

Java Start

Introduction à Java et au JDK

Introduction au JDK

Origines du Projet

- Développé par Sun Microsystèmes
- Sous la direction de James Gosling
- Développement de systèmes embarqués (Machine à laver)
- Projet « Oak »,
- Évolution gérée par Sun, puis Oracle depuis 2007



Intérêt de Java

- Simplicité : pointeurs ; Garbage collector ; Outils
 - relatif par rapport à PHP
- Cross-Platform : aujourd'hui en ARM
- Open Source
- Librairies
- Outilage
- Ressources humaines disponibles
- Salaire néanmoins élevés
- La Java Virtual Machine :
 - une machine extraordinaire
 - mais gourmande en mémoire

Le JDK

- JDK = Java Development Kit
- Peu de changement de Java 5 à Java 7
- Grosses modifications pour Java 8 : les lambdas
- JRE = Java Runtime Environment
 - Permet l'execution d'une application Java
 - N'inclut pas les outils de développement
 - Plus petit que le JDK

Le JDK contient :

- Un ensemble de classes de base regroupées en packages
- Des outils : jconsole, vmstat, iostat, apt, javadoc
- Des exécutables : machine virtuelle, compilateur
- Des bibliothèques permettant de coupler du Java et du C

La compilation

- Le code source Java est compilé en bytecode
- Le bytecode permet l'execution du même code sur tous les OS
 - qui ont installé la JVM...
- Le bytecode est généralement interprété par la JVM
 - si nécessaire la JVM compile des morceaux de code
 - c'est la technique du JIT : Just in Time
- Aujourd'hui Java est aussi rapide que C
 - à condition d'écrire des structures similaires au C
 - car la création d'instances est couteuse
 - et d'avoir de la mémoire
- La JVM permet d'utiliser beaucoup de langages
 - Groovy, Scala, Jython, JRuby, Quercus (PHP) ...

Mise en Oeuvre

TP : Création de projets avec Eclipse

- Création d'un fichier
- Création d'une classe
- `main` + Ctrl + Space

```
MyApp.java
class MyApp {
    public static void main (String[] args){
        System.out.println("Hello "+args[0]+" !");
    }
}
```

- javac MyApp.java
- java MyApp World

Règles de *bonnes conduites*

- CamelCase :
 - Les classes commencent par une Majuscule
 - les variables et méthodes par une minuscule
 - Les mots sont séparés par la majuscule
- Indentation *à la Java* : Alt + Shift + F
- Codez en anglais, thanx
- Ecrivez de vrai mots

```
class mauvse_app
{
    public static void main (String[] Argmts)
    {
        System.out.println("Hello "+Argmts[0]+" !");
    }
}
```

Applications graphiques

- Démonstration de l'outil Matisse
- Limite de ces outils
- Introduction au MVC

TP : Exécuter et déboguer des applications

- Ecrire un programme minimaliste
- Lancer le programme avec l'IDE
- Débuguer le programme
- Lire le code source du JDK

Documentation : la Javadoc

```
/**  
 * This interfaces explains how a <strong>Resource</strong> can be represented.  
 * A <strong>Representation</strong> object contains the data structures AND methods to change it.  
 * A constructor should be able to accept a Resource object and built a default representation  
 * @author n.zozol  
 * @version 0.3.0  
 */  
public interface Representation {  
    /**  
     * Returns a node value, or throw a RepresentationException if it's not found  
     * @param nodeName name of the searched node  
     * @return the node value  
     * @throws RepresentationException if the nodeName is not found  
     * @author Nicolas  
     * @since 0.2.0  
     */  
    public String get(String nodeName) throws RepresentationException;
```

Documentation : la Javadoc

Avantages :

- Relie le code à une documentation
- Génère un site web facilement indexable par Google
- Permet d'intégrer la doc à l'IDE
- l'IDE peut générer des avertissement en cas de lacune de javadoc

Problèmes :

- Si le code change, il faut changer finement la javadoc
- Le HTML n'est pas un format très lisible ni productif
- `@deprecated` est changé par ... `@Deprecated`

```
@Deprecated  
public String get(String nodeName) throws RepresentationException;
```

Structures de programmation du Java

Déclaration de variables et affectation de valeurs

Exemples

```
int a = 2;  
String b = "Hello";  
String c = b+a;  
int d = b+a;
```

Les types de base

- byte, short, int, long
- float, double
- char
- boolean
- String n'est pas un type de base, mais *wrap* un tableau de char
- String est *immutable*, car le char[] est final

Declaration des types de base

```
byte b = 120;
short sh = 12300;
int i = 50456;
long l = 82300000;
char c ='c';
boolean flag = true;
```

Passage par valeur

- Nous avions dit précédemment que Java utilisait les pointeurs
- Néanmoins ce n'est pas vrai pour les types de base

```
public void useParameters(){ MyString chocolate = new
MyString("chocolate"); int number = 2; stringParameters(chocolate);
baseParameter(number);
```

```
    System.out.println(chocolate); // modified
    System.out.println(number); // not modified
}
private void stringParameters(MyString str){
    str.change("toffee");
    //str = new MyString("toffee"); quel comportement ?
}
public void baseParameter(int number){
    number += -12;
}
```

Quelques conversions à savoir

```
long    b = 5L;    // Entier long
short   a = 0xA;   // En hexadécimal
int     b = 055;   // En octal
float   f = (float)2.5;
float   f2 = 2.5F;
double  dl = 6.5e10;
char    c = 48;
int     v = 'a';
```

