

Programmation objet en Java

Vincent Vidal

Maître de Conférences

Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage

Recherche : Laboratoire LIRIS - bât. Nautibus

E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : <http://spiralconnect.univ-lyon1.fr> module "S2 - M2103 - Java et POO"

48H prévues ≈ 39H de cours-TDs + 6H de TPs, 1H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Plan

- 1 Les flux et les fichiers
 - Les flux
 - Les flux binaires
 - Les flux texte
 - La persistance des objets avec les flux d'objets
- 2 La programmation générique
- 3 Les collections et les algorithmes
- 4 Java 8 : Expressions lambda, streams et l'API java.time

Définitions

Un flux (*Stream* en anglais)

Un flux est un objet manipulant de l'information sous la forme d'une **suite d'octets ordonnée**.

L'information va d'un point à un autre : **circulation d'information ordonnée**. Du disque dur à la RAM et vice versa, de la RAM d'un ordi distant à celle de son ordi...

Flux vidéo, flux audio, ... : flux de données.

Définitions

Un flux (*Stream* en anglais)

Un flux est un objet manipulant de l'information sous la forme d'une **suite d'octets ordonnée**.

L'information va d'un point à un autre : **circulation d'information ordonnée**. Du disque dur à la RAM et vice versa, de la RAM d'un ordi distant à celle de son ordi...

Flux vidéo, flux audio, ... : flux de données.

Définitions

Un flux (*Stream* en anglais)

Un flux est un objet manipulant de l'information sous la forme d'une **suite d'octets ordonnée**.

L'information va d'un point à un autre : **circulation d'information ordonnée**. Du disque dur à la RAM et vice versa, de la RAM d'un ordi distant à celle de son ordi...

Flux vidéo, flux audio, ... : flux de données.

Définitions

Un flux (*Stream* en anglais)

Un flux est un objet manipulant de l'information sous la forme d'une **suite d'octets ordonnée**.

L'information va d'un point à un autre : **circulation d'information ordonnée**. Du disque dur à la RAM et vice versa, de la RAM d'un ordi distant à celle de son ordi...

Flux vidéo, flux audio, ... : flux de données.

Définitions

Un flux de sortie

Un flux de sortie est un objet **recevant** de l'information sous la forme d'une suite d'octets.

Un flux d'entrée

Un flux d'entrée est un objet **délivrant** de l'information sous la forme d'une suite d'octets.

Définitions

Un flux de sortie

Un flux de sortie est un objet **recevant** de l'information sous la forme d'une suite d'octets.

Un flux d'entrée

Un flux d'entrée est un objet **délivrant** de l'information sous la forme d'une suite d'octets.

Les flux binaires et les flux texte

Un flux binaire

Un flux binaire est un flux dont l'information ne subit aucune transformation au moment du passage de/vers la mémoire vive.

On parle souvent de **flux contenant des données brutes** (*raw data* en anglais).

Un flux texte

Un flux texte est un flux dont l'information subit un formatage afin de manipuler des caractères.

Le codage d'un caractère dans un fichier texte (ASCII, UTF-8...) est différent du codage unicode de Java sur 2 octets (UTF-16).

Les flux binaires et les flux texte

Un flux binaire

Un flux binaire est un flux dont l'information ne subit aucune transformation au moment du passage de/vers la mémoire vive. On parle souvent de **flux contenant des données brutes** (*raw data* en anglais).

Un flux texte

Un flux texte est un flux dont l'information subit un formatage afin de manipuler des caractères.

Le codage d'un caractère dans un fichier texte (ASCII, UTF-8...) est différent du codage unicode de Java sur 2 octets (UTF-16).

Les flux binaires et les flux texte

Un flux binaire

Un flux binaire est un flux dont l'information ne subit aucune transformation au moment du passage de/vers la mémoire vive. On parle souvent de **flux contenant des données brutes** (*raw data* en anglais).

Un flux texte

Un flux texte est un flux dont l'information subit un formatage afin de manipuler des caractères.

Le codage d'un caractère dans un fichier texte (ASCII, UTF-8...) est différent du codage unicode de Java sur 2 octets (UTF-16).

Les flux binaires et les flux texte

Un flux binaire

Un flux binaire est un flux dont l'information ne subit aucune transformation au moment du passage de/vers la mémoire vive. On parle souvent de **flux contenant des données brutes** (*raw data* en anglais).

Un flux texte

Un flux texte est un flux dont l'information subit un formatage afin de manipuler des caractères.

Le codage d'un caractère dans un fichier texte (ASCII, UTF-8...) est différent du codage unicode de Java sur 2 octets (UTF-16).

Les flux binaires et les flux texte

Un flux se représente "mentalement" par une suite d'octets ordonnés (l'ordre d'apparition des octets y est important). Attention un flux n'est pas obligatoirement associé à un tampon (*buffer* en anglais).

On peut lire ces octets séquentiellement (des caractères pour un flux texte) ou accéder directement à un octet particulier. Cette dernière possibilité n'est possible que pour les flux binaires.

Les flux binaires et les flux texte

Un flux se représente "mentalement" par une suite d'octets ordonnés (l'ordre d'apparition des octets y est important). Attention un flux n'est pas obligatoirement associé à un tampon (*buffer* en anglais).

On peut lire ces octets séquentiellement (des caractères pour un flux texte) ou accéder directement à un octet particulier. Cette dernière possibilité n'est possible que pour les flux binaires.

Les flux binaires et les flux texte

Un flux se représente "mentalement" par une suite d'octets ordonnés (l'ordre d'apparition des octets y est important). Attention un flux n'est pas obligatoirement associé à un tampon (*buffer* en anglais).

On peut lire ces octets séquentiellement (des caractères pour un flux texte) ou accéder directement à un octet particulier. Cette dernière possibilité n'est possible que pour les flux binaires.

Concrètement ça sert à quoi un flux ?

- Un flux sert à **représenter des sources ou des destinations "physiques"** (fichiers, programmes, blocs de mémoire, périphériques, connexions distantes, saisies clavier).
- On peut y prendre/mettre des données simples (octets, types scalaires) ou des types objets.
- Un flux peut modifier les données qu'il manipule, ou simplement les transmettre.
- D'un point de vue POO, un flux est une abstraction qui soit consomme des données (`OutputStream` et `Writer`), soit en produit (`InputStream` et `Reader`) (≈schéma producteur-consommateur) .

Concrètement ça sert à quoi un flux ?

- Un flux sert à **représenter des sources ou des destinations "physiques"** (fichiers, programmes, blocs de mémoire, périphériques, connexions distantes, saisies clavier).
- On peut **y prendre/mettre des données simples** (octets, types scalaires) **ou des types objets**.
- Un flux peut modifier les données qu'il manipule, ou simplement les transmettre.
- D'un point de vue POO, un flux est une abstraction qui soit consomme des données (`OutputStream` et `Writer`), soit en produit (`InputStream` et `Reader`) (≈schéma producteur-consommateur) .

Concrètement ça sert à quoi un flux ?

- Un flux sert à **représenter des sources ou des destinations "physiques"** (fichiers, programmes, blocs de mémoire, périphériques, connexions distantes, saisies clavier).
- On peut **y prendre/mettre des données simples** (octets, types scalaires) **ou des types objets**.
- Un flux peut modifier les données qu'il manipule, ou simplement les transmettre.
- D'un point de vue POO, un flux est une abstraction qui soit consomme des données (`OutputStream` et `Writer`), soit en produit (`InputStream` et `Reader`) (≈schéma producteur-consommateur) .

