

Stage de Programmation Objet

LP DAWIN



Informatique



Partie 4



Collections et bonus

Partie 4 : Collections et bonus

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Tester l'égalité de variables : types primitifs

Pour les types primitifs (int, float, char, etc), on utilise == :

```
int i, j;  
char c, d;  
...  
if (i == j || c == d) {  
    ...  
}
```

Tester l'égalité de variables : objets

Quand deux variables de type objet sont-elles "égales" ?

- Si elles référencent **la même instance**, elles sont forcément égales. On peut le tester avec `==`.
- Mais dans le cas général, il faut le préciser...

Par exemple pour les salles on peut considérer que *deux instances représentent la même salle si elles ont le même numéro.*

Tester l'égalité de variables : objets

Pour cela tout objet possède une méthode `equals` que l'on peut redéfinir :

```
@Override
public boolean equals(Object obj) {
    if (this == obj) { // meme instance => egaux
        return true;
    }
    if (obj == null) { // this n'est jamais null, par definition...
        return false;
    }
    if (getClass() != obj.getClass()) { // pas la meme classe => differents
        return false;
    }
    final Salle other = (Salle) obj; // on cast en Salle
    if (this.numero != other.numero) { // on compare leurs numeros
        return false;
    }
    return true;
}
```

On peut générer ce code automatiquement. Dans NetBeans : Alt+Insérer / equals() and hashCode() puis choisir les attributs discriminants.

Tester l'égalité de variables : objets

```
class Salle {  
    static Salle laPlusCool() {  
        ...  
        return new Salle (...);  
    }  
    ...  
}
```

```
Salle salle301 = new Salle (...);  
Salle salleLaPlusCool = Salle.laPlusCool();  
  
if (salle301.equals(salleLaPlusCool)) {  
    ...  
}
```

`Salle.laPlusCool()` renvoie une nouvelle instance de la salle 301, donc

`salle301 == salleLaPlusCool`

renvoie **false**, alors que

`salle301.equals(salleLaPlusCool)`

renvoie **true**.

Tester l'égalité de variables

Moralité :

Types primitifs : utiliser `==`

```
int i, j;  
...  
if (i == j) {  
    ...  
}
```

Objets : utiliser `equals()`

```
Salle salle1, salle2;  
...  
if (salle1.equals(salle2)) {  
    ...  
}
```

Seule exception : tester si un objet est null

```
if (salle1 == null) { ... }
```

Partie 4 : Collections et bonus

- 1 Collections
 - Préambule : Test d'égalité
 - **Généricité**
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Généricité

Nous allons nous intéresser aux collections :
listes, piles/files, tableaux extensibles, tableaux associatifs...

Ce sont naturellement des **types génériques** :

- **liste = liste de quelque chose** : d'entiers, de booléens. . .
- pour chaque type (entiers, booléens), on veut les mêmes fonctionnalités : ajouter un élément, accéder au premier élément, etc
- on veut donc une implémentation commune, et pas `ListInteger`, `ListBoolean`, etc

→ La **généricité** permet de *paramétrer une classe (ou interface) par un type quelconque*.

Généricité : exemple

```
class Entrepot<T> {  
    Position pos;  
    T[] elements;  
    ...  
}
```

```
...  
// un entrepot de 10 containers  
Entrepot<Container> entrepot1 = new Entrepot<>();  
entrepot1.elements = new Container[10];  
entrepot1.elements[0] = new Container(12);  
entrepot1.elements[1] = new Container(31);  
...  
entrepot1.elements[9] = new Container(47);  
...  
// un entrepot de 1500 caisses  
Entrepot<Caisse> entrepot2 = new Entrepot<>();  
entrepot2.elements = new Caisse[1500];  
entrepot2.elements[0] = new Caisse("AH-2385-B29");  
...
```

penser au diamant : `new Entrepot<>()` et pas `new Entrepot()`

autrefois : `new Entrepot<Container>()`

maintenant : inférence de type

Généricité : intérêt

```
class Entrepot<T> {
    Position pos;
    T[] elements;
    ...
}
```

Plusieurs intérêts :