Les opérations de base

- Comparaison
 - `==, !=, <, <=, >, >=`
 - retournent une valeur booléenne
- Logique (seulement sur des booléens)
 - `!, &&, ||, &, |`
- Arithmétiques (sur entier et flottant)
 - `+, -, *, /, %` (modulo)
- Arithmétiques et affectation
 - `+=, -=, *=, /=; %=%`
- Unaires (sur entier et flottant)
 - `--, ++`
- Binaires bit à bit (seulement sur des entiers)
 - `~, &, |, ^, <<, >>` (décalages), `>>>` (idem, non signé)

Conditional AND

```
// safe
if (str != null && !str.isEmpty()) {
    doSomethingWith(str.charAt(0));
}

// NOT safe
if (str != null & !str.isEmpty()) {
    doSomethingWith(str.charAt(0));
}
```

Le *scope* de déclaration

Dans `MyApp.java` :

```
public class MyApp{  
    static int classAttribute = 0;  
    int instanceAttribute = 12;  
    int withNoAffectation;  
    final int noChange; // où donc l'initialiser ?  
    final static int noChangeAndAffectation = 2;  
    public static void main(String [] args){  
        MyApp myApp = new MyApp();  
        myApp.doStuff();  
        System.out.println(a);  
    }  
    public void doStuff(){  
        System.out.println("Instance attribut : "+this.instanceAttribute);  
        System.out.println("Class attribute in instance : "+this.classAttribute);  
        System.out.println("Static call of class attribute : "+MyApp.classAttribute);  
        withNoAffectation = 12;  
        int a=2;  
        System.out.println(a);  
    }  
    static String a = "where am I ?";  
}
```

Regles de déclaration

- Les variables peuvent se déclarer à beaucoup d'endroits
- Elles peuvent être static ou final
- Les variables *attributs* peuvent également être private/protected/package/public
- Le scope de visibilité fonctionne par "encapsulation"

Règles d'affectation

- Les règles sont classées en deux catégories:
 - La visibilité : private/protected/package/public
 - static/final
- static :
 - la variable appartient à la classe
 - toutes les *instances* partagent la même variable
 - et c'est donc toujours la même valeur
 - !!! mortel pour le clustering !!!
- final
 - La variable doit être initialisée tout de suite
 - Ou dans chaque constructeur

Utilisation d'expression et d'opérateurs

Les Expressions et *Statement*

- Une *Statement* résulte en une instruction executable par le processeur
- Une expression est une construction résultant en une seule valeur

```
int cadence = 0;  
anArray[0] = 100;  
System.out.println("Element 1 at index 0: " + anArray[0]);  
int result = x + y / 100 ;  
int result = (x + y) / 100 ;  
if (value1 == value2)  
    System.out.println("value1 == value2");  
x + y / 100 ; // not a statement : error
```

Création et utilisation des tableaux

Déclaration d'un tableau

```
int array[] = new int[5]; // déclaration et allocation
int[] array = new int[5]; // alternative
/* déclaration, allocation, assignation */
int array[] = new int[]{1,3,5,6,7};
int array[] = {1,3,5,6,7};
System.out.println(array[2]); // affiche ?
int tab[]; // déclaration
int size = 5;
tab = new int[size]; // allocation ; impossible en C
int[][] multiTab = {{1,2,3,4,5,6},
                     {1,2,3,4},
                     {1,2,3,4,5,6,7,8,9}};
```

itération d'un tableau

```
for (int i = 0; i < array.length ; i++) { ... }  
//à utiliser dans le cadre d'un parcours partiel de tableau  
while (cursor < array.length){  
    ....;  
    if (foundSomething) break;  
    cursor++;  
}
```

- do ... while est légal ; just don't do it

TP1 : Arrays

- `public static int getLastValue(int[] array)`
- `public static int[] increaseAllValues(int[] array)`
- `public static int[] sort(int[] array)`

Déclaration et appel de méthode

Méthode d'instance, méthode de classe, constructeur

```
public class Comment extends UltraPowerfulModel {  
    String user;  
    String content;  
    Date date;  
    public Comment(String user, String content) {  
        this.user = user;  
        this.content = content;  
        this.date = new Date();  
    }  
    public String getContent(){  
        return this.content;  
    }  
    public static int count(){  
        return query("Count(*) FROM Comment").asInt();  
    }  
}
```

Méthode d'instance, méthode de classe, constructeur

- Un constructeur permet de créer une nouvelle instance

```
Comment myCom = new Comment("Joe", "Something cool");
```

- Une méthode d'instance s'execute... sur une instance

```
String templateContent = myCom.getContent();
```

- Une méthode statique est appelé sur la classe

```
String templateCount = String.valueOf(Comment.count());
```

Varargs

- Les varargs permettent d'appeler un nombre variable d'objets
- Le varargs est toujours le dernier argument
- Il n'y a qu'un seul varargs
- Le varargs est un array avec du *sucre syntaxique*

```
public void useVarargs(String stuff, int... otherStuff){  
}  
myObject.useVarargs("direct", 2,3,5);  
myObject.useVarargs("direct", 2,3,5,9);  
myObject.useVarargs("none");  
myObject.useVarargs("direct", new int[]{2,3,5});
```

Gestion d'exceptions

Déclaration des exceptions

- Une exception est une erreur qui se produit dans un programme
- Les Exceptions sont récupérables par le programme grâce au bloc try/catch
- Une Exception *devrait* rester... exceptionnelle

```
try {  
    Q q1 = new Q(a,b);  
} catch (DivisionByZeroException ex) {  
    ex.printStackTrace();  
}
```