Concrètement ça sert à quoi un flux ?

- Un flux sert à **représenter des sources ou des destinations "physiques"** (fichiers, programmes, blocs de mémoire, périphériques, connexions distantes, saisies clavier).
- On peut **y prendre/mettre des données simples** (octets, types scalaires) **ou des types objets**.
- Un flux peut modifier les données qu'il manipule, ou simplement les transmettre.
- D'un point de vue POO, un flux est une abstraction qui soit **consomme des données** (`OutputStream` et `Writer`), soit **en produit** (`InputStream` et `Reader`) (≈schéma producteur-consommateur) .

Ecriture de données séquentiellement

Objectif : écrire un programme Java qui enregistre dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin ?

- **De la classe** FileOutputStream pour manipuler un flux binaire associé à un fichier en écriture : c'est une classe dérivée de la classe abstraite OutputStream qui est la classe de base de tous les flux binaires de sortie.
- D'un constructeur de la classe FileOutputStream : FileOutputStream("nomFichier.dat").

FileOutputStream f = new FileOutputStream("nomFichier.dat"); va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il est créé "vide", sinon le contenu de son ancienne version est détruit.

Ecriture de données séquentiellement

Objectif : écrire un programme Java qui enregistre dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin ?

- **De la classe** `FileOutputStream` pour manipuler un flux binaire associé à un fichier en écriture : c'est une classe dérivée de la classe abstraite `OutputStream` qui est la classe de base de tous les flux binaires de sortie.
- D'un constructeur de la classe `FileOutputStream` : `FileOutputStream("nomFichier.dat")`.

`FileOutputStream f = new FileOutputStream("nomFichier.dat");` va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il est créé "vide", sinon le contenu de son ancienne version est détruit.

Ecriture de données séquentiellement

Objectif : écrire un programme Java qui enregistre dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin ?

- **De la classe** `FileOutputStream` pour manipuler un flux binaire associé à un fichier en écriture : c'est une classe dérivée de la classe abstraite `OutputStream` qui est la classe de base de tous les flux binaires de sortie.

- D'un constructeur de la classe `FileOutputStream` : `FileOutputStream("nomFichier.dat")`.

`FileOutputStream f = new FileOutputStream("nomFichier.dat");`
va associer à un objet f un fichier de nom nomFichier.dat.
Si ce fichier n'existe pas, il est créé "vide", sinon le contenu
de son ancienne version est détruit.

Ecriture de données séquentiellement

Objectif : écrire un programme Java qui enregistre dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin ?

- **De la classe** `FileOutputStream` pour manipuler un flux binaire associé à un fichier en écriture : c'est une classe dérivée de la classe abstraite `OutputStream` qui est la classe de base de tous les flux binaires de sortie.
- D'un constructeur de la classe `FileOutputStream` : `FileOutputStream("nomFichier.dat")`.

`FileOutputStream f = new FileOutputStream("nomFichier.dat");`
va associer à un objet f un fichier de nom nomFichier.dat.
Si ce fichier n'existe pas, il est créé "vide", sinon le contenu
de son ancienne version est détruit.

Ecriture de données séquentiellement

Objectif : écrire un programme Java qui enregistre dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin ?

- **De la classe** `FileOutputStream` pour manipuler un flux binaire associé à un fichier en écriture : c'est une classe dérivée de la classe abstraite `OutputStream` qui est la classe de base de tous les flux binaires de sortie.

- **D'un constructeur de la classe** `FileOutputStream` : `FileOutputStream("nomFichier.dat")`.

`FileOutputStream f = new FileOutputStream("nomFichier.dat");` va associer à un objet `f` un fichier de nom `nomFichier.dat`.

Si ce fichier n'existe pas, il est créé "vide", sinon le contenu de son ancienne version est détruit.

Ecriture de données séquentiellement

Objectif : écrire un programme Java qui enregistre dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin ?

- **De la classe FileOutputStream pour manipuler un flux binaire associé à un fichier en écriture** : c'est une classe dérivée de la classe abstraite OutputStream qui est la classe de base de tous les flux binaires de sortie.

- **D'un constructeur de la classe FileOutputStream** : FileOutputStream("nomFichier.dat").

FileOutputStream f = new FileOutputStream("nomFichier.dat"); va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il est créé "vide", sinon le contenu de son ancienne version est détruit.

Ecriture de données séquentiellement

De quoi a-t-on besoin ? (suite)

- De la classe `DataOutputStream` pour envoyer sur un flux de sortie des valeurs d'un type primitif quelconque via des méthodes `writeInt`, `writeFloat` etc. : c'est une classe dérivée de la classe abstraite `OutputStream`.
La classe `DataOutputStream` dispose du constructeur : `DataOutputStream(FileOutputStream f)`.
- `DataOutputStream sortie = new DataOutputStream(f);`
- `DataOutputStream sortie = new DataOutputStream(new FileOutputStream("nomFichier.dat"));`

Ecriture de données séquentiellement

De quoi a-t-on besoin ? (suite)

- **De la classe DataOutputStream pour envoyer sur un flux de sortie des valeurs d'un type primitif quelconque via des méthodes writeInt, writeFloat etc.** : c'est une classe dérivée de la classe abstraite OutputStream.
La classe DataOutputStream dispose du constructeur : DataOutputStream(FileOutputStream f).
- DataOutputStream sortie = new DataOutputStream(f);
- DataOutputStream sortie = new DataOutputStream(new FileOutputStream("nomFichier.dat"));

Ecriture de données séquentiellement

De quoi a-t-on besoin ? (suite)

- **De la classe DataOutputStream pour envoyer sur un flux de sortie des valeurs d'un type primitif quelconque via des méthodes writeInt, writeFloat etc.** : c'est une classe dérivée de la classe abstraite OutputStream.

La classe DataOutputStream dispose du constructeur : DataOutputStream(FileOutputStream f).

- DataOutputStream sortie = new DataOutputStream(f);
- DataOutputStream sortie = new DataOutputStream(new FileOutputStream("nomFichier.dat"));

Ecriture de données séquentiellement

De quoi a-t-on besoin ? (suite)

- **De la classe DataOutputStream pour envoyer sur un flux de sortie des valeurs d'un type primitif quelconque via des méthodes writeInt, writeFloat etc.** : c'est une classe dérivée de la classe abstraite OutputStream.
La classe DataOutputStream dispose du constructeur : DataOutputStream(FileOutputStream f).
- DataOutputStream sortie = new DataOutputStream(f);
- DataOutputStream sortie = new DataOutputStream(new FileOutputStream("nomFichier.dat"));

Ecriture de données séquentiellement

De quoi a-t-on besoin ? (suite)

- **De la classe DataOutputStream pour envoyer sur un flux de sortie des valeurs d'un type primitif quelconque via des méthodes writeInt, writeFloat etc.** : c'est une classe dérivée de la classe abstraite OutputStream.
La classe DataOutputStream dispose du constructeur : DataOutputStream(FileOutputStream f).
- DataOutputStream sortie = new DataOutputStream(f);
- DataOutputStream sortie = new DataOutputStream(new FileOutputStream("nomFichier.dat"));

Ecriture de données séquentiellement

De quoi a-t-on besoin ? (suite)

