



CSC4102: Programmation orientée objet en Java et construction de logiciel

Denis Conan, avec Christian Bac

Janvier 2022



Sommaire

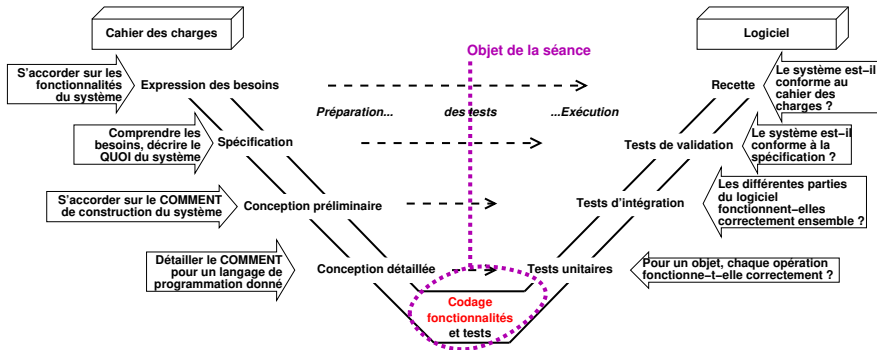
1. Motivations et objectifs de la séance
2. Retours sur l'héritage
3. Classe paramétrée (par un type)
4. Énumération, annotation, argument variadique
5. Visibilité, égalité et hachage
6. Bibliothèque, avec l'exemple des collections
7. Construction de logiciel avec Maven
8. Mise en pratique en TP (2h) + HP (3h)

1 Motivations et objectifs de la séance

- 1.1 Contexte — Où en sommes-nous ?
- 1.2 Qui demande à outiller la construction du logiciel ?
- 1.3 Quels sont les objectifs de la séance ?

1.1 Contexte — Où en sommes-nous ?

- Mise en place du processus de construction du logiciel
- Première ébauche des classes (sans les exceptions et sans les tests)
- À venir dans la séance 6
 - Gestion des cas d'erreur avec les exceptions
 - Programmation des tests unitaires et de validation

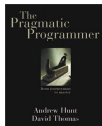


1.2 Qui demande à outiller la construction du logiciel ?

Un éditeur de texte comme emacs ou un environnement de développement comme Eclipse ne suffisent-ils pas ?

Section 42 de « *The Pragmatic Programmer : From journeyman to master* »

[Hunt and Thomas, 2000]



- *We were once at a client where all the developers were using the same IDE. Their system administrator gave each developer a set of instructions on installing add-on packages to the IDE.*
 - *These instructions filled many pages—pages full of click here, scroll there, drag this, double-click that, and do it again [...]*
- *IDEs have their advantages, but with IDEs alone it can be hard to achieve the level of automation that we are looking for.*
 - *We want to check out, build, test, and ship with a single command.*
- Dans ce module, nous utilisons MAVEN + GITLAB CI
 - Pour construire notre application et la tester à chaque commande git push

1.3 Quels sont les objectifs de la séance ?

- En chemin vers un premier noyau fonctionnel de l'application
 - Programmation des classes
 - Attributs et opérations des premiers cas d'utilisation
 - Uniquement ceux qui sont nécessaires pour la réalisation des cas d'utilisation du Sprint 1
 - Ce cours : programmation sans prise en compte des cas d'erreurs
- Outillage de la construction du logiciel
 - Étude de MAVEN : squelette de configuration fourni au démarrage du dépôt
 - Nous faisons le « pari » que nous gagnerons du temps ensuite
 - « *Let the computer do the repetitions, the mundane—it will do a better job of it than we would. We've got more important and more difficult things to do.* » [Hunt and Thomas, 2000]

2 Retours sur l'héritage

- 2.1 Transtypage
- 2.2 Redéfinition (en anglais, *overriding*)
- 2.3 Classes abstraites
- 2.4 Interfaces

2.1 Transtypage

- **Vers le haut (en anglais, *upcast*) = implicite** : une référence d'une classe T peut référencer toute instance de la classe T ou de ses classes enfants
- **Vers le bas (en anglais, *downcast*) = explicite** : une référence d'une classe S dérivée d'une classe T peut recevoir une référence de la classe T si et seulement si :
 - **L'objet référencé est autorisé** : bien du type S ou d'un de ses types dérivés
 - **Le changement de type est explicite vers le nouveau type**