Checked et Runtime Exceptions

- On distingue les exceptions au *Runtime* des *checked exceptions*
 - Runtime exception : erreur imprévue, en général un bug
 - Checked Exception : condition vérifiées et invalidées
- Tout bout de code susceptible de produire une checked exception doit soit :
 - capturer l'exception dans un bloc try/catch
 - relancer l'exception
- Si l'API est mal designée/utilisée, on risque une cascade de relance

Exemples

```
public Q(int numerator, int denominator) throws DivisionByZeroException {  
    this.numerator = numerator;  
    this.denominator = denominator;  
    if (this.denominator == 0){  
        throw new DivisionByZeroException();  
    }  
}  
... then ....  
public Q createQ(int a, int b){  
    try {  
        return new Q(a,b);  
    } catch (DivisionByZeroException ex) {  
        return null;  
    }  
}  
public Q createRethrowQ(int a, int b) throws DivisionByZeroException{  
    return new Q(a,b); //rethrow obligé  
}
```

Runtime Exception

DivideByZero Exception :

```
public float getQValue(int a, int b){  
    return (a/b); // Exception risk  
}
```

NullPointerException :

```
public float getQValue(int a, int b){  
    Q q1 = null;  
    if (1==2) q1=new Q(a,b);  
    return q1.getRealValue();  
}
```

Finally

- Finally est utilisé pour s'assurer de l'état d'un objet
- Typiquement, on utilise finally pour :
 - fermer une connection à la base de donnée
 - ouverture d'un fichier (libération pour l'OS)
 - libérer un lock dans le multithreading

Exemple

```
File f = new File("calculous");
FileOutputStream fos = null;
try {
    fos = new FileOutputStream(f);
    fos.write(12);
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
} catch (IOException ex) {
    ex.printStackTrace();
} finally{
    if (fos != null){
        try {
            fos.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

try/finally

- Il est possible d'utilise finally sans le catch
- Cela permet de fermer une connexion même en cas de RuntimeException

```
Connection myConnection = new Connection();
try{
    myConnection.useIntensively();
}finally{
    myConnexion.close();
}
```

Exemple dans un Thread

```
Thread writerThread = new Thread() {  
    @Override  
    public void run() {  
        lock.lock();  
        try {  
            writer.write(datas);  
        } finally {  
            lock.unlock();  
        }  
    }  
};
```

Finally, Return et Error

- `finally` a toujours lieu avant le `return`
- Une `Error` est lancée quand la JVM est imprédictible

Multiple catch/throws

- Une méthode peut lancer plusieurs exceptions

```
public void fileRead() throws EOFException, FileNotFoundException
```

- Il est alors nécessaire de *catcher* chaque exception

```
try{  
    fileRead()  
}  
catch (EOFException ex){....}  
catch (FileNotFoundException){...}
```

Lire et écrire dans des fichiers

Les flux

- Java utilise un concept de haut niveau appelé flux
- Ce flux peut contenir des octets ou des caractères
- Il existe des flux d'entrée ou de sortie

Flux d'octets

Flux de caractères

Flux d'entrée InputStream Reader Flux de sortie OutputStream Writer

Les flux de conversion

- Il existe des ponts permettant de transformer un flux vers un autre
 - OutputStreamWriter : flux de caractères -> flux d'octet
 - InputStreamReader : flux d'octet -> flux de caractères
- Les PipedInputStream et PipedOutputStream peuvent aussi jouer ce rôle

Les flux liées aux fichier

- FileInputStream : lire des octets dans un fichier
- FileOutputStream : ecrire des octets dans un fichier
- FileReader : Lit des caractères
- FileWriter : Ecrit des caractères
- FileReader et FileWriter *embarquent* leur flux de conversion

Lire un fichier

Premier exemple

```
public void readJBossFile() throws IOException {
    FileInputStream reader = new FileInputStream(file);
    String line = null;
    StringBuilder builder = new StringBuilder();
    int currentChar;
    while ((currentChar = reader.read()) != -1) {
        builder.append((char)currentChar);
    }
    System.out.println(builder.toString());
    reader.close();
}
```

Utilisation d'un Buffer

```
public void writeInFile() throws FileNotFoundException, IOException {
    String content = "<?xml version=\"1.0\"?>\n<root>Some nice xml</root>";
    File file = new File(xmlFile);
    if (!file.exists()) {file.createNewFile();}
    //wow !!!
    BufferedOutputStream output = new BufferedOutputStream(
        new FileOutputStream(file));
    byte[] contentInBytes = content.getBytes("UTF-8");
    output.write(contentInBytes);
    output.flush();
    output.close();
}
```

Quelques détails

```
BufferedOutputStream output = new BufferedOutputStream(  
    new FileOutputStream(new File("file.xml")));  
...  
byte[] contentInBytes = content.getBytes("UTF-8");  
...  
output.flush();  
output.close();
```

TP

- Ecrire une méthode pour lire un fichier avec un buffer
- Ecrire une méthode pour copier un fichier, utilisant un buffer
- Ecrire une méthode pour déplacer un fichier

Les problèmes de l'API io

- C'est lourd....
- Certaines méthodes lèvent une exception sur un problème, d'autres renvoient false
- La méthode rename() n'a pas le même comportement sur toutes les plateformes
- Il n'y a pas de réel support pour les liens symboliques (symbolic links)
- Les métadonnées du fichier sont peu supportées
- Certaines fonctionnalités comme copy() ou move() basiques sont absentes
- Les performances sont parfois faibles
- L'API est dite bloquante : le thread est en pause quand un flux attend un octet/char

Les API nio et nio2 résolvent ces problèmes

Exemple d'utilisation de nio2