- **De la classe DataOutputStream pour envoyer sur un flux de sortie des valeurs d'un type primitif quelconque via des méthodes writeInt, writeFloat etc.** : c'est une classe dérivée de la classe abstraite OutputStream.
La classe DataOutputStream dispose du constructeur : DataOutputStream(FileOutputStream f).
- DataOutputStream sortie = new DataOutputStream(f);
- DataOutputStream sortie = new DataOutputStream(new FileOutputStream("nomFichier.dat"));

Ecriture de données séquentiellement : Exemple

```
import java.io.* ; // pour les classes de flux
                  // java.io est le package des entrées/sorties
public class FilesUtils {

    public static void writeIntArray(String fileName, int tab []) throws IOException
    {
        DataOutputStream sortie = new DataOutputStream(
            new FileOutputStream( fileName )
        );
        if( tab != null )
        {
            for(int i=0; i<tab.length; i++)
            {
                sortie.writeInt( tab[i] ); // writeInt peut déclencher une exception explicite
                                         // de type IOException
            }
        }
        sortie.close() ; // ferme le flux, donc ensuite il ne sera plus possible
                         // d'écrire dans le fichier
    }
}
```

Ecriture de données séquentiellement : Remarques

- On peut écrire différents types de données, les uns à la suite des autres, par exemple avec un `writeInt` suivi d'un `writeFloat` etc., mais il faudra être capable de retrouver cet ordre au moment de la lecture du fichier.
- Il est possible d'avoir également des flux avec un tampon : les informations sont d'abord enregistrées dans le tampon, puis quand le tampon est plein (ou à la fermeture du flux), il est vidé dans le flux.
Ce tampon sert à minimiser les échanges entre le flux et la mémoire qui sont coûteux (accès disque coûteux).

Ecriture de données séquentiellement : Remarques

- On peut écrire différents types de données, les uns à la suite des autres, par exemple avec un `writeInt` suivi d'un `writeFloat` etc., mais **il faudra être capable de retrouver cet ordre au moment de la lecture** du fichier.
- Il est possible d'avoir également des flux avec un tampon : les informations sont d'abord enregistrées dans le tampon, puis quand le tampon est plein (ou à la fermeture du flux), il est vidé dans le flux.
Ce tampon sert à minimiser les échanges entre le flux et la mémoire qui sont coûteux (accès disque coûteux).

Ecriture de données séquentiellement : Remarques

- On peut écrire différents types de données, les uns à la suite des autres, par exemple avec un `writeInt` suivi d'un `writeFloat` etc., mais **il faudra être capable de retrouver cet ordre au moment de la lecture** du fichier.
- Il est possible d'avoir également des flux avec un tampon : les informations sont d'abord enregistrées dans le tampon, puis quand le tampon est plein (ou à la fermeture du flux), il est vidé dans le flux.
Ce tampon sert à minimiser les échanges entre le flux et la mémoire qui sont coûteux (accès disque coûteux).

Ecriture de données séquentiellement : Remarques

- On peut écrire différents types de données, les uns à la suite des autres, par exemple avec un `writeInt` suivi d'un `writeFloat` etc., mais **il faudra être capable de retrouver cet ordre au moment de la lecture** du fichier.
- Il est possible d'avoir également des flux avec un tampon : les informations sont d'abord enregistrées dans le tampon, puis quand le tampon est plein (ou à la fermeture du flux), il est vidé dans le flux.

Ce tampon sert à minimiser les échanges entre le flux et la mémoire qui sont coûteux (accès disque coûteux).

Ecriture de données séquentiellement : Remarques

- On peut écrire différents types de données, les uns à la suite des autres, par exemple avec un `writeInt` suivi d'un `writelnFloat` etc., mais **il faudra être capable de retrouver cet ordre au moment de la lecture** du fichier.
- Il est possible d'avoir également des flux avec un tampon : les informations sont d'abord enregistrées dans le tampon, puis quand le tampon est plein (ou à la fermeture du flux), il est vidé dans le flux.
Ce tampon sert à minimiser les échanges entre le flux et la mémoire qui sont coûteux (accès disque coûteux).

Ecriture de données séquentiellement : Remarques

- Si on écrit des chaînes de caractères dans un fichier (via `writeChars`), sans connaître leur longueur exacte, on ne saura pas les relire par la suite.

2 solutions : soit on fixe une taille pour toutes les chaînes d'un fichier (tableau de char de taille fixe, *les chaînes étant soit tronquées, soit complétées par des espaces*) ; soit on écrit la taille de la chaîne avant chaque chaîne.

Une autre solution sera d'utiliser la méthode `writeObject` (cf. section persistance des objets).

Ecriture de données séquentiellement : Remarques

- Si on écrit des chaînes de caractères dans un fichier (via `writeChars`), sans connaître leur longueur exacte, on ne saura pas les relire par la suite.

2 solutions : soit on fixe une taille pour toutes les chaînes d'un fichier (tableau de char de taille fixe, *les chaînes étant soit tronquées, soit complétées par des espaces*) ; *soit on écrit la taille de la chaîne avant chaque chaîne*.

Une autre solution sera d'utiliser la méthode `writeObject` (cf. section persistance des objets).

Ecriture de données séquentiellement : Remarques

- Si on écrit des chaînes de caractères dans un fichier (via `writeChars`), sans connaître leur longueur exacte, on ne saura pas les relire par la suite.

2 solutions : soit on fixe une taille pour toutes les chaînes d'un fichier (tableau de char de taille fixe, *les chaînes étant soit tronquées, soit complétées par des espaces*) ; **soit on écrit la taille de la chaîne avant chaque chaîne.**

Une autre solution sera d'utiliser la méthode `writeObject` (cf. section persistance des objets).

Ecriture de données séquentiellement : Remarques

- Si on écrit des chaînes de caractères dans un fichier (via `writeChars`), sans connaître leur longueur exacte, on ne saura pas les relire par la suite.

2 solutions : soit on fixe une taille pour toutes les chaînes d'un fichier (tableau de char de taille fixe, *les chaînes étant soit tronquées, soit complétées par des espaces*) ; **soit on écrit la taille de la chaîne avant chaque chaîne.**

Une autre solution sera d'utiliser la méthode `writeObject` (cf. section persistance des objets).

Lecture de donnée séquentiellement

Objectif : écrire un programme Java qui lit dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin par analogie avec l'écriture ?

- De la classe `FileInputStream` pour manipuler un flux binaire associé à un fichier en lecture : c'est une classe dérivée de la classe abstraite `InputStream` qui est la classe de base de tous les flux binaires d'entrée.
- D'un constructeur de la classe `FileInputStream` : `FileInputStream("nomFichier.dat")`.

`FileInputStream f = new FileInputStream("nomFichier.dat");` va associer à un objet `f` un fichier de nom `nomFichier.dat`. Si ce fichier n'existe pas, il aura une exception de déclenchée (`FileNotFoundException`).

Lecture de donnée séquentiellement

Objectif : écrire un programme Java qui lit dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin par analogie avec l'écriture ?

- **De la classe FileInputStream pour manipuler un flux binaire associé à un fichier en lecture** : c'est une classe dérivée de la classe abstraite InputStream qui est la classe de base de tous les flux binaires d'entrée.

- **D'un constructeur de la classe FileInputStream** : FileInputStream("nomFichier.dat").

FileInputStream f = new FileInputStream("nomFichier.dat"); va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il aura une exception de déclenchée (FileNotFoundException).