Classe `seance5.mediathequesimplifiee.transtypage.ExempleTranstypage`

```
1 Document doc = null;
2 Audio audio=new Audio("3","Rock_bottom","Rober_Wyatt","73","Progressif");
3 doc = audio; // transtypage vers le haut, upcast
4 System.out.println("doc_est_un_Document:" + (doc instanceof Document)
5 + " ;_doc_est_un_Audio:" + (doc instanceof Audio));
6 audio = (Audio) doc; // transtypage vers le bas, downcast
7 System.out.println("audio_est_un_Document:"+(audio instanceof Document)
8 + " ;_audio_est_un_Audio:" + (audio instanceof Audio));
```

```
doc est un Document : true; doc est un Audio : true
audio est un Document : true; audio est un Audio : true
```


2.2 Redéfinition (en anglais, *overriding*)

- Redéfinition possible des méthodes publiques ou protégées héritées
 - C'est le principe de substitution de Liskov&Wing (polymorphisme d'inclusion)
- Mettre l'annotation `@Override` = vérification du prototype par le compilateur
- Quelle méthode est appelée? \implies liaison dynamique
 - Redéfinition la plus proche dans l'arbre d'héritage de la classe actuelle

Classe `seance5.mediathequesimplifiee.liaisonTardive.ExempleLiaisonTardive`

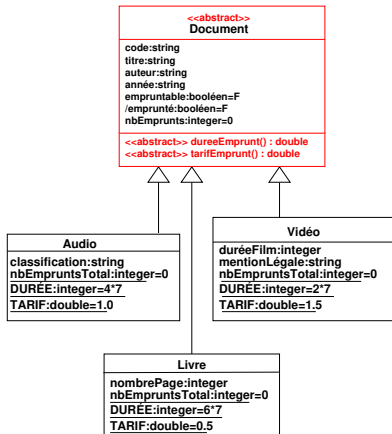
```
1 Document doc = new Document("C007", "Le seigneur des anneaux", "Tolkien",
    "1950");
2 Audio audio = new Audio("C003", "Rock bottom", "Rober Wyatt", "1973", "
    Progressif");
3 System.out.println(doc.toString());
4 System.out.println(audio.toString());
5 doc=audio; // transtypage vers le haut, type formel/actuel=Document/Audio
6 System.out.println(doc.toString()); // appel de toString de Audio
```

`Document` [code=C007, titre=Le seigneur des anneaux, auteur=Tolkien, annee=1950, empruntable=false, emprunte=false, nbEmprunts=0]

`Audio` [classification=Progressif, toString()=Document [code=C003, titre=Rock bottom, auteur=Rober Wyatt, annee=1973, empruntable=false, emprunte=false, nbEmprunts=0]]

`Audio` [classification=Progressif, toString()=Document [code=C003, titre=Rock bottom, auteur=Rober Wyatt, annee=1973, empruntable=false, emprunte=false, nbEmprunts=0]]

2.3 Classes abstraites



- Classe abstraite Document qui ne peut pas être instanciée
 - Raison 1 : on ne veut pas d'instance
 - Raison 2 : des méthodes ne sont pas définies
- Spécialisation nécessaire jusqu'à obtenir des classes instanciables
 - P.ex. la classe Audio

⇒ La classe abstraite impose un comportement commun à toutes les classes enfants

- Possibilité de créer des tableaux ou des collections de références de type de la classe abstraite

2.3.1 Exemple de classe abstraite

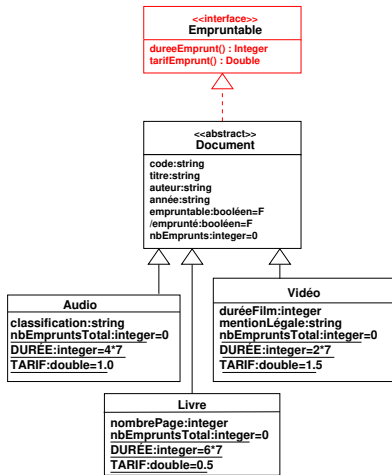
Classe seance5.mediathequesimplifiee.classeabstraite.Document

```
1 public abstract class Document {
2     // ...
3     public boolean emprunter() {
4         emprunte = true; genre.emprunter(); nbEmprunts++; return true; }
5     public abstract int dureeEmprunt(); // non definie
6     public abstract double tarifEmprunt(); // non definie
7     // ...
```