```
String readFile(String path, Charset encoding) throws IOException
{
    byte[] encoded = Files.readAllBytes(Paths.get(path));
    return encoding.decode(ByteBuffer.wrap(encoded)).toString();
}
```

- On reconnaît l'API nio avec **Paths** et **Files**

La Programmation Objet en Java

Les règles d'une nouvelle classe

- Une classe représente un concept
- Toute classe Java est une sous-classe de la classe Object.
- Il n'existe pas d'héritage multiple
 - mais on pourra utiliser plusieurs interfaces
- Une classe fille hérite des attributs et des méthodes
- Une classe fille peut modifier l'implémentation des méthodes héritées
- Une classe fille peut ajouter un attribut du même nom : BAD IDEA

Exemple

```
public class User /* extends Object */{  
    String name;  
    String email;  
    int age;  
    Address adress;  
    @Override //redéfinit Object.toString()  
    public String toString() {  
        return this.name+" is "+age+" years old";  
    }  
}
```

Exemple d'Héritage

```
public class Admin extends User{  
    String directory;  
    @Override  
    public String toString() {  
        return this.email+">>> "+this.directory;  
    }  
}
```

Les Constructeurs

- Un constructeur permet d'instancier un nouvel objet grâce à `new X()`
- Une classe fille appelle **toujours** le constructeur parent avec `super()`
- Au bout, on appelle le constructeur par défaut de Object

Le constructeur par défaut

```
public class User {  
    public User() {  
        //super(); //appelle Object()  
    } }
```

L'appel du parent se fait en cascade

```
public class Admin extends User {  
    public Admin() {  
        //super() //qui appelle User() qui appelle...  
    } }
```

L'appel du constructeur par défaut est facultatif

```
public class User {  
    //public User() {}  
}
```

Objectifs d'un constructeur

```
public class User {  
    String name;  
    String email;  
    int age;  
    Address adress;  
    public User(String name, String email) {  
        //super() //on appelle toujours Object()  
        this.name = name;  
        this.email = email;  
        //adress == null ; age =0  
    }  
}
```

- `new User("John", "john.doe@robusta.io")` permet de créer le User souhaité
- Il n'y a alors plus de constructeur par défaut
- Et donc le code Admin a une erreur

Constructeur multiple

```
public class User {  
    String name;  
    String email;  
    int age;  
    Address adress;  
    public User(String name, String email) {  
        this.name = name;  
        this.email = email;  
    }  
    public User(String name, String email, int age, Address adress) {  
        this.name = name;  
        this.email = email;  
        this.age = age;  
        this.adress = adress;  
    }  
}
```

Constructeurs multiples (mieux)

```
public class User {  
    String name;  
    String email;  
    int age;  
    Address adress;  
    public User(String name, String email) {  
        if (email==null||email.isEmpty()) {  
            throw new IllegalArgumentException("invalid");  
        }  
        this.name = name;  
        this.email = email;  
    }  
    public User(String name, String email, int age, Address adress) {  
        this(name, email); //<-- c'est mieux !  
        this.age = age;  
        this.adress = adress;  
    }  
}
```

- Créer des constructeurs adéquats pour `Comment` et `ImageComment`

IS-A et HAS-A

```
public class Comment {  
    String content;  
    User user; //HAS-A  
    public String html(){  
        return "<div>" + this.getContent() + "</div>";  
    }  
}  
//IS-A  
public class ImageComment extends Comment{  
    String file;  
    public void deleteImage(){  
        new File(file).delete();  
    }  
}
```

- Implémenter `html()` pour `ImageComment`
- Soit le code :

```
//the ImageComment IS-A Comment
Comment c = new ImageComment();
System.out.println(c.html()); // Quel résultat ?
c.deleteImage(); // possible ?
```

Overriding et Overloading

- Overriding :
 - *redéfinition*
 - Une classe fille réimplémente une méthode du père
- Overloading :
 - *surcharge*
 - Une classe contient plusieurs méthodes du même nom
 - mais avec des paramètres différents
 - on parle de *signature* d'une méthode
 - Nous avons déjà vu l'overloading du constructeur !

Exemple d' Overloading

Noter l'utilisation de `super.html()` :

```
public class ImageComment extends Comment {  
    @Override  
    public String html() {  
        return "<div>" + this.file + "</div>";  
    }  
    public String html(boolean total) {  
        if (total) {  
            return super.html() + "<div>" + this.file + "</div>";  
        } else {  
            return html();  
        }  
    }  
}
```

Problème de l'overriding