- une seule implémentation !
- contrôle du type en entrée :

```
Entrepot<Container> entrepot = new Entrepot<>();
entrepot.elements[0] = new Container(); // ok
entrepot.elements[0] = new Caisse(); // interdit (a la compilation)
```

- garantie du type en sortie :

```
Entrepot<Caisse> entrepot = new Entrepot<>();
...
for (int i=0; i<entrepot.elements.length; i++) {
    Caisse c = entrepot[i]; // pas besoin de transtypage
    c.afficher(); // polymorphisme, si plusieurs types de caisses
    ...
}
```

Partie 4 : Collections et bonus

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Cas 1 : stocker un ensemble d'objets

Les tableaux

```
Caisse[] elements = new Caisse[1500];  
elements[0] = new Caisse("EJ-9505-C11");  
...  
Caisse c = elements[10]; // acces direct
```

- taille fixe / temps d'accès constant
- déconseillé, au profit des tableaux extensibles (prochain slide)

Les listes

- List est une interface

```
List<Caisse> elements = new LinkedList<>(); // par exemple  
elements.add(new Caisse("EJ-9505-C11"));  
...  
Caisse c = elements.get(10); // parcourt 11 elements
```

- taille variable / temps d'accès linéaire

Cas 1 : stocker un ensemble d'objets

Les tableaux extensibles

- taille variable, temps d'accès/ajout constants (en moyenne)

```
List<Caisse> elements = new ArrayList<>(); // implements List
elements.add(new Caisse("EJ-9505-C11"));
...
Caisse c = elements.get(10); // acces direct
```

Les ensembles (non ordonnés)

- Set = interface d'un ensemble (non ordonné, sans doublons)
- implémentations courantes : HashSet, TreeSet
- taille variable

```
Set<Caisse> elements = new HashSet<>(); // par exemple
Caisse c = new Caisse("EJ-9505-C11");
elements.add(c);
...
if (elements.contains(c)) { ... }
```

Cas 2 : stocker une association (ou dictionnaire)

Les tableaux associatifs (ou dictionnaires)

- Un **tableau associatif** peut être vu comme :
 - un ensemble de **couples clé/valeur**, ou
 - une **fonction** (en math), associant à chaque clé x une valeur y
- pas une collection
- l'interface **Map** décrit les opérations des tableaux associatifs.
- implémentations courantes : **HashMap**, **TreeMap**

```
Map<Caisse, Entrepot<Caisse>> localisation = new HashMap<>(); // par ex
Caisse c = new Caisse("EJ-9505-C11");
Entrepot<Caisse> e = new Entrepot<>();
localisation.put(c, e);
...
Caisse c2 = ...
if (localisation.containsKey(c2)) {
    Entrepot<Caisse> e2 = localisation.get(c2);
    ...
}
```

Collections et égalité

ATTENTION : les collections sont basées sur des **tests d'égalité**, donc :

- il faut une implémentation d'**equals** correcte
par exemple `Salle.equals()` quand on fait un `Set<Salle>`
- pour `HashSet` et `HashMap`, sur les clés, il faut :
 - une implémentation de **hashCode()** correcte (cf doc `Object.hashCode()`) :
si `e1.equals(e2)` alors `e1.hashCode() == e2.hashCode()`
 - si possible des **clés immuables** (sinon le `hashCode` peut changer)

Utilitaires

Des algorithmes utiles “classiques” (sort...) sont disponibles :

- dans `java.util.Arrays` pour les tableaux
- dans `java.util.Collections` pour les collections

Partie 4 : Collections et bonus

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Boucles : sur les ensembles

Sur les collections comme sur les tableaux, une **boucle simplifiée** parcourt les éléments :

```
Caisse[] caisses = new Caisse[1200];  
...  
for (Caisse c : caisses) {  
    c.affranchir();  
}
```

```
List<Caisse> caisses = new ArrayList<>();  
...  
for (Caisse c : caisses) {  
    c.affranchir();  
}
```

plus lisible, mais :

- pas d'accès aux indices
- pas pour les boucles "tant que" (ou alors, break)

Boucles : sur les tableaux associatifs

```
Map<Caisse , Entrepot<Caisse>> localisation = ...
```

3 boucles :

- sur les **clés** :

```
for (Caisse c : localisation.keySet()) {  
    Entrepot<Caisse> e = localisation.get(c);  
    ...  
}
```