Classe seance5.mediathequesimplifiee.classeabstraite.Audio

```
1 // ...
2 @Override // redefinition avec respect du prototype
3 public boolean emprunter() { super.emprunter(); nbEmpruntsTotal++; return
4     true; }
5 @Override // definition avec respect du prototype
6 public int dureeEmprunt() { return DUREE; }
7 @Override // definition avec respect du prototype
8 public double tarifEmprunt() { return TARIF; }
9 // ...
```

2.4 Interfaces



- Les membres d'une interface sont principalement¹ des méthodes abstraites
- Une interface est déclarée avec le mot réservé `interface`
- Une interface peut hériter (extends) de plusieurs interfaces
- Une classe peut implémenter (implements) une ou plusieurs interfaces
- Rappel : une classe n'hérite (extends) que d'une seule classe

1. Depuis la version 5, possibilité de définir des constantes (`public + final`), et depuis la version 8, possibilité de définir des *default methods* ainsi que des méthodes *static*

2.4.1 Exemple d'interface

Interface seance5.mediathequesimplifiee.exempleinterface.Empruntable

```
1 public interface Empruntable {
2     int dureeEmprunt(); // definition d'un prototype (contrat)
3     double tarifEmprunt(); // definition d'un prototype (contrat)
4 }
```

Classe seance5.mediathequesimplifiee.exempleinterface.Document

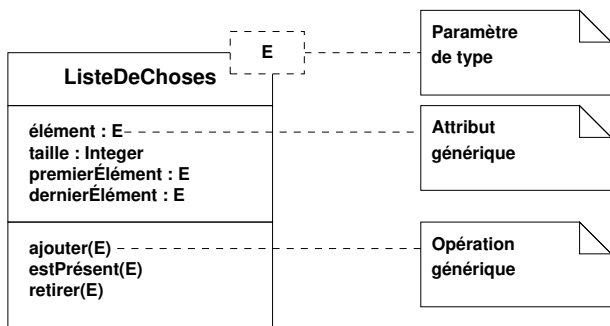
```
1 public abstract class Document implements Empruntable { // implementation
```

Classe seance5.mediathequesimplifiee.exempleinterface.Audio

```
1 public final class Audio extends Document { // heritage
2     // ...
3     @Override // respect du prototype de l'interface
4     public int dureeEmprunt() { return DUREE; }
5     @Override // respect du prototype de l'interface
6     public double tarifEmprunt() { return TARIF; }
7     // ...
```

3 Classe paramétrée (par un type)

- Polymorphisme paramétrique; *generics* dans la terminologie JAVA



- Paramètre de type = classe associée à la classe des objets manipulés
- Classe associée spécifiée entre « <> »
 - P.ex. `ArrayList<String>` `myStringArray = new ArrayList<>()`;

4 Énumération, annotation, argument variadique

- **Énumération = type énuméré en JAVA** Classe `seance5.enumeration.Dimension`

```
1 public enum Dimension { A4, A5, LETTER; }
```

- **Annotation = sucre syntaxique associant des méta-données à un élément**

- Nom introduit par « @ » avec possiblement des arguments (`nom=valeur`)
- Annotations standards
 - **@Override** : utilisée par le compilateur pour vérifier l'invariance des prototypes lors de la redéfinition de méthode
 - **@Deprecated** : utilisée par Javadoc pour déconseiller l'utilisation de la méthode
 - **@SuppressWarnings** : utilisée par le compilateur pour ne pas générer les avertissements concernant l'élément annoté

- Dans la séance 6, les annotations de **JUNIT**

- **Argument variadique en JAVA** Classe `seance5.argumentvariadique.ExempleVarArg`

```
1 private static String format(final String... strings) {  
2     Objects.requireNonNull(strings);  
3     var result = new StringBuffer();  
4     for (String string : strings) { result.append(" ").append(string); }  
5     return result.toString(); }
```

5 Visibilité, égalité et hachage

- 5.1 Visibilité des attributs et des opérations
- 5.2 Égalité de contenu — Méthode `equals`
- 5.3 Hachage — Méthode `hashCode`