```
public class Comment {  
    String content;  
    public Comment(String content) {  
        this.content = content;  
    }  
    public String html() {  
        return "<span>" + this.content + "</span>";  
    }  
    public String html(User user) {  
        return html() + "<span>" + user.name + "</span>";  
    }  
    public String html(Admin admin) {  
        return html() + "<span>" + admin.directory + "</span>";  
    }  
}  
Admin joe = new Admin("/jo", "jo", "jo@robusta.io");  
User jack = new Admin("/jack", "jack", "jack@robusta.io");  
Comment c = new Comment("hello ");  
c.html(joe) => "" hello /jo"  
c.html(jack) => ?
```

Conclusion

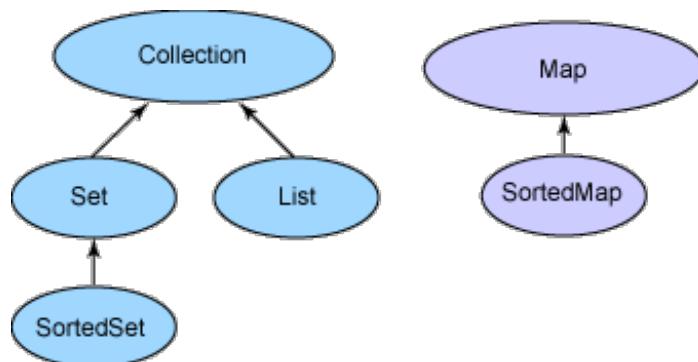
- Le Polyformphisme signifie prendre plusieurs formes
 - Object, User, Admin
 - Object, Comment, ImageComment
- Le choix d'une méthode *polymorphe* sur un objet se fait à l'exécution
- Le choix d'une méthode *surchargée* se fait à la compilation

Les Collections

Les différentes branches de l'API Collection

Interfaces

- Map : collections indexées par des clés (Cf. Entrées d'un dictionnaire)
- Collection



Classes

- Abstraites : AbstractCollection, AbstractMap, AbstractList, ...
- Concrètes : ArrayList, HashSet, TreeSet, HashMap, ...

Interface Collection

Un objet qui est un groupe d'objets Principales méthodes:

- boolean add(Object obj)
- boolean contains(Object obj)
- boolean containsAll(Collection collection)
- Iterator iterator()
- int size()
- Object[] toArray()
- Object[] toArray(Object[] tableau)

Interface Set

- Un groupe d'objets n'acceptant pas 2 objets égaux (au sens de `equals`).
- Implémentations :
 - `HashSet`
 - `TreeSet`
- Les Set sont assez rarement utilisés

Interface List

- Un groupe d'objets repérés par des numéros (en débutant à l'indice 0)
- Classes implémentées : `ArrayList` , `LinkedList`

Interface Map

- Des couples clé-valeur.
- Une clé repère une unique valeur
- Plusieurs clés peuvent pointer vers la même valeur
- Implémentations : HashMap et TreeMap

Les Listes

La Classe Vector et Hastable

- Obsolète
- Dites Thread-safe
- Ne plus utilisé

Classe ArrayList

- Un tableau contenant un nombre quelconque d'instances de la classe Object .
- Emplacements sont repérés par des valeurs entières .
- Les méthodes ne sont pas synchronisées (Vector).

Méthodes de ArrayList

- boolean add(Object obj)
- void add(int indice, Object obj)
- boolean contains(Object obj)
- Object get(int indice) => casting l'élément.
- int indexOf(Object obj)
- Iterator iterator()
- void remove(int indice)
- void set(int indice, Object obj)
- int size()

Exemple d'ArrayList

```
ArrayList liste = new ArrayList();
Comment c1 = new Comment("Toto");
liste.add(c1);
// création d'autres instances de commentaires
for (int i=0; i<liste.size();i++){
    System.out.println(((Comment)liste.get(cpt)).getTitle());
}
```

Exemple d'ArrayList avec les génériques

```
ArrayList<Comment> liste = new ArrayList<Comment>();  
liste.add(new Comment("Toto"));  
// création d'autres instances de commentaires  
for (int i=0; i<liste.size();i++){  
    System.out.println(liste.get(cpt).getTitle());  
}
```

Les Sets

HashSet

- Implémentation simple
- Utilise une table de Hash
- Les éléments sont uniques
- Unicité au sens *equals*

equals et hashCode

La classe `Object` contient les méthodes :

- `equals()` : égalité logique
 - par défaut égalité de référence
- `hashCode()` : déterminant d'un objet
 - deux objets égaux => même hashCode
 - deux hashCode égaux => vérifier via `equals()`

`hashCode()` doit toujours être très rapide.

TreeSet

- Set trié
- Les objets du Set doivent implémenter Comparable
- ou le TreeSet utilise un Comparator

Les Maps

Les Méthodes des Map

- boolean containsKey(Object clé)
- boolean containsValue(Object valeur)
- Set entrySet()
- Object get(Object clé) // null si objet n'existe pas.
- boolean isEmpty()
- Set keySet()
- int size()
- Collection values()

Classe HashMap

- Une HashMap est un tableau associatif
 - Certains langages utilisent **Dictionary**
- On regroupe des valeurs par clés
- Permet une recherche efficace
- La méthode hashCode() est exploitée pour répartir les clés dans la table de hachage
- HashMap a succédé à Hashtable (deprecated)

Exemple de HashMap

```
Map<String, Subject> sb = new HashMap<String, Subject>();
sb.put("s101",new Subject("Toto"));
sb.put("s102",new Subject("Titi"));
Subject p = (Subject) sb.get("s101");
System.out.println(s.getTitle());
Collection elements = sb.values();
Iterator<Subject> it = elements.iterator();
while(it.hasNext()) {
    System.out.println(it.next().getTitle());
}
```

Les outils des Listes

Trier une liste

```
import java.util.*;
public class Coll1 {
    public static void main(String args[]) {
        List<Integer> liste = new ArrayList<Integer>();
        liste.add( new Integer(9));
        liste.add(new Integer(12));
        liste.add(new Integer(5));
        Collections.sort(liste);
        Iterator<Integer> it = liste.iterator();
        while (it.hasNext()) {
            System.out.println((it.next()).toString());
        }
    }
}
```

Recherche dans une liste

