
}
```

```
class java.lang.String  
class [I  
java.lang.ClassNotFoundException: com.leuville.notAClass  
at java.net.URLClassLoader$1.run(Unknown Source)
```

Manipulation des classes

Notes

Manipulation des classes

Méthodes retournant des instances de Class

- Ces méthodes sont invoquables sur une instance de la classe `Class`.
 - `getSuperClass` : renvoie la classe mère de la classe.
 - `getClasses` : renvoie l'ensemble des classes, interfaces et énumérations membre de la classe (y compris celles héritées).
 - `getDeclaredClasses` : renvoie l'ensemble des classes, interfaces et énumérations membre de la classe.
 - `getEnclosingClass` : renvoie la classe au sein de laquelle la classe courante est déclarée.
- Il est également possible d'obtenir une instance de la classe `Class` en invoquant la méthode `getDeclaringClass` sur un champ (`Field`), une méthode (`Method`) ou un constructeur (`Constructor`). On obtient alors la classe déclarant ce champ / méthode / constructeur.

Manipulation des classes

Notes

Manipulation des classes

Informations sur une classe

- Les méthodes de `Class` permettent d'obtenir certaines informations sur une classes :
 - `getCanonicalName()` : renvoie le nom canonique (avec package) de la classe.
 - `getName()` : renvoie le nom de la classe.
 - `getPackage()` : renvoie le package auquel la classe appartient.
 - `isAnnotation()` : indique si la classe est une annotation.
 - `isAnonymous()` : indique si la classe est anonyme.
 - `isArray()` : indique si la classe représente un tableau.
 - `isEnum()` : indique si la classe est une énumération.

Manipulation des classes

Notes

Manipulation des classes

Découverte des membres d'une classe

- Le tableau ci dessous indique les méthodes de la classe Class permettant de découvrir les membres d'une classe :

Type de membre	Méthode	Liste de membres ?	Membres hérités ?	Membres privés ?
Champ (Field)	getDeclaredField()	Non	Non	Oui
	getField()	Non	Oui	Non
	getDeclaredFields()	Oui	Non	Oui
	getFields()	Oui	Oui	Non
Méthode (Method)	getDeclaredMethod()	Non	Non	Oui
	getMethod()	Non	Oui	Non
	getDeclaredMethods()	Oui	Non	Oui
	getMethods()	Oui	Oui	Non
Constructeur (Constructor)	getDeclaredConstructor()	Non	-	Oui
	getConstructor()	Non	-	Non
	getDeclaredConstructors()	Oui	-	Oui
	getConstructors()	Oui	-	Non

Manipulation des classes

Notes

Manipulation des classes

Exemple

```
package com.leuville.exemple1;

public class Compte {

    private float solde;
    private String rib;
    private String nomClient;

    public Compte(String rib, String nomClient) {
        this.rib = rib;
        this.nomClient = nomClient;
    }

    public void effectuerDepot(float montant) {
        solde += montant;
    }

    public boolean effectuerRetrait(float montant)
    {
        if (solde <= montant) {
            solde -= montant;
        }
    }
}
```

```
package com.leuville.exemple1;

public class CompteRemunere extends
Compte {

    private float taux;

    public CompteRemunere(String rib,
        String nomClient, float
taux) {
        super(rib, nomClient);
        this.taux = taux;
    }

    public float getTaux() {
        return taux;
    }

    ...
}
```

Manipulation des classes

Notes

Manipulation des classes

Exemple

```
public static void displayClassInfos(String className) throws ClassNotFoundException {
    Class c = Class.forName(className);
    System.out.println("Name = " + c.getName());
    System.out.println("Canonical name = " + c.getCanonicalName());
    System.out.println("Simple name = " + c.getSimpleName());
    System.out.println("Super class = " + c.getSuperclass());
    System.out.println("Nombre de méthodes déclarées = " + c.getDeclaredMethods().length);
    System.out.println("Nombre total de méthodes = " + c.getMethods().length);
    System.out.println("Nombre de champs déclarés = " + c.getDeclaredFields().length);
    System.out.println("Nombre total de champs = " + c.getFields().length);
    System.out.println("Nombre de constructeurs = " + c.getConstructors().length);
}

public static void main(String[] args) {
    try {
        displayClassInfos("com.leuville.exemple1.Compte");
        System.out.println("-----");
        displayClassInfos("com.leuville.exemple1.CompteRemunere");
    } catch (Throwable e) {
        e.printStackTrace();
    }
}
```

Manipulation des classes

Notes

Manipulation des classes

Exemple