5.1 Visibilité des attributs et des opérations

```
package p1;

class C1 {
    public int a;
    protected int b;
    int c; // package
    private int d;
}

class C2 extends C1 {
    ...
}

class C3 {
    ...
}
```

```
package p2;

class C4 extends C1 {
    ...
}

class C5 {
    ...
}
```

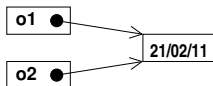
- **private** = seule la classe courante
- **package** = private + classes du paquetage
 - *package* par défaut
- **protected** = *package* + classes enfants
- **public** = toutes les classes

	a	b	c	d
Accessible de C2	oui	oui	oui	–
Accessible de C3	oui	oui	oui	–
Accessible de C4	oui	oui	–	–
Accessible de C5	oui	–	–	–

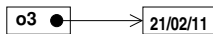
5.2 Égalité de contenu — Méthode equals

Égalité de référence *Versus* égalité de contenu d'objet

- `o1 == o2` \triangleq références égales
(même objet en mémoire)



- `o1.equals(o3)` \triangleq contenus égaux
(objets peut-être différents en mémoire)



Classe `seance5.collections.ExempleCalendarEquals`

```
1 Date initial = Calendar.getInstance().getTime(); // valeur initiale
2 Calendar o1 = Calendar.getInstance(); o1.setTime(initial);
3 Calendar o2 = o1; // egalite de reference
4 Calendar o3 = // meme valeur initiale
5     Calendar.getInstance(); o3.setTime(initial);
6 System.out.println("o1==o2:" + (o1 == o2)
7     + ",\t" + "o1.equals(o2):" + (o1.equals(o2)));
8 System.out.println("o1==o3:" + (o1 == o3)
9     + ",\t" + "o1.equals(o3):" + (o1.equals(o3)));
```

Traces d'exécution :

```
o1 == o2 : true, o1.equals(o2) : true
o1 == o3 : false, o1.equals(o3) : true
```

5.2.1 Méthode equals avec instanceof

Classe `seance5.mediathequesimplifiee.equalshashcode.ClientAvecInstanceof`

```
1 private String nom, prenom, adresse;
2 private int nbEmpruntsEnCours = 0, nbEmpruntsDepasses = 0,
   nbEmpruntsEffectues = 0;
3 private Date dateRenouvellement, dateInscription;
4 private int codeReduction = 0;
```

Méthode `equals` (égalité de contenu) : clients « égaux » $\hat{=}$ mêmes noms et mêmes prénoms

```
1 @Override // Default implementation in Object is (this == obj)
2 public boolean equals(final Object obj) { // Generated with Eclipse
3     if (obj == null) { return false; }
4     if (this == obj) { return true; }
5     if (!obj instanceof ClientAvecInstanceof) { return false; }
6     ClientAvecInstanceof c = (ClientAvecInstanceof) obj;
7     return (nom.equals(c.nom) && prenom.equals(c.prenom));
8 }
```

■ Dans ECLIPSE, menu *Source* puis *Generate hashCode() and equals()*

■ Lors de la génération, cocher la case « Use “instanceof” to compare types »

. Comme suggéré par J. Bloch [Bloch, 2008], notre préférence va à cette façon de faire, c'est-à-dire `equals` mis en œuvre avec `instanceof`

5.2.2 Méthode equals avec getClass

Classe `seance5.mediathequesimplifiee.equalshashcode.ClientAvecGetClass`

```
1 private String nom, prenom, adresse;
2 private int nbEmpruntsEnCours = 0, nbEmpruntsDepasses = 0,
   nbEmpruntsEffectues = 0;
3 private Date dateRenouvellement, dateInscription;
4 private int codeReduction = 0;
```

Méthode `equals` (égalité de contenu) : clients « égaux » $\hat{=}$ mêmes noms et mêmes prénoms

```
1 @Override // Default implementation in Object is (this == obj)
2 public boolean equals(final Object obj) { // Generated with Eclipse
3     if (obj == null) { return false; }
4     if (this == obj) { return true; }
5     if (getClass() != obj.getClass()) { return false; }
6     ClientAvecGetClass c = (ClientAvecGetClass) obj;
7     return (nom.equals(c.nom) && prenom.equals(c.prenom));
8 }
```