```
import java.util.*;
public class ListeSubjectTest {
    public static void main(String[] args) {
        List<Subject> liste = new ArrayList<Subject>();
        Subject s1 = new Subject("Java","Developpement en java");
        Subject s2 = new Subject("python ","python dev");
        Subject s3 = new Subject("Xcode","apple dev");
        liste.add(s1);
        liste.add(s2);
        liste.add(s3);
        Collections.sort(liste, new ComparatorSubject());
        System.out.println("Recherche ="+
        Collections.binarySearch(liste,s1,new
        ComparatorSubject()));
    }
}
```

Interface Comparator

Elle propose une méthode :

- int compare(Object o1, Object o2)

Exemple d'un Comparator

```
import java.util.Comparator;
public class ComparatorSubject implements Comparator<Subject> {
    public int compare(Subject o1, Subject o2) {
        String titre1 = o1.getTitle();
        String titre2 = o2.getTitle();
        return titre1.compareTo(titre2);
    }
}
```

Utilisation d'un Comparator

```
import java.util.*  
public class ListeSubjectTest {  
    public static void main(String[] args) {  
        List liste = new ArrayList();  
        Subject s1 = new Subject("Java", "java dev");  
        Subject s2 = new Subject("Python", "python dev");  
        Subject s3 = new Subject("Xcode", "apple dev");  
        liste.add(s1);  
        liste.add(s2);  
        liste.add(s3);  
        Collections.sort(liste, new ComparatorSubject());  
    }  
}
```

Exemples d'utilisation d'HashMap

```
Subject s1 = new Subject("Java", "Developpement en java") ;  
Subject s2 = new Subject("python", "Developpement en python") ;  
Subject s3 = new Subject("Java", "Developpement android") ;  
Subject s4 = new Subject("Xcode", "Developpement iOS") ;  
Map<String, Subject> map = new HashMap<String, Subject>() ;  
map.put(s1.getTitle(), s1) ;  
map.put(s2.getTitle(), s2) ;  
map.put(s3.getTitle(), s3) ;  
map.put(s4.getTitle(), s4) ;  
// première interrogation de la table  
System.out.println("[Java] -> " + map.get("Java")) ;  
System.out.println("Eléments : " + map.size()) ;  
// parcours de la table par ses entrées  
for (Map.Entry<String, Subject> entry : map.entrySet()) {  
    System.out.println("[" + entry.getKey() + "] -> " + entry.getValue()) ;  
}
```

Les Itérators et la nouvelle boucle For

Interface Iterator

- Une interface permettant d'énumérer les éléments contenus dans une collection.
- Toutes les collections proposent une méthode itérатор renvoyant un itérateur.
- Parcours dans un sens uniquement. Cf. ListIterator pour un parcours bidirectionnel

Méthodes d'Iterator

- boolean hasNext()
- object next()

Exemple d'Iterator

```
ArrayList<Comment> liste = new ArrayList<Comment>();  
liste.add(new Comment("Toto"));  
// création d'autres instances de Commentaire  
Iterator<Comment> it = liste.iterator();  
while(it.hasNext()) {  
    System.out.println(it.next().getTitle());  
}
```

foreach

- Une autre alternative pour l'itération sur des structures.
- Exemple : `for(String s : myList)`, avec myList

```
ArrayList<Comment> liste = new ArrayList<Comment>();  
liste.add(new Comment("Toto"));  
// création d'autres instances de Commentaire  
for(Comment c : liste) {  
    System.out.println(c.getTitle());  
}
```

- On parcours la liste de la même façon que `myList.iterator()`
- Il faut donc que myList implémente Iterator, ou soit un tableau

For : Pour les HashMap

Le principe est d'itérer sur les **Entries** :

```
HashMap<String, Integer> map = new HashMap<String, Integer>();  
//fill map  
for (Map.Entry<String, Integer> entry : map.entrySet()) {  
    System.out.println(entry.getKey() + " -> " + entry.getValue());  
}
```

Conclusion

Le nouveau for est :

- Plus court
- Plus lisible
- Plus efficace

Les Générics

Principe

Type Générique

Les Types génériques permettent de paramétrer une classe

- La lecture devient plus simple
- Le code devient plus robuste
- Le paramètre peut utiliser une **wildcard**
 - T extends Y
 - T super Z
- Les paramètres sont très souvent utilisés dans les **Collections**

Type Générique : Toutefois

Il faut savoir que

- Les paramètres sont retirés dans le **bytecode**
- La classe paramétrée est interprétée comme une nouvelle classe
- Certains cas liés à l'inférence sont vraiment compliqués

Génériques sur les Collections

- Les Génériques permettent d'imposer le contenu d'une Collection
- Elles deviennent très simple à utiliser
- On évite une multitude de casts peu robustes

Code

```
ArrayList<Number> numbers = new ArrayList<Number>();  
numbers.add(2.25f);  
numbers.add(-20000);  
numbers.add("28"); // Compilation fails  
Number n = numbers.get(1);  
int myInt = (Integer) numbers.get(1);
```

Generics hors des Collections

- Le principe est le même
- On définit souvent le type d'un attribut de la classe

```
public class Travailleur {
```

```
    public T getSalaire();  
}  
Travailleur employe = new Travailleur<Integer>();  
Travailleur boss = new Travailleur<Long>();
```

Plus de précision avec Extends

```
public class Travailleur <T extends Number> extends Humain{  
    public T getSalaire();  
}  
Travailleur employe = new Travailleur<Integer>();  
Travailleur boss = new Travailleur<Long>();
```

Type Générique sur une méthode

On peut définir un paramètre uniquement sur une méthode

```
public <FruitDeSonTravail> int getSalaire(List<FruitDeSonTravail> fruits){  
    Set<FruitDeSonTravail> uniques = new HashSet<FruitDeSonTravail>(fruits);  
    return uniques.size()*350;//calcul salaire  
}
```

Exemple de JPA

L'API décrit :