Lecture de donnée séquentiellement

Objectif : écrire un programme Java qui lit dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin par analogie avec l'écriture ?

- **De la classe FileInputStream pour manipuler un flux binaire associé à un fichier en lecture** : c'est une classe dérivée de la classe abstraite InputStream qui est la classe de base de tous les flux binaires d'entrée.

- **D'un constructeur de la classe FileInputStream** : FileInputStream("nomFichier.dat").

FileInputStream f = new FileInputStream("nomFichier.dat"); va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il aura une exception de déclenchée (FileNotFoundException).

Lecture de donnée séquentiellement

Objectif : écrire un programme Java qui lit dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin par analogie avec l'écriture ?

- **De la classe FileInputStream pour manipuler un flux binaire associé à un fichier en lecture** : c'est une classe dérivée de la classe abstraite InputStream qui est la classe de base de tous les flux binaires d'entrée.
- **D'un constructeur de la classe FileInputStream** : FileInputStream("nomFichier.dat").

FileInputStream f = new FileInputStream("nomFichier.dat"); va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il aura une exception de déclenchée (FileNotFoundException).

Lecture de donnée séquentiellement

Objectif : écrire un programme Java qui lit dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin par analogie avec l'écriture ?

- **De la classe FileInputStream pour manipuler un flux binaire associé à un fichier en lecture** : c'est une classe dérivée de la classe abstraite InputStream qui est la classe de base de tous les flux binaires d'entrée.

- **D'un constructeur de la classe FileInputStream** : FileInputStream("nomFichier.dat").

FileInputStream f = new FileInputStream("nomFichier.dat"); va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il aura une exception déclenchée (FileNotFoundException).

Lecture de donnée séquentiellement

Objectif : écrire un programme Java qui lit dans un fichier binaire des nombres entiers (de type int).

De quoi a-t-on besoin par analogie avec l'écriture ?

- **De la classe FileInputStream pour manipuler un flux binaire associé à un fichier en lecture** : c'est une classe dérivée de la classe abstraite InputStream qui est la classe de base de tous les flux binaires d'entrée.
- **D'un constructeur de la classe FileInputStream** : FileInputStream("nomFichier.dat").

FileInputStream f = new FileInputStream("nomFichier.dat"); va associer à un objet f un fichier de nom nomFichier.dat. Si ce fichier n'existe pas, il aura une exception de déclenchée (FileNotFoundException).

Lecture de donnée séquentiellement

De quoi a-t-on besoin ? (suite)

- De la classe `DataInputStream` pour fournir sur un flux d'entrée des valeurs d'un type primitif quelconque via des méthodes `readInt`, `readFloat` etc. : c'est une classe dérivée de la classe abstraite `InputStream`.
La classe `DataInputStream` dispose du constructeur : `DataInputStream(FileInputStream f)`.
- `DataInputStream entree = new DataInputStream(f);`
- `DataInputStream entree = new DataInputStream(new FileInputStream("nomFichier.dat"));`

Lecture de donnée séquentiellement

De quoi a-t-on besoin ? (suite)

- De la classe `DataInputStream` pour fournir sur un flux d'entrée des valeurs d'un type primitif quelconque via des méthodes `readInt`, `readFloat` etc. : c'est une classe dérivée de la classe abstraite `InputStream`.
La classe `DataInputStream` dispose du constructeur : `DataInputStream(FileInputStream f)`.
- `DataInputStream entree = new DataInputStream(f);`
- `DataInputStream entree = new DataInputStream(new FileInputStream("nomFichier.dat"));`

Lecture de donnée séquentiellement

De quoi a-t-on besoin ? (suite)

- De la classe `DataInputStream` pour fournir sur un flux d'entrée des valeurs d'un type primitif quelconque via des méthodes `readInt`, `readFloat` etc. : c'est une classe dérivée de la classe abstraite `InputStream`.

La classe `DataInputStream` dispose du constructeur : `DataInputStream(FileInputStream f)`.

- `DataInputStream entree = new DataInputStream(f);`
- `DataInputStream entree = new DataInputStream(new FileInputStream("nomFichier.dat"));`

Lecture de donnée séquentiellement

De quoi a-t-on besoin ? (suite)

- De la classe `DataInputStream` pour fournir sur un flux d'entrée des valeurs d'un type primitif quelconque via des méthodes `readInt`, `readFloat` etc. : c'est une classe dérivée de la classe abstraite `InputStream`.
La classe `DataInputStream` dispose du constructeur : `DataInputStream(FileInputStream f)`.
- `DataInputStream entree = new DataInputStream(f);`
- `DataInputStream entree = new DataInputStream(new FileInputStream("nomFichier.dat"));`

Lecture de donnée séquentiellement

De quoi a-t-on besoin ? (suite)

- De la classe `DataInputStream` pour fournir sur un flux d'entrée des valeurs d'un type primitif quelconque via des méthodes `readInt`, `readFloat` etc. : c'est une classe dérivée de la classe abstraite `InputStream`.
La classe `DataInputStream` dispose du constructeur : `DataInputStream(FileInputStream f)`.
- `DataInputStream entree = new DataInputStream(f);`
- `DataInputStream entree = new DataInputStream(new FileInputStream("nomFichier.dat"));`

Lecture de donnée séquentiellement

De quoi a-t-on besoin ? (suite)

- De la classe `DataInputStream` pour fournir sur un flux d'entrée des valeurs d'un type primitif quelconque via des méthodes `readInt`, `readFloat` etc. : c'est une classe dérivée de la classe abstraite `InputStream`.
La classe `DataInputStream` dispose du constructeur : `DataInputStream(FileInputStream f)`.
- `DataInputStream entree = new DataInputStream(f);`
- `DataInputStream entree = new DataInputStream(new FileInputStream("nomFichier.dat"));`

Lecture de données séquentiellement : Exemple

```
import java.io.* ; // pour les classes de flux
public class FilesUtils {
    public static int[] readIntArray(String fileName) throws IOException
    {
        DataInputStream entree = new DataInputStream(
                new FileInputStream( fileName ) );
        List<Integer> v = new ArrayList</*Integer*/>(); // tableau de taille variable!
        boolean endOfFile = false ;
        while(!endOfFile){
            int val=0 ; // init artificielle pour faire plaisir au compilateur
            try{
                val = entree.readInt() ; // la fin de fichier est traitée par une exception
            }
            catch(EOFException e){
                endOfFile = true ;
            }
            if(!endOfFile)
                v.add( new Integer( val ) ); // readInt peut déclencher une exception explicite
            } // de type IOException
        entree.close() ; // ferme le flux, donc ensuite il ne sera plus possible de lire
        int tab [] = new int[v.size()] ;
        for(int i=0; i<tab.length; i++)
            tab[i] = v.get(i) ; // get est très coûteuse pour une LinkedList, mais
        return tab ; // rapide pour une ArrayList
    }
}
```



Accès direct à un fichier binaire

Java dispose d'une classe d'accès direct (*random access* en anglais) en lecture et écriture aux différents octets d'un fichier binaire : `RandomAccessFile`.

- `RandomAccessFile entree = new RandomAccessFile("fichier.dat", "r");`
- `RandomAccessFile entreeSortie = new RandomAccessFile("fichier.dat", "rw");`

Accès direct à un fichier binaire

Java dispose d'une classe d'accès direct (*random access* en anglais) en lecture et écriture aux différents octets d'un fichier binaire : RandomAccessFile.