■ Dans ECLIPSE, menu *Source* puis *Generate hashCode() and equals()*

5.3 Hachage — Méthode hashCode

- Méthode utilisée pour calculer un indice dans certaines collections, plus précisément les dictionnaires
- Contrat de base :
 - Doit retourner la même valeur lorsqu'elle est appelée plusieurs fois sur le même objet
 - $\forall o_1, o_2 : o_1.equals(o_2) \implies o_1.hashCode() = o_2.hashCode()$
 - La contraposée n'est pas vraie
 - Peut retourner la même valeur pour deux objets différents
 - C'est la limite du hachage

Espace des objets



Espace des entiers
du hachage



5.3.1 Exemple hashCode

Classe `seance5.mediathequesimplifiee.equalshashcode.Client`

```
1 private String nom, prenom, adresse;
2 private int nbEmpruntsEnCours = 0, nbEmpruntsDepasses = 0,
   nbEmpruntsEffectues = 0;
3 private Date dateRenouvellement, dateInscription;
4 private int codeReduction = 0;
```

Méthode `hashCode` : mêmes attributs que ceux utilisés dans `equals` (nom, prénom)

```
1 public int hashCode() { // Generated with Eclipse
2     final int prime = 37, magic = 17;
3     int result = magic;
4     if (nom != null) { result += nom.hashCode(); }
5     if (prenom != null) { result = prime * result + prenom.hashCode(); }
6     return result;
7 }
```

- Algorithme utilisé par Eclipse = celui de J. Bloch [Bloch, 2008]
 - Dans ECLIPSE, menu *Source* puis *Generate hashCode() and equals()*

6 Bibliothèque, avec l'exemple des collections

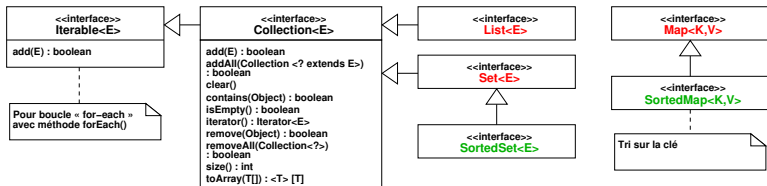
- 6.1 Organisation des bibliothèques
- 6.2 Quelques interfaces
- 6.3 Quelques classes concrètes
- 6.4 Algorithmes dans la classe Collections
- 6.5 Dictionnaires Map<K,V>
- 6.6 Structure d'une HashMap, equals et hashCode
- 6.7 Parcours de collections

6.1 Organisation des bibliothèques

- P.ex. les collections pour les associations avec des multiplicités $\neq 0..1$
- Conceptuellement, une **bibliothèque** =
 - + **Interface** = types de données abstraits
 - Les interfaces constituent un **standard** de fait du langage
 - + **Classes concrètes** = structures de données
 - Les classes concrètes sont les mises en œuvre de **référence**
 - Possibilité de construire de nouvelles mises en œuvre
 - + **Algorithmes** = méthodes de classe pour trouver, trier, etc.
 - Méthodes **polymorphiques**, donc applicables à beaucoup de classes concrètes

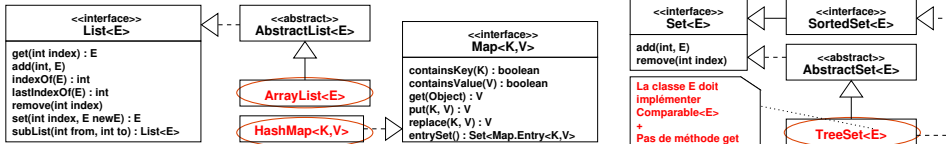
. Autres langages : *C++ Standard Template Library (STL)*, *Smalltalk's collection hierarchy*

6.2 Quelques interfaces



- **List<E>** avec doublons possibles (liste), **Set<E>** sans doublon (ensemble)
- **Map<K,V>** apparie une clé ($k \in K$) et un objet ($v \in V$) (dictionnaire)
 - La clé k indentifie de façon unique v
- Toutes les interfaces sont paramétrées avec un ou plusieurs types
- Certaines méthodes sont dites optionnelles dans la documentation
 - Une classe concrète peut ne pas « supporter » des méthodes optionnelles. Dans ce cas, la mise en œuvre lève l'exception `UnsupportedOperationException`