```
Interface EntityManager :  
<T> T find(Class<T> entityClass, Object primaryKey)
```

Ce qui s'utilise ainsi :

```
EntityManager em = getEntityManagerFactory().createEntityManager();  
Travailleur t = em.<Travailleur>find(Travailleur.class, 12L);
```

ou en raccourci :

```
Travailleur t = em.find(Travailleur.class, 12L);
```

Type Générique : l'inférence

```
Integer[] array = new Integer[]{1,2,3,4,5};  
List<Integer> list = new ArrayList<Integer>();  
list.addAll(Arrays.asList(array));  
//array a un type connu pour le bytecode  
Integer[] result = list.toArray(new Integer[]{});  
//on ne peut pas faire list.toArray();  
System.out.println(list);
```

Type Générique : covariance

```
List<Integer> myInts = newArrayList<Integer>();  
myInts.add(1);  
myInts.add(2);  
List<Number> myNums = myInts; //compiler error
```

Conclusion

- Les Génériques sont **extrêmement** utiles dans les Collections
- On peu cependant vite s'emmeler dans les autres situations
- Ecrire une API nécessite toutefois une profonde analyse des Génériques
- On les retrouve aussi dans les API de persistence

Les Annotations

Exemple

```
@Override  
public String toString{  
    return this.name;  
}
```

Les Annotations : Définition

- Une annotation n'est pas une *instruction*
- Une annotation est une *indication* qui peut être utilisée par :
 - le développeur ou l'IDE(relecture de code, `@deprecated`)
 - un outil externe : génération de code, de fichiers ...
 - le compilateur : `@Deprecated`, `@SuppressWarnings` //warning au compile
 - Au démarrage d'une application : `@WebService`, `@Path`
 - ou à tout moment de l'exécution du code (API reflection)

Les Annotations réservées du JDK

Parmi les annotations réservées par le JDK, on distingue les annotations standard des méta-annotations

Les Annotations standard :

- `@Deprecated` : du coup, il n'y a pas de commentaire !
- `@Override` : ! $\text{JDK5} \neq \text{JDK6}$
- `@SuppressWarnings` : indique au compilateur (et à l'IDE) de ne pas avertir

Les Méta-Annotations : elles s'appliquent à la définition d'une annotation

- `@Retention` : qui peut l'utiliser ?
- `@Target` : sur quels éléments s'applique t-elle ?
- `@Inherit` : l'annotation s'hérite

- `@Documented` : l'annotation se retrouve dans la Javadoc (fait partie de l'API)

Les Annotations : Retention

- `RetentionPolicy.SOURCE` : reste au code Source
- `RetentionPolicy.CLASS` : Rétention par défaut
- `RetentionPolicy.RUNTIME` : Lisible par reflection

Les Annotations : Retention.Source

- Un programme extérieur peut alors utiliser les annotations
- On peut alors tout imaginer à condition d'avoir les fichiers source

Voir exemple en groovy pour récupérer les annotations

Les Annotations : Retention.Class

- Les jar/war ne contiennent en général pas les .java, mais les .class
- Permet de parser les fichier compilés sans les executer
- On peut faire autant qu'avec la Retention Source, mais c'est beaucoup moins lisible
- Exemple : Jersey

```
<init-param>
  <param-name>com.sun.jersey.config.property.packages</param-name>
  <param-value>de.vogella.jersey.first</param-value>
</init-param>
```

Les Annotations : Retention.Runtime

- La Retention au Runtime permet d'appliquer des traitement dynamiquement
- Selon le flow, le programme agira différemment grâce aux annotations
- On utilise alors l'API de reflection
- L'annotation seule ne suffit pas :
 - Il faut que l'application embarque un moteur pour gérer ces annotations

Exemple : JPA, JAX-RS...

Les Annotations : Target

- Target permet de définir l'objet susceptible d'accueillir l'annotation
- Par défaut, l'annotation est utilisable partout
- Plusieurs éléments peuvent devenir une cible

```
@Target( {ElementType.CONSTRUCTOR, ElementType.METHOD} )
```

Les Annotations : Valeurs d'une Annotation

- On peut attribuer une ou plusieurs valeurs à une Annotation
- Ces valeurs sont définies dans la pseudo-interface de l'annotation
- Cette valeur pour être analysée par l'outil externe ou au Runtime

Les Annotations : Exemple de valeur

```
public @interface Slow {  
    public Speed value() default Speed.SLOW;  
    public float lostMoney() default -2.2;  
}  
public class SlowGuy implements Movable {  
    @Slow(value = NEGATIVE, lostMoney = -1200) //multiple values  
    public void move() {  
        System.out.println("SlowGuy has moved slowly");  
    }  
}  
System.out.println(annotation.lostMoney()); //-> -1200
```

Les Annotations : value()

La clé **value** de l'annotation est facultative

```
public class SlowGuy {  
    @Slow(NEGATIVE) // lié à la méthode value  
    public void move() {  
        System.out.println("SlowGuy has moved slowly");  
    }  
    System.out.println(annotation.value()); //-> NEGATIVE
```

Les Annotations : value()

... à condition d'être seule

```
public class SlowGuy {  
    @Slow(NEGATIVE, lostMoney = 1200)//erreur compile  
    public void move() {  
        System.out.println("SlowGuy has moved slowly");  
    }  
}
```

Les Annotations : Type tableau

- Les annotations peuvent être typées par une primitive ou une enum
- Ou un tableau de primitives ou enums
 - exemple : `@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})`

Les Annotations : Type tableau