- RandomAccessFile entree = new RandomAccessFile("fichier.dat", "r");
- RandomAccessFile entreeSortie = new RandomAccessFile("fichier.dat", "rw");

Accès direct à un fichier binaire

Java dispose d'une classe d'accès direct (*random access* en anglais) en lecture et écriture aux différents octets d'un fichier binaire : RandomAccessFile.

- ➊ RandomAccessFile entree = new RandomAccessFile("fichier.dat", "r");
- ➋ RandomAccessFile entreeSortie = new RandomAccessFile("fichier.dat", "rw");

Accès direct à un fichier binaire

Java dispose d'une classe d'accès direct (*random access* en anglais) en lecture et écriture aux différents octets d'un fichier binaire : RandomAccessFile.

- RandomAccessFile entree = new RandomAccessFile("fichier.dat", "r");
- RandomAccessFile entreeSortie = new RandomAccessFile("fichier.dat", "rw");

Accès direct à un fichier binaire

RandomAccessFile dispose des fonctionnalités des classes DataInputStream et DataOutputStream, ainsi que d'une méthode de positionnement du pointeur de fichier :

- `readInt`, `readFloat` etc.
- `writeInt`, `writeFloat` etc.
- `public void seek(long posProchainOctet)`.

Accès direct à un fichier binaire

RandomAccessFile dispose des fonctionnalités des classes DataInputStream et DataOutputStream, ainsi que d'une méthode de positionnement du pointeur de fichier :

- `readInt`, `readFloat` etc.
- `writeInt`, `writeFloat` etc.
- `public void seek(long posProchainOctet)`.

Accès direct à un fichier binaire

RandomAccessFile dispose des fonctionnalités des classes DataInputStream et DataOutputStream, ainsi que d'une méthode de positionnement du pointeur de fichier :

- `readInt`, `readFloat` etc.
- `writeInt`, `writeFloat` etc.
- `public void seek(long posProchainOctet)`.

Lecture de données en accès direct : Exemple

```
import java.io.* ; // pour les classes de flux
public class FilesUtils {
    public static int readIntRandomAccess(String fileName, long pos) throws IOException
    {
        int val;
        // JDK 7.0: try( ressource à fermer )
        // les fichiers ouverts ici seront automatiquement libérés (close())
        // même en cas d'exception
        try ( RandomAccessFile entree = new RandomAccessFile( fileName, "r" ) ) {
            long nbOctetsInFile = entree.length() ;
            if( pos<0 || pos >= nbOctetsInFile - 3 || (pos%4!=0) )
            {
                throw new IOException("Mauvais usage de readIntRandomAccess." ) ;
            }
            entree.seek(pos) ;
            val = entree.readInt(); // après cette lecture le pointeur de fichier est
                                   // incrémenté de 4
        }
        return val ;
    }
}
```

Création d'un fichier texte

- La classe abstraite Writer sert de base à toutes les classes relatives à un flux texte de sortie.
- La classe concrète FileWriter, dérivée de OutputStreamWriter, permet de manipuler un flux texte associé à un fichier :
`FileWriter fw = new FileWriter("f.txt");` (ouverture classique en écriture).
- On peut aussi utiliser PrintWriter qui offre plus de fonctionnalités (print et println) que FileWriter et peut se construire à partir de ce dernier :
`PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"));`

Création d'un fichier texte

- La classe abstraite Writer sert de base à toutes les classes relatives à un flux texte de sortie.
- La classe concrète FileWriter, dérivée de OutputStreamWriter, permet de manipuler un flux texte associé à un fichier :

`FileWriter fw = new FileWriter("f.txt");` (ouverture classique en écriture).

- On peut aussi utiliser PrintWriter qui offre plus de fonctionnalités (print et println) que FileWriter et peut se construire à partir de ce dernier :

`PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"));`

Création d'un fichier texte

- La classe abstraite Writer sert de base à toutes les classes relatives à un flux texte de sortie.
- La classe concrète FileWriter, dérivée de OutputStreamWriter, permet de manipuler un flux texte associé à un fichier :
FileWriter fw = new FileWriter("f.txt"); (ouverture classique en écriture).
- On peut aussi utiliser PrintWriter qui offre plus de fonctionnalités (print et println) que FileWriter et peut se construire à partir de ce dernier :
PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"));

Création d'un fichier texte

- La classe abstraite Writer sert de base à toutes les classes relatives à un flux texte de sortie.
- La classe concrète FileWriter, dérivée de OutputStreamWriter, permet de manipuler un flux texte associé à un fichier :
`FileWriter fw = new FileWriter("f.txt");` (ouverture classique en écriture).
- On peut aussi utiliser PrintWriter qui offre plus de fonctionnalités (print et println) que FileWriter et peut se construire à partir de ce dernier :
`PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"));`

Les flux texte

Création d'un fichier texte : Exemple

```
import java.io.* ; // pour les classes de flux

public class FilesUtils {

    public static void writeIntArrayTextFile(String fileName, int tab []) throws IOException
    {
        try (PrintWriter sortie = new PrintWriter( new FileWriter(fileName) )) {
            sortie.println("dim="+tab.length);
            for(int i=0; i< tab.length; i++)
                sortie.println("t["+i+"]="+tab[i]);
        }
    }
}
```

Ecriture de données séquentiellement : Remarques

- Comment choisir le format texte de sortie ?

```
PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"),  
        "UTF-8");
```

- Bonnes pratiques :

[http://www.javapractices.com/topic/TopicAction.do?
Id=42](http://www.javapractices.com/topic/TopicAction.do?Id=42)

- Java class Files :

[http://docs.oracle.com/javase/7/docs/api/java/nio/
file/Files.html](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html)

Ecriture de données séquentiellement : Remarques

- Comment choisir le format texte de sortie ?

```
PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"),  
        "UTF-8");
```

- Bonnes pratiques :

[http://www.javapractices.com/topic/TopicAction.do?
Id=42](http://www.javapractices.com/topic/TopicAction.do?Id=42)

- Java class Files :

[http://docs.oracle.com/javase/7/docs/api/java/nio/
file/Files.html](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html)

Ecriture de données séquentiellement : Remarques

- Comment choisir le format texte de sortie ?

```
PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"),  
        "UTF-8");
```

- Bonnes pratiques :

[http://www.javapractices.com/topic/TopicAction.do?
Id=42](http://www.javapractices.com/topic/TopicAction.do?Id=42)

- Java class Files :

[http://docs.oracle.com/javase/7/docs/api/java/nio/
file/Files.html](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html)

Ecriture de données séquentiellement : Remarques

- Comment choisir le format texte de sortie ?

```
PrintWriter sortie = new PrintWriter(new FileWriter("f.txt"),  
        "UTF-8");
```

- Bonnes pratiques :

[http://www.javapractices.com/topic/TopicAction.do?
Id=42](http://www.javapractices.com/topic/TopicAction.do?Id=42)

- Java class Files :

[http://docs.oracle.com/javase/7/docs/api/java/nio/
file/Files.html](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html)