6.3 Quelques classes concrètes



- Différentes catégories de classes concrètes
 - Mises en œuvre de base
 - Mises en œuvre pour la concurrence d'accès (cf. `java.util.concurrent`)
 - Mises en œuvre spécifiques, p.ex. avec tri des éléments
- Uniquement trois mises en œuvre de base nécessaires dans ce module
 - Pour les collections de petite taille, `ArrayList` \equiv tableau de taille dynamique
 - Pour les collections de grande taille, `HashMap` \equiv dictionnaire
 - Pour les petites collections, `TreeSet` \equiv Ensemble trié

6.4 Algorithmes dans la classe Collections

- `sort` : tri d'une liste (*merge sort algorithm*)
- `reverse` : réordonne en sens inverse
- `fill` : remplissage avec un élément donné
- `copy` : d'une liste à une autre
- `swap` : permuter deux éléments donnés
- `addAll` : ajouter tous les éléments d'une liste
- `binarySearch` : pour trouver un élément
- `frequency` : compte le nombre d'occurrences
- `disjoint` : pas d'éléments en commun
- `min` et `max` : selon l'opérateur de tri
- `shuffle` : mélange aléatoire des éléments

Classe `seance5.collections.ExempleTriListe`

```
1 import java.util.Arrays;
2 import java.util.Collections;
3 import java.util.List;
4 public class ExempleTriListe {
5     public static void main(String[] args) {
6         List<String> liste = Arrays.asList(new String[] {"x", "d", "e", "d"});
7         Collections.sort(liste); // methode generique de tri sur une liste
8         System.out.println(liste);
9     }
10 }
```

6.5 Dictionnaires Map<K, V>

- La classe associée à la clé doit redéfinir les méthodes `hashCode()` et `equals()`
 - Si la clé d'une classe contient plusieurs attributs alors il faut créer une classe pour rassembler ces attributs² (concept de n-uplet)

Classe `seance5.collections.ClefClient`

```
1 public final class ClefClient {
2     private String nom;
3     private String prenom;
4     public ClefClient(final String n, final String p) { this.nom = n; this.
        prenom = p; }
5     public boolean equals(final Object obj) { //...
6     }
7     public int hashCode() { //...
8     }
```

Classe `seance6.mediathequesimplifiee.pourlestests.Mediatheque`

```
1     private Map<ClefClient, Client> lesClients;
2     lesClients = new Hashtable<ClefClient, Client>();
3     ClefClient hc = new ClefClient(cl.getNom(), cl.getPrenom());
```

2. Sinon, besoin de créer des objets factices pour obtenir un élément

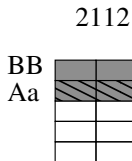
6.6 Structure d'une HashMap, equals et hashCode

Classe `seance5.collections.ExempleHashMap`

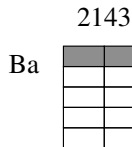
```
1 Map<String, String> hm = new HashMap<>();
2 hm.put("Ba", "Bah"); hm.put("Aa", "aha"); hm.put("BB", "bebe");
3 for (Entry<String,String> e : hm.entrySet()) {
4     System.out.println(e.getKey() + " (" + e.getValue().hashCode() + ") " + e
        .getValue()); }
```

Traces d'exécution

Clef	hashCode()	:	Valeur
Aa	(2112)	:	aha
BB	(2112)	:	bebe
Ba	(2143)	:	Bah



Bebe
Aha



Bah

■ Recherche d'une valeur v à partir d'une clé k :

1. `hashCode(k)` donne l'ensemble des clés ayant même valeur de hachage
2. `equals(k)` donne la bonne clé, puis la valeur v

6.7 Parcours de collections

- Construction *for-each* : utilise **implicitement** un itérateur

- Pas d'accès à l'itérateur \implies pas de suppression d'élément dans la boucle

Classe `seance5.collections.ExempledeParcoursDeCollection`

```
1 for ((Entry<String,Integer> entry:dictionnaire.entrySet()) {
2     System.out.print(entry.getKey() + "->" + entry.getValue() + ",_");
3     // dictionnaire.remove(entry.getKey()); //=> exception ConcurrentModif...
4 }
```

- Construction avec *while* et un itérateur **explicite**

- `hasNext()` : retourne vrai si l'itérateur contient encore un élément
- `next()` : retourne l'élément suivant
- `remove()` : supprime le dernier élément retourné par `next()` de la collection³