- sur les **valeurs** :

```
for (Entrepot<Caisse> e : localisation.values()) {  
    ...  
}
```

- sur les **couples clé/valeur** :

```
for (Map.Entry<Caisse , Entrepot<Caisse>> ce : localisation.entrySet()) {  
    Caisse c = ce.getKey();  
    Entrepot<Caisse> e = ce.getValue();  
    ...  
}
```

Itérateurs

Pour gérer des boucles “à la main” (passage d'un élément au suivant, test si dernier, etc) on utilise des **itérateurs**.

```
interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove(); // optionnel  
}
```

Exemple d'utilisation :

```
ArrayList<String> al = new ArrayList<>();  
...  
Iterator<String> itr = al.iterator();  
while (itr.hasNext()) {  
    String element = itr.next();  
    System.out.print(element + " ");  
    if (element.equals("D")) {  
        itr.remove();  
    }  
}  
System.out.println();
```

Partie 3 : Héritage

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Mini-projet : v6 → v7

Niveau 1

- on gère désormais un ensemble de pièces disponibles par joueur, plutôt qu'un compteur
- pour cela ajouter une classe `Piece`.
- on retire `LimitedGame` et `LimitedGameState` et, pour `Fill` et `Crush`, on met autant de pièces que de cases (pour simplifier)
- `GameState` contient les pièces restantes par joueur :
`private Map<Player, Set<Piece>> pieces;`

Partie 4 : Collections et bonus

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Un exemple

```
class Salle implements Comparable {  
    int numero;  
    int capacite;  
    boolean salleMachine;  
    Salle(int num, int capa, boolean sMachine) { ... }  
    ...  
}
```

J'ai l'ensemble des salles du bâtiment 9A :

```
final Set<Salle> bat9A = new HashSet<>();  
bat9A.add(new Salle(3, 28, false));  
bat9A.add(new Salle(4, 40, true));  
bat9A.add(new Salle(5, 28, false));  
...
```

Je veux afficher les numéros des salles machines, dans l'ordre, préfixés par "9A :".

```
9A : 4  
9A : 208  
...
```

```
bat9A.stream()  
    .filter(s -> s.salleMachine) // salles machines
```

Comparons

Version “streams” :

```
bat9A.stream()
    .filter(s -> s.salleMachine) // salles machines
    .sorted()                    // dans l'ordre
    .map(s -> "9A : " + s.numero) // prefixees par 9A
    .forEach(System.out::println); // on affiche
```

Version “collections” :

```
// salles machines
ArrayList<Salle> machines = new ArrayList<>();
for (Salle s : bat9A) {
    if (s.salleMachine) {
        machines.add(s);
    }
}
// dans l'ordre
Collections.sort(machines);
// on affiche
for (Salle s : machines) {
    System.out.println("9A : " + s.numero);
}
```

- code deux fois plus long
- structure intermédiaire (machines)
- deux boucles

Analysons

```
bat9A.stream()
    .filter(s -> s.salleMachine) // salles machines
    .sorted()                     // dans l'ordre
    .map(s -> "9A : " + s.numero) // prefixees par 9A
    .forEach(System.out::println); // on affiche
```

- “pipeline” (enchaînement) :
bat9A.stream().filter(...).sorted().map(...).forEach(...);
- il y a 4 streams :
 - bat9A.stream()
 - puis ceux obtenus après filter(...), sorted() et map(...)
- puis forEach(...) effectue un traitement sur chaque élément, sans produire de stream.

Noter aussi les **lambdas** : `s -> s.salleMachine`.