```
Name = com.leuville.exemple1.Compte  
Canonical name = com.leuville.exemple1.Compte  
Simple name = Compte  
Super class = class java.lang.Object  
Nombre de méthodes déclarées = 8  
Nombre total de méthodes = 17  
Nombre de champs déclarés = 3  
Nombre total de champs = 0  
Nombre de constructeurs = 1
```

```
Name = com.leuville.exemple1.CompteRemunere  
Canonical name = com.leuville.exemple1.CompteRemunere  
Simple name = CompteRemunere  
Super class = class com.leuville.exemple1.Compte  
Nombre de méthodes déclarées = 2  
Nombre total de méthodes = 19  
Nombre de champs déclarés = 1  
Nombre total de champs = 0  
Nombre de constructeurs = 1
```

Manipulation des classes

Notes

Manipulation des champs

- La classe `java.lang.reflect.Field` permet d'obtenir des informations sur un champ d'une classe.
- Elle permet également d'accéder à la valeur de ce champ en lecture ou en écriture.

Obtenir une instance de `Field`

- L'obtention d'une instance de `Field` se fait par l'invocation d'une méthode de `Class` :
 - `getFields()` : retourne l'ensemble des champs publics de la classe (y compris ceux hérités).
 - `getField(String)` : retourne le champ public de la classe dont le nom est spécifié (la recherche se fait aussi dans les champs publics hérités).
 - `getDeclaredFields()` : retourne l'ensemble des champs déclarés dans la classe.
 - `getDeclaredField(String)` : retourne le champ déclaré dans la classe dont le nom est spécifié.

Manipulation des champs

Notes

Manipulation des champs

Obtenir des informations sur un champ

- Les méthodes suivantes de la classe `Field` permettent d'obtenir des informations sur le champ :
 - `getName()` : nom du champ.
 - `getModifiers()` : ensemble des modificateurs (cf `java.lang.reflect.Modifier`) appliqués au champ.
 - `getType()` : type (`Class`) du champ

Manipulation des champs

Notes

Manipulation des champs

Lire et écrire la valeur d'un champs

- Pour lire la valeur d'un champ :
 - `get(Object)` : renvoie la valeur du champ pour l'instance passée en paramètre.
 - `getXxx(Object)` : renvoie la valeur du champ, castée en type primitif, pour l'instance passée en paramètre.
- Pour écrire la valeur d'un champ :
 - `set(Object, Object)` : affecte une valeur (2eme paramètre) au champ pour une instance donnée (1er paramètre).
 - `setXxx(Object, Xxx)` : affecte une valeur typée (2eme paramètre) au champ pour une instance donnée (1er paramètre).
- Une exception de `IllegalAccessException` est levée si l'accès direct au champ est interdit.
- Il est possible de vérifier l'accessibilité avec la méthode `isAccessible()`, et de la lever avec la méthode `setAccessible(boolean)`.

Manipulation des champs

Notes

Manipulation des méthodes

- La classe `java.lang.reflect.Method` représente une méthode d'une classe.
- A partir d'une instance de cette classe, il est possible d'obtenir des informations sur la méthode, mais également de l'invoquer.

Obtenir une instance de `Method`

- L'obtention d'une instance de `Method` se fait par l'invocation d'une méthode de `Class` :
 - `getMethods()` : retourne l'ensemble des méthodes publiques de la classe (y compris celles héritées).
 - `getMethod(String, Class<?>...)` : retourne la méthode de la classe dont le nom et les paramètres sont spécifiés (la recherche se fait aussi dans les méthodes publiques héritées).
 - `getDeclaredMethods()` : retourne l'ensemble des méthodes déclarées dans la classe.
 - `getDeclaredMethod(String, Class<?>...)` : retourne la méthode déclarée dans la classe dont le nom et les paramètres sont spécifiés.

Manipulation des méthodes

Notes

Manipulation des méthodes

Obtenir des informations sur une méthode

- A partir d'une instance de `Method` :
 - `getName()` : retourne le nom de la méthode.
 - `getModifiers()` : retourne les modificateurs appliqués à la méthode.
 - `getParameterTypes()` : retourne le type de chaque paramètre attendu par la méthode.
 - `getReturnType()` : retourne le type de la valeur de retour de la méthode.
 - `isBridge()` : indique si la méthode est de type "bridge", tel que défini dans la spécification du langage.
 - `isSynthetic()` : indique si la méthode est "synthétique", tel que défini dans la spécification du langage.
 - `isVarArgs()` : indique si la méthode a été déclarée comme acceptant un nombre variable d'arguments.
 - `getExceptionTypes()` : retourne le type des exceptions lancées par la méthode.

Manipulation des méthodes

Notes

Manipulation des méthodes

Invocation d'une méthode

- A partir d'une instance de la classe Method :
- `invoke(Object, Object...)` : permet d'invoquer la méthode, portant sur l'objet indiqué en premier paramètre, en utilisant les arguments indiqués par les paramètres suivants.
- Si la méthode est statique, il n'est pas besoin d'instance de l'objet pour l'invoquer, le premier paramètre peut donc être valorisé à `null`.