Les flux texte

Lecture d'un fichier texte : Exemple

```
import java.io.* ; // pour les classes de flux
import java.util.StringTokenizer;
public class FilesUtils {
    public static int[] readIntArrayTextFile(String fileName) throws IOException
    {   int tab [] = null ;
        // il n'existe pas de classe symétrique à PrintWriter...
        try (BufferedReader entree = new BufferedReader( new FileReader(fileName) )) {
            String ligne = entree.readLine() ; // si fin de fichier readLine() renvoie null
            if( ligne != null )
            { // lecture de la taille
                StringTokenizer tok = new StringTokenizer(ligne, "=") ; // "=" carac séparateurs
                int nbt = tok.countTokens() ;
                if( nbt!= 2 ) throw new IOException("Format non reconnu.") ;
                tok.nextToken(); // récupérer la valeur du 1er token et passer au suivant
                tab = new int[ Integer.parseInt( tok.nextToken() ) ] ;
            }
            int index = 0 ;
            do
            {   ligne = entree.readLine() ;
                if( ligne != null ){
                    StringTokenizer tok = new StringTokenizer(ligne, "=") ;
                    int nbt = tok.countTokens() ;
                    if( nbt!= 2 ) throw new IOException("Format non reconnu.") ;
                    tok.nextToken();
                    tab[index++] = Integer.parseInt( tok.nextToken() ) ;
                }
            }while(ligne!=null) ;
        } return tab ; } }
```



La persistance des objets avec les flux d'objets

La sérialisation et désérialisation des objets

Java fournit par ce mécanisme une manière de rendre persistant les objets, et cela indépendamment du système d'exploitation :

on pourra sérialiser un objet sur un système et le transmettre à un autre système qui le désérialisera.

La sérialisation et désérialisation des objets

Java fournit par ce mécanisme une manière de rendre persistant les objets, et cela indépendamment du système d'exploitation :

on pourra sérialiser un objet sur un système et le transmettre à un autre système qui le désérialisera.

Les flux d'objects en pratique

- La classe `ObjectOutputStream` pour gérer un flux d'objets de sortie : en plus de méthodes pour écrire des types de base (`writeInt`, `writeFloat...`), elle possède la méthode `writeObject`.
- La classe `ObjectInputStream` pour gérer un flux d'objets d'entrée : en plus de méthodes pour lire des types de base (`readInt`, `readFloat...`), elle possède la méthode `readObject`.

Il faut autoriser la classe des objets à être manipulée via des flux. Pour ce faire il faut **implémenter l'interface** `Serializable` (simple marqueur, aucune méthode à implanter).

Les flux d'objects en pratique

- La classe `ObjectOutputStream` pour gérer un flux d'objets de sortie : en plus de méthodes pour écrire des types de base (`writeInt`, `writeFloat...`), elle possède la méthode `writeObject`.
- La classe `ObjectInputStream` pour gérer un flux d'objets d'entrée : en plus de méthodes pour lire des types de base (`readInt`, `readFloat...`), elle possède la méthode `readObject`.

Il faut autoriser la classe des objets à être manipulée via des flux. Pour ce faire il faut **implémenter l'interface** `Serializable` (simple marqueur, aucune méthode à implanter).

Les flux d'objects en pratique

- La classe `ObjectOutputStream` pour gérer un flux d'objets de sortie : en plus de méthodes pour écrire des types de base (`writeInt`, `writeFloat...`), elle possède la méthode `writeObject`.
- La classe `ObjectInputStream` pour gérer un flux d'objets d'entrée : en plus de méthodes pour lire des types de base (`readInt`, `readFloat...`), elle possède la méthode `readObject`.

Il faut autoriser la classe des objets à être manipulée via des flux. Pour ce faire il faut **implémenter l'interface** `Serializable` (simple marqueur, aucune méthode à implanter).

Les flux d'objects en pratique

- La classe `ObjectOutputStream` pour gérer un flux d'objets de sortie : en plus de méthodes pour écrire des types de base (`writeInt`, `writeFloat...`), elle possède la méthode `writeObject`.
- La classe `ObjectInputStream` pour gérer un flux d'objets d'entrée : en plus de méthodes pour lire des types de base (`readInt`, `readFloat...`), elle possède la méthode `readObject`.

Il faut autoriser la classe des objets à être manipulée via des flux. Pour ce faire il faut **implémenter l'interface** `Serializable` (simple marqueur, aucune méthode à implanter).

Les flux d'objects en pratique

- La classe `ObjectOutputStream` pour gérer un flux d'objets de sortie : en plus de méthodes pour écrire des types de base (`writeInt`, `writeFloat...`), elle possède la méthode `writeObject`.
- La classe `ObjectInputStream` pour gérer un flux d'objets d'entrée : en plus de méthodes pour lire des types de base (`readInt`, `readFloat...`), elle possède la méthode `readObject`.

Il faut autoriser la classe des objets à être manipulée via des flux. Pour ce faire il faut **implémenter l'interface** `Serializable` (simple marqueur, aucune méthode à implanter).

Les flux d'objects en pratique

- La classe ObjectOutputStream pour gérer un flux d'objets de sortie : en plus de méthodes pour écrire des types de base (writeInt, writeFloat...), elle possède la méthode **writeObject**.
- La classe ObjectInputStream pour gérer un flux d'objets d'entrée : en plus de méthodes pour lire des types de base (readInt, readFloat...), elle possède la méthode **readObject**.

Il faut autoriser la classe des objets à être manipulée via des flux. Pour ce faire il faut **implémenter l'interface** Serializable (simple marqueur, aucune méthode à implanter).

La persistance des objets avec les flux d'objets

Les flux d'objects en pratique

Si une classe a des attributs références sur un objet, alors le type de ces attributs doit lui aussi être Serializable.

Les classes Java prédéfinies ainsi que les types de base sont tous Serializable.

Attention : le flag Serializable ne s'hérite pas.

Les flux d'objects en pratique

Si une classe a des attributs références sur un objet, alors le type de ces attributs doit lui aussi être Serializable.

Les classes Java prédéfinies ainsi que les types de base sont tous Serializable.

Attention : le flag Serializable ne s'hérite pas.

Les flux d'objects en pratique

Si une classe a des attributs références sur un objet, alors le type de ces attributs doit lui aussi être Serializable.

Les classes Java prédéfinies ainsi que les types de base sont tous Serializable.

Attention : le flag Serializable ne s'hérite pas.

La persistance des objets avec les flux d'objets

Ecriture d'un fichier texte : Exemple

```
import java.io.* ; // pour les classes de flux
class Point2D implements Serializable{ // implements Serializable => autoriser la sérialisation
    private int x, y ;
    public Point2D(int abs, int ord){
        x = abs ; y = ord ;
    }
    @Override
    public boolean equals(Object obj) {
        if(this == obj) return true ;
        if(obj==null) return false ;
        if(getClass() != obj.getClass()) return false ;
        Point2D p = (Point2D) obj ;
        return x==p.x && y==p.y ;
    }
}

public class FilesUtils {
    public static void writePointArray(String fileName, Point2D tab []) throws IOException {
        try (ObjectOutputStream sortie = new ObjectOutputStream( new FileOutputStream(fileName)
)) {
            sortie.writeInt(tab.length);
            for(int i=0; i< tab.length; i++)
                sortie.writeObject(tab[i]);
        }
    }
}
```