= fonctions anonymes

Évaluation paresseuse

Au lieu de trier, on affiche une seule salle machine :

```
bat9A.stream()  
  .filter(s -> s.salleMachine) // salles machines  
  .limit(1)                     // une seule  
  .map(s -> "9A : " + s.numero) // prefixee par 9A  
  .forEach(System.out::println); // on affiche
```

L'évaluation se fait **à la demande** :

- 1 foreach demande un premier élément à map,
- 2 map demande à limit,
- 3 limit demande à filter,
- 4 filter demande à stream,
- 5 stream prend un premier élément de bat9A et le renvoie à filter,
- 6 filter le rejette car ce n'est pas une salle machine (par exemple),
- 7 filter demande un nouvel élément à stream
- 8 stream prend un deuxième élément dans bat9A et le renvoie à filter,
- 9 filter l'accepte (par exemple), et le transmet à limit,
- 10 limit le renvoie à map en mémorisant qu'il a renvoyé un élément
- 11 map transforme la salle en String et la renvoie à forEach,
- 12 forEach affiche cette String, et demande l'élément suivant à map
- 13 map demande à limit
- 14 limit indique à map qu'il a fini
- 15 map indique à forEach qu'il a fini
- 16 forEach se termine.

Structure en pipeline

```
bat9A.stream()  
  .filter(s -> s.salleMachine) // salles machines  
  .limit(1)                     // une seule  
  .map(s -> "9A : " + s.numero) // prefixee par 9A  
  .forEach(System.out::println); // on affiche
```

De manière générale, il y a souvent 3 types d'opérateurs sur les streams :

- une **source** (ici `bat9A.stream()`),
- des **opérations intermédiaires** (ici `filter`, `limit`, `map`),
- une **opération terminale** (ici `forEach`).

Nous allons considérer ces 3 types d'opérateurs.

Partie 3 : Héritage

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Sources liées aux collections

Chaque **collection** (Set, ArrayList, etc) possède une méthode **stream()** qui renvoie un stream avec les mêmes éléments.

```
ArrayList<Caisse> caisses = ...  
caisses.stream(). ...
```

Pour les **Map**, c'est la même chose sur les 3 collections liées : `keySet()`, `values()` et `entrySet()`.

```
Map<Caisse, Batiment> emplacement = ...  
emplacement.keySet().stream(). ...
```

Autres sources

- entiers d'un intervalle :

```
IntStream.range(10, 20) // entiers entre 10 et 20
           .filter(i -> i%2==0) // seulement les pairs
           .forEach(System.out::println);
```

- lignes d'un fichier :

```
Stream<String> lignes = Files.lines(Paths.get("data.txt"));
```

- fichiers d'un répertoire :

```
Files.walk(new File("/tmp").toPath())
         .map(Path::getFileName)
         .forEach(System.out::println);
```

- etc.

Sources, via les méthodes statiques de Stream

- `Stream.empty()` retourne le stream vide (aucun élément)
- `Stream.of(new Salle(202, 28, false))`
→ Stream à 1 élément
- `Stream.of(salle1, salle2, salle3)`
→ Stream à 3 éléments
- `Stream.iterate(1, n -> n + 2)`
→ premier élément, et fonction pour passer au suivant.
Stream infini, donc utiliser `limit(n)` (présenté après)
- `Stream.generate(Math::random)`
→ appel à une fonction ne prenant aucun argument.
Stream infini aussi.

Partie 3 : Héritage

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Opérations intermédiaires : filter et map

Ces opérations prennent un **stream**, et retournent un **stream**.

2 opérations ont une importance centrale en prog. fonctionnelle :

- **filter** prend en paramètre un **prédicat**, qui s'applique à un élément du stream et indique s'il est sélectionné.
- **map** prend en paramètre **une fonction**, qui sera appliquée à chaque élément du stream.

Remarque

les **prédicats** et les **fonctions** sont des objets, on peut les définir ailleurs, pour simplifier les enchaînements de streams.

```
Predicate<Salle> nonMachineAu3eme =  
    s -> !s.salleMachine && s.numero / 100 == 3;  
bat9A.stream()  
    .filter(nonMachineAu3eme)  
    .forEach(System.out::println);
```

Opérations intermédiaires

D'autres opérations intermédiaires sont disponibles :

- `sorted()` : pour trier (nécessite d'implémenter `Comparable`)
- `limit(int n)` : prend uniquement les `n` premiers éléments
- `skip(int n)` : saute les `n` premiers éléments
- `distinct()` : supprime les doublons
- etc.

Partie 3 : Héritage

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - **Opérations terminales**
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Opérations terminales : replier / réduire

La troisième **opération fondamentale** en prog. fonctionnelle, après `filter` et `map`, est la **réduction** (`reduce`, ou `fold`, selon les langages).