```
Compte compte = new Compte("1234", "Dupont");

Class cptClass = Compte.class;
Method depotMethod = cptClass.getMethod("effectuerDepot", float.class);
depotMethod.invoke(compte, 1000f);
Method soldeMethod = cptClass.getMethod("getSolde");
float solde = (Float) soldeMethod.invoke(compte);
System.out.println(solde);
```

Manipulation des méthodes

Notes

Manipulation des constructeurs

- La classe `java.lang.reflect.Constructor` représente un constructeur d'une classe.
- A partir d'une instance de cette classe, il est possible d'obtenir des informations sur le constructeur, et également de l'invoquer pour créer une instance de la classe.

Obtenir une instance de Constructor

- L'obtention d'une instance de `Constructor` se fait par l'invocation d'une méthode de `Class` :
 - `getConstructors()` : retourne l'ensemble des constructeurs publics de la classe.
 - `getConstructor(Class<?>...)` : retourne le constructeur de la classe dont les paramètres sont spécifiés.
 - `getDeclaredConstructors()` : retourne l'ensemble des constructeurs déclarés dans la classe.
 - `getDeclaredConstructor(Class<?>...)` : retourne le constructeur déclaré dans la classe et dont les paramètres sont spécifiés.

Manipulation des constructeurs

Notes

Manipulation des constructeurs

Obtenir des informations sur un constructeur

- A partir d'une instance de `Constructor` :
 - `getExceptionTypes()` : retourne le type des exceptions lancées par le constructeur.
 - `getModifiers()` : retourne les modificateurs appliqués au constructeur.
 - `getParameterTypes()` : retourne le type de chaque paramètre attendu par le constructeur.

Manipulation des constructeurs

Notes

Manipulation des constructeurs

Invoquer un constructeur

- Si la classe dispose d'un constructeur sans paramètre :
 - la méthode `newInstance()` de la classe permet d'instancier un objet.
- Sinon, il faut utiliser la méthode `newInstance(Object...)` de `Constructor`.

```
Class compteClass = Compte.class;
Constructor<Compte> c = compteClass.getConstructor(String.class, String.class);
Compte cpt = c.newInstance("1234", "Dupont");
System.out.println(cpt);
```


Manipulation des constructeurs

Notes

La classe `Array`

- La classe `java.lang.reflect.Array` fournit un ensemble de méthodes statiques pour créer et accéder dynamiquement à des tableaux Java.

Obtenir des informations sur un tableau

- Pour déterminer si un objet est un `Array`, il suffit d'utiliser la méthode `isArray()` de sa classe.
- Pour obtenir le type des éléments d'un tableau, utiliser la méthode `getComponentType()` de sa classe.
- Pour déterminer la taille d'un tableau, utiliser la méthode statique `getLength(Object)` de `Array`.

Manipuler les données d'un tableau

- Pour lire les données stockées dans un tableau :
 - la méthode statique `get(Object, int)` permet de récupérer l'objet stocké à l'index donné, dans le tableau passé en premier argument.
 - les méthodes statiques `getXxx(Object, int)` permettent de récupérer l'objet, typé selon `Xxx`, stocké à l'index donné, dans le tableau passé en premier argument.
- Pour écrire les données dans un tableau :
 - `set(Object, int, Object)`
 - `setXxx(Object, int, Xxx)`

La classe Array

Notes

La classe Array

Création dynamique d'un tableau

- La méthode statique `newInstance(Class<?>, int)` permet de créer un tableau à une dimension
- La méthode statique `newInstance(Class<?>, int...)` permet de créer un tableau multi-dimensionnel.

```
Compte[] comptes = (Compte[]) Array.newInstance(Compte.class, 2);
Array.set(comptes, 0, new Compte("0000", "Dupont"));
Array.set(comptes, 1, new Compte("0001", "Dupond"));
System.out.println(Array.getLength(comptes));
System.out.println(Array.get(comptes, 1).getClass());

int[][] matrix = (int[][]) Array.newInstance(int.class, 10, 10);
matrix[3][5] = 2;
```

La classe Array

Notes