La persistance des objets avec les flux d'objets

Lecture d'un fichier texte : Exemple

```
import java.io.* ; // pour les classes de flux
public class FilesUtils {
    public static Point2D [] readPointArray(String fileName) throws IOException
    {   Point2D tab [] = null ;
        try (ObjectInputStream entree = new ObjectInputStream( new FileInputStream(fileName) )) {
            try{ tab = new Point2D[entree.readInt()] ; }
            catch(Exception e){
                System.out.println("readPointArray: not able to find tab size. " );
                e.printStackTrace();
                System.exit(-1) ;
            }
            int index=0 ;   boolean endOfFile = false ;
            while(!endOfFile)
            {   try{
                    tab[index++] = (Point2D) entree.readObject() ;
                }
                catch(EOFException e){ endOfFile = true ; }
                catch(ClassNotFoundException e){ // cas ou la classe est inconnue
                    e.printStackTrace() ;
                    System.exit(-1) ;
                }
            }
        }
        return tab ;
    }
}
```



Plan

- 1 Les flux et les fichiers
- 2 La programmation générique
 - Classes génériques
 - Méthodes génériques
 - Allons plus loin
- 3 Les collections et les algorithmes
- 4 Java 8 : Expressions lambda, streams et l'API java.time

Introduction

Définition de générnicité : *algorithme et classe génériques*

En POO, la générnicité signifie manipuler/traiter des objets de types différents ***au sein d'un algorithme ou à l'aide d'une classe écrit une seule fois "pour tous les types".***

Exemple 1 : algorithme de tri générique : on écrit un algorithme de tri une seule fois et ensuite on est capable de trier des tableaux de int, de double, mais aussi des tableaux d'objets (si les objets sont comparables).

Exemple 2 : une classe Paire générique : on écrit un modèle de classe une seule fois et ensuite on est capable d'instancier des paires de int, de double ou d'objets.

Introduction

Définition de générnicité : *algorithme et classe génériques*

En POO, la générnicité signifie manipuler/traiter des objets de types différents ***au sein d'un algorithme ou à l'aide d'une classe écrit une seule fois "pour tous les types".***

Exemple 1 : algorithme de tri générique : on écrit un algorithme de tri une seule fois et ensuite on est capable de trier des tableaux de int, de double, mais aussi des tableaux d'objets (si les objets sont comparables).

Exemple 2 : une classe Paire générique : on écrit un modèle de classe une seule fois et ensuite on est capable d'instancier des paires de int, de double ou d'objets.

Introduction

Définition de générnicité : *algorithme et classe génériques*

En POO, la générnicité signifie manipuler/traiter des objets de types différents ***au sein d'un algorithme ou à l'aide d'une classe écrit une seule fois "pour tous les types".***

Exemple 1 : algorithme de tri générique : on écrit un algorithme de tri une seule fois et ensuite on est capable de trier des tableaux de int, de double, mais aussi des tableaux d'objets (si les objets sont comparables).

Exemple 2 : une classe Paire générique : on écrit un modèle de classe une seule fois et ensuite on est capable d'instancier des paires de int, de double ou d'objets.

Introduction

Définition de généricité : *algorithme et classe génériques*

En POO, la généricité signifie manipuler/traiter des objets de types différents ***au sein d'un algorithme ou à l'aide d'une classe écrit une seule fois "pour tous les types".***

Exemple 1 : algorithme de tri générique : on écrit un algorithme de tri une seule fois et ensuite on est capable de trier des tableaux de int, de double, mais aussi des tableaux d'objets (si les objets sont comparables).

Exemple 2 : une classe Paire générique : on écrit un modèle de classe une seule fois et ensuite on est capable d'instancier des paires de int, de double ou d'objets.

Introduction

Définition de générnicité : *algorithme et classe génériques*

En POO, la générnicité signifie manipuler/traiter des objets de types différents ***au sein d'un algorithme ou à l'aide d'une classe écrit une seule fois "pour tous les types".***

Exemple 1 : algorithme de tri générique : on écrit un algorithme de tri une seule fois et ensuite on est capable de trier des tableaux de int, de double, mais aussi des tableaux d'objets (si les objets sont comparables).

Exemple 2 : une classe Paire générique : on écrit un modèle de classe une seule fois et ensuite on est capable d'instancier des paires de int, de double ou d'objets.

Introduction

La générativité peut être mise en place par 2 approches différentes (mais complémentaires) :

- Grâce à l'héritage et au polymorphisme, le type "parent" étant manipulé, mais ces manipulations affectant un type concret enfant.
- Choisir le ou les types "paramétrables" au moment de l'instanciation d'un objet (depuis une *classe générique*) ou de l'appel à la méthode (méthode générique). **Disponible à partir du JDK 5.0.**

Introduction

La générativité peut être mise en place par 2 approches différentes (mais complémentaires) :

- **Grâce à l'héritage et au polymorphisme**, le type "parent" étant manipulé, mais ces manipulations affectant un type concret enfant.
- Choisir le ou les types "paramétrables" au moment de l'instanciation d'un objet (depuis une *classe générique*) ou de l'appel à la méthode (méthode générique). **Disponible à partir du JDK 5.0.**

Introduction

La générativité peut être mise en place par 2 approches différentes (mais complémentaires) :

- **Grâce à l'héritage et au polymorphisme**, le type "parent" étant manipulé, mais ces manipulations affectant un type concret enfant.
- **Choisir le ou les types "paramétrables" au moment de l'instanciation d'un objet** (depuis une *classe générique*) ou de l'appel à la méthode (méthode générique). **Disponible à partir du JDK 5.0.**

Introduction

La générativité peut être mise en place par 2 approches différentes (mais complémentaires) :

- **Grâce à l'héritage et au polymorphisme**, le type "parent" étant manipulé, mais ces manipulations affectant un type concret enfant.
- **Choisir le ou les types "paramétrables" au moment de l'instanciation d'un objet** (depuis une *classe générative*) ou de l'appel à la méthode (méthode générative). **Disponible à partir du JDK 5.0.**

Classes génériques

Exemple de classe générique

```
public class Pair<T> { // T est un paramètre type
    // T sera spécifié à la construction/instanciation
    // d'un object (T doit obligatoirement être une classe)
    private T first, second ;

    public Pair(T f, T s){
        first = f ; second = s ;
    }

    public String toString(){
        return "("+first+", "+second+")" ;
    }

    public T getFirst(){ return first ; }
    public T getSecond(){ return second ; }

    public void setFirst(T newF){ first = newF ; }
    public void setSecond(T newS){ second = newS ; }
}
```

Classes génériques

Utilisation de la classe générique précédente

```
public class TestPair {  
    public static void main(String [] args)  
    {  
        // on a besoin de spécifier le paramètre type  
        // pour déclarer la référence et pour utiliser  
        // le constructeur. Pour les autres méthodes rien  
        // ne change.  
        Pair< Integer > p = new Pair< Integer >( new Integer(5), new Integer(3) ) ;  
        // on pouvait utiliser directement 5 et 3  
        System.out.println(p) ;  
  
        p.setFirst(78) ; // 78 est converti automatiquement en Integer  
  
        p.setSecond(-45) ; // -45 est converti automatiquement en Integer  
  
        System.out.println(p) ;  
  
        Pair< int > p2 = new Pair< int >(5, 3) ; // ERREUR car int n'est pas une classe  
  
        Pair< Point2D > p3 = new Pair < Point2D >( new Point2D(4, 3), new Point2D(8, 7) ) ; // OK  
  
        Pair< Double > p4 = new Pair<>(2.1, -7.4) ;// <> est l'opérateur diamant (depuis JDK 7.0)  
    }  
}
```

Classes génériques

Classe générique à plusieurs paramètres

```
public class Pair<T, U> {  
    private T first ;  
    private U second ;  
  
    public Pair(T f, U s){  
        first = f ; second = s ;  
    }  
  
    public String toString(){  
        return "("+first+", "+second+")" ;  
    }  
  
    public T getFirst(){ return first ; }  
    public U getSecond(){ return second ; }  
  
    public void setFirst(T newF){ first = newF ; }  
    public void setSecond(U newS){ second = newS ; }  
}
```

Remarques

- Pour les classes génériques Java a opté pour le **mécanisme d'effacement** : *chaque type paramètre est remplacé par Object dans le bytecode compilé* ; lors de l'appel des méthodes, le compilateur mettra en place les conversions nécessaires en utilisant le type fourni à linstanciation.
Par exemple, pour `Pair< Integer > p = ...;`, `p.getFirst()` sera remplacé par `(Integer)p.getFirst()`.
- Il n'est donc pas possible d'instancier un type T dans une classe générique paramétrée par T : `new T(..)` est interdit.