```
public @interface Slow {  
    public Speed value() default Speed.SLOW;  
    public Speed[] values();  
}  
public class SlowGuy {  
    @Slow(value = NEGATIVE, values = {NEGATIVE, SLOW})  
    public void move() {  
        ...  
    }  
}
```

Les Annotations : Type tableau

Toutefois les tableaux ne peuvent renvoyer un tableau par défaut ...autre qu'un tableau vide

```
public @interface Slow {  
    public Speed value() default Speed.SLOW;  
    public Speed[] values() default {} // Renvoit un tableau vide  
}  
public class SlowGuy {  
    @Slow // all default values  
    public void move() {  
        ...  
    }  
}  
System.out.println( annotation.values().length); //-> 0
```

Conclusion

- Bien designée, les annotations peuvent être impressionnantes
- Analyser les Annotations au Runtime a un coût
- Sans expérience, on risque de compliquer plutôt que simplifier

Les Datas en Java

Trois catégories

- Pur JDBC : connection binaire vers SQL
- Framework Spring : Interface plus sympathique
- JPA / Hibernate : ORM plus sophistiqué

JDBC

- Définition du Driver
- Ecriture de la requête
- Envoi de la requête
- Parser la réponse

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");
ResultSet résultats = null;
String requete = "SELECT * FROM client";
try {
    Statement stmt = con.createStatement();
    res = stmt.executeQuery(requete);
    user.name = res.getString("USER_NAME");
    user.id = res.getInt("USER_ID");
    //...
} catch (SQLException e) {
    //working on exception
}
```

Spring

- Spring rajoute quelques "simplifications"
- Chacun ses goûts....

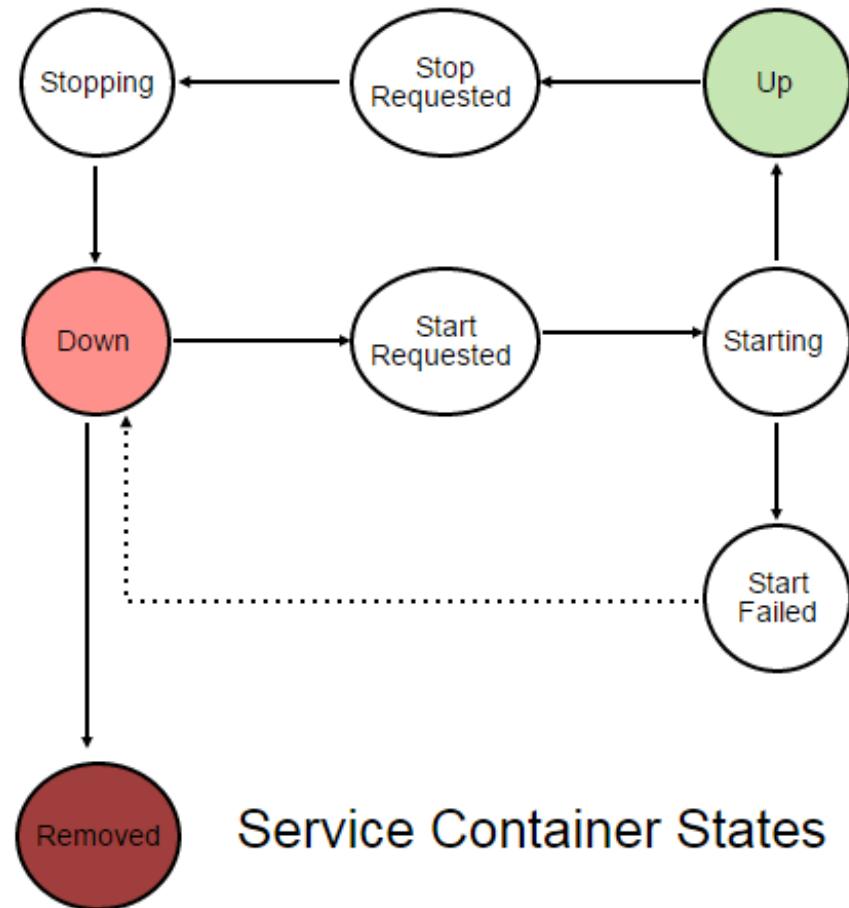
Code Spring avec un RowMapper

```
RowMapper<IUser> mapper = new RowMapper<IUser>() {
    @Override
    public IUser mapRow(ResultSet rs, int row) {
        User user = new User();
        user.setId(rs.getLong("id"));
        user.setEmail(rs.getString("email"));
        user.setName(rs.getString("name"));
        return user;
    }
};
@Override
public IUser findUserById(long id) {
    return this.jdbcTemplate.queryForObject(
        "SELECT * FROM User u WHERE id=?",
        new Object[]{1L}, // query args
        mapper);
}
```

Code Spring avec un BeanPropertyRowMapper

```
@Override
public IUser findUserById(long id) {
    IUser queryForObject = this.jdbcTemplate.queryForObject(
        "SELECT * FROM User u WHERE id=?",
        new Object[]{1L},
        new BeanPropertyRowMapper<User>(User.class)
    );
    return queryForObject;
}
```

JPA / Hibernate



```
TypedQuery<User> q2 =  
    em.createQuery("SELECT u FROM User u WHERE u.money > :money", User.class);  
    q1.setParameter("money", 12);  
    List<User> users = q1.getResultList();
```

Avantages de JPA

- Plus simple
- Plus clair
- Plus facile d'améliorer les performances

Défauts

- Très facile de **détruire** les performances
- Necessite un collègue TRES expérimenté