Classe `seance5.collections.ExempledeParcoursDeCollection`

```
1 Iterator<String> itElem = liste.iterator();
2 while (itElem.hasNext()) { // iterateur explicite
3     String elem = itElem.next(); System.out.print(elem + ",_");
4     itElem.remove(); // suppression possible
5 }
```

- Cf. `...ExempledeParcoursDeCollection` pour des exemples de parcours

3. Possible, mais un peu dangereux et à éviter

7 Construction de logiciel avec Maven

7.1 Motivations — Pourquoi Maven ?

7.2 Dépôt et arborescence standard

7.3 Processus de construction, version simplifiée

. <https://maven.apache.org> et
<http://books.sonatype.com/mvnex-book/reference/index.html>

7.2 Dépôt et arborescence standard

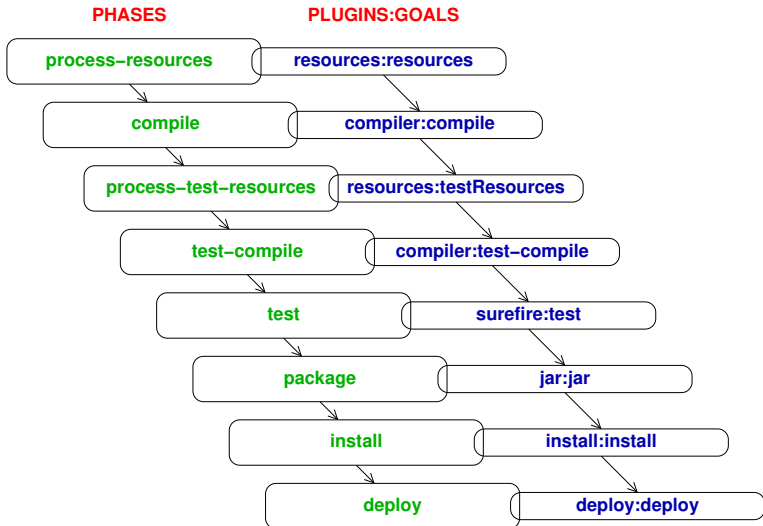
■ Dépôts (*Repositories*)

- Beaucoup de dépôts sont disponibles
 - P.ex. <http://search.maven.org/> ou <https://mvnrepository.com/>
- Dépôt local dans le répertoire: `~/.m2/repository`
 - Peuplé comme un *cache* des dépôts distants

■ Arborescence standard — version simplifiée

- `pom.xml`: POM (*Project Object Model*), toujours au niveau le plus élevé
 - Déclaration de configuration du processus de construction du module
- `LICENSE.txt` et `README.md`
- `src/main/java`: code source du module
- `src/test/java`: code source des tests du module
- `target`: tous les contenus générés (`*.class`, `*.jar`, site Web...)

7.3 Processus de construction, version simplifiée



8 Mise en pratique en TP (2h) + HP (3h)

- Étude du module `MAVEN` préparé, création du projet Eclipse
Découverte de la bibliothèque `eu.telecomsudparis.csc4102.util` (dates, console)
- Début de la programmation de l'application
 - Classes, attributs, constructeurs, méthodes `invariant()`
 - Au besoin, méthodes `equals` et `hashCode`, voire `toString`
 - On programme des cas d'utilisation (algo. des méthodes dans les diagrammes de séquences)
 - Et non « une classe complète, puis une autre, etc. »
- Rendu de la séance en HP (3h): PDF + JAVA, dans « `sprint1` »
 - Le script de fin de séance vérifie que le code de votre projet compile
- Compléments « Pour aller plus loin » sur la construction de logiciel avec `MAVEN`

Références I

Bloch, J. (2008).

Effective Java, 2nd Edition.

Addison-Wesley.

Hunt, A. and Thomas, D. (2000).

The Pragmatic Programmer: From journeyman to master.

Addison-Wesley.

Maven (2016).

Apache Maven Project.

<https://maven.apache.org>.

O'Brien, T., Brian, J., van Zyl, J., Xu, J., Locher, T., Fabulich, D., Redmond, E., and Snyder, B. (2016).

Maven by Example.

<http://books.sonatype.com/mvnex-book/reference/index.html>.

Creative Commons License.