Elle **effectue un calcul** à partir des éléments du stream, en accumulant une valeur intermédiaire (**accumulateur**).

Exemple

```
int somme = IntStream.range(1, 10).reduce(0, (a, i) -> a + i);
```

calcule la somme des entiers entre 1 et 10 :

- `IntStream.range(1, 10)` renvoie le stream des entiers de 1 à 10
- l'argument 0 de `reduce` est la valeur de départ de l'accumulateur
- la fonction `(a,i) -> a + i` indique la nouvelle valeur de l'accumulateur `a` après la lecture de l'élément `i`

Opérations terminales : vers les collections

On peut vouloir **convertir un stream en collection**.

Pour cela on utilise :

- l'opération terminale **collect** et
- les méthodes statiques de la classe utilitaire **Collectors**

vers une liste :

```
List<Salle> sallesMachines = bat9A.stream()
    .filter(s -> s.salleMachine) // salles machines
    .sorted() // dans l'ordre
    .collect(Collectors.toList()); // on convertit en liste
```

vers un ensemble :

```
List<Salle> sallesMachines = bat9A.stream()
    .filter(s -> s.salleMachine) // salles machines
    .sorted() // dans l'ordre
    .collect(Collectors.toCollection(TreeSet::new)); // vers TreeSet
```

Opérations terminales : autres

D'autres opérations terminales sont disponibles :

- `sum()` retourne la somme des éléments d'un stream d'entiers
- `min()`, `max()` : minimum et maximum d'un stream (nécessite un comparateur)
- `count()` retourne le nombre d'éléments du stream
- `findFirst()` renvoie uniquement le premier élément du stream, s'il existe (via un `Optional`)
- `findAny()` renvoie n'importe quel élément du stream, s'il existe (parallélisable)
- `noneMatch(predicat)`, `anyMatch(predicat)`, `allMatch(predicat)` : renvoie vrai ssi (respectivement) 0, au moins 1, ou tous les éléments, satisfont le prédicat

Partie 3 : Héritage

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Mini-projet : v7 → v8

Niveau 1

- créer la classe `game.Coord`, stockant des coordonnées
- l'utiliser partout, à la place des couples (ligne, colonne)
- implémenter et tester `Coord.near()`, qui renvoie un `Stream<Coord>` des coordonnées des cases voisines du plateau, dans les 4 directions N/S/E/O
- implémenter le jeu `near`, similaire à `fill`, mais lorsqu'un joueur pose une pièce sur une case voisine d'une case occupée par l'adversaire, il pose en même temps des pièces sur toutes les cases voisines libres.

Niveau 2

→ slide suivant

Mini-projet : v7 → v8

Niveau 2

- dans `Coord`, implémenter la méthode :

```
Optional<Coord> nextCoord(  
    int stepL, int stepC, Predicate<Coord> pred)
```

qui renvoie (si elle existe) les coordonnées de la prochaine case vérifiant `pred`, en partant des coordonnées de `this`, et en sautant de `(stepL, stepC)`

- utiliser cette méthode pour trouver la prochaine case non vide vers la droite occupée par l'adversaire

Partie 4 : Collections et bonus

- 1 Collections
 - Préambule : Test d'égalité
 - Généricité
 - Collections usuelles
 - Boucles
 - Mini-projet
- 2 Streams et fonctionnel
 - Un premier exemple
 - Sources
 - Opérations intermédiaires
 - Opérations terminales
 - Mini-projet
- 3 Conclusion
 - Extensions possibles

Pour aller plus loin

Ce que l'on a pas abordé faute de temps...

... mais vous avez le droit de vous y intéresser...

Java standard

- les **exceptions** : principe, lancer, rattraper
- **var** et l'inférence de type
- les **annotations**
- les **varargs**
- le **nouveau switch** et les **blocs de texte** (Java 13)
- la **réflexion**
- ...

Conception objet

- UML
- les patrons de conception :
MVC, composite, décorateurs, factory...
- les principes SOLID

et JEE...

Outils

- intégration continue : gitlab, jenkins...
- couverture de code : emma...
- qualité du code : SonarQube...
- compilation / gestion des dépendances : gradle, maven, ant...