Remarques

- Pour les classes génériques Java a opté pour le **mécanisme d'effacement** : *chaque type paramètre est remplacé par Object dans le bytecode compilé* ; lors de l'appel des méthodes, le compilateur mettra en place les conversions nécessaires en utilisant le type fourni à l'instanciation.
Par exemple, pour `Pair< Integer > p = ...;`, `p.getFirst()` sera remplacé par `(Integer)p.getFirst()`.
- Il n'est donc pas possible d'instancier un type T dans une classe générique paramétrée par T : `new T(..)` est interdit.

Remarques

- Pour les classes génériques Java a opté pour le **mécanisme d'effacement** : *chaque type paramètre est remplacé par Object dans le bytecode compilé* ; lors de l'appel des méthodes, le compilateur mettra en place les conversions nécessaires en utilisant le type fourni à l'instanciation.

Par exemple, pour `Pair< Integer > p = ...;`, `p.getFirst()` sera remplacé par `(Integer)p.getFirst()`.

- Il n'est donc pas possible d'instancier un type T dans une classe générique paramétrée par T : `new T(..)` est interdit.

Remarques

- Pour les classes génériques Java a opté pour le **mécanisme d'effacement** : *chaque type paramètre est remplacé par Object dans le bytecode compilé* ; lors de l'appel des méthodes, le compilateur mettra en place les conversions nécessaires en utilisant le type fourni à linstanciation.
Par exemple, pour `Pair< Integer > p = ... ;`, `p.getFirst()` sera remplacé par `(Integer)p.getFirst()`.
- Il n'est donc pas possible d'instancier un type T dans une classe générique paramétrée par T : `new T(..)` est interdit.

Remarques

- Pour les classes génériques Java a opté pour le **mécanisme d'effacement** : *chaque type paramètre est remplacé par Object dans le bytecode compilé* ; lors de l'appel des méthodes, le compilateur mettra en place les conversions nécessaires en utilisant le type fourni à linstanciation.
Par exemple, pour `Pair< Integer > p = ... ;`, `p.getFirst()` sera remplacé par `(Integer)p.getFirst()`.
- Il n'est donc pas possible d'instancier un type T dans une classe générique paramétrée par T : **new T(..) est interdit.**

Remarques

- Il n'est pas possible d'instancier un tableau de T dans une classe générique paramétrée par T : `new T[5]` sera rejeté.
- Il n'est pas possible d'instancier un tableau de références sur des instances particulières d'un type générique :
`Pair< Integer > tab [] = new Pair< Integer >[5];` sera rejeté à la compilation.
- `Pair< Integer > p = ...` et `Pair< Double > p2 = ...` : pour Java p et p2 seront du même type Pair.
- Un champ statique ne peut pas être d'un type paramtré.

Remarques

- Il n'est pas possible d'instancier un tableau de T dans une classe générique paramétrée par T : **new T[5] sera rejeté.**
- Il n'est pas possible d'instancier un tableau de références sur des instances particulières d'un type générique :
`Pair< Integer > tab [] = new Pair< Integer >[5];` sera rejeté à la compilation.
- `Pair< Integer > p = ...` et `Pair< Double > p2 = ...` : pour Java p et p2 seront du même type Pair.
- **Un champ statique ne peut pas être d'un type paramtré.**

Remarques

- Il n'est pas possible d'instancier un tableau de T dans une classe générique paramétrée par T : **new T[5] sera rejeté.**
- Il n'est pas possible d'instancier un tableau de références sur des instances particulières d'un type générique :
`Pair< Integer > tab [] = new Pair< Integer > [5];` sera rejeté à la compilation.
- `Pair< Integer > p = ...` et `Pair< Double > p2 = ...` : pour Java p et p2 seront du même type Pair.
- **Un champ statique ne peut pas être d'un type paramtré.**

Remarques

- Il n'est pas possible d'instancier un tableau de T dans une classe générique paramétrée par T : **new T[5] sera rejeté.**
- Il n'est pas possible d'instancier un tableau de références sur des instances particulières d'un type générique :
`Pair< Integer > tab [] = new Pair< Integer >[5];` sera rejeté à la compilation.
- `Pair< Integer > p = ...` et `Pair< Double > p2 = ...` : pour Java p et p2 seront du même type Pair.
- **Un champ statique ne peut pas être d'un type paramtré.**

Remarques

- Il n'est pas possible d'instancier un tableau de T dans une classe générique paramétrée par T : **new T[5] sera rejeté.**
- Il n'est pas possible d'instancier un tableau de références sur des instances particulières d'un type générique :
`Pair< Integer > tab [] = new Pair< Integer >[5];` sera rejeté à la compilation.
- `Pair< Integer > p = ...` et `Pair< Double > p2 = ...` : pour Java p et p2 seront du même type Pair.
- Un champ statique ne peut pas être d'un type paramtré.

Remarques

- Il n'est pas possible d'instancier un tableau de T dans une classe générique paramétrée par T : **new T[5] sera rejeté.**
- Il n'est pas possible d'instancier un tableau de références sur des instances particulières d'un type générique :
`Pair< Integer > tab [] = new Pair< Integer >[5];` sera rejeté à la compilation.
- `Pair< Integer > p = ...` et `Pair< Double > p2 = ...` : pour Java p et p2 seront du même type Pair.
- Un champ statique ne peut pas être d'un type paramétré.**

Remarques

- Il n'est pas possible d'instancier un tableau de T dans une classe générique paramétrée par T : **new T[5] sera rejeté.**
- Il n'est pas possible d'instancier un tableau de références sur des instances particulières d'un type générique :
`Pair< Integer > tab [] = new Pair< Integer >[5];` sera rejeté à la compilation.
- `Pair< Integer > p = ...` et `Pair< Double > p2 = ...` : pour Java p et p2 seront du même type Pair.
- **Un champ statique ne peut pas être d'un type paramétré.**

Exemple de méthode générique

La démarche d'introduire des paramètres de type dans une classe peut s'appliquer à une méthode.

```
public class MethodGen {  
  
    public static <T> T hasard(T [] valeurs) // la plupart des méthodes génériques seront  
    {                                         // statiques  
        if(valeurs==null) return null ;  
        int n = valeurs.length ;  
        if(n==0) return null ;  
        int i = (int)(n * Math.random()) ;  
        return valeurs[i] ;  
    }  
}
```

Méthodes génériques

Utilisation d'un méthode générique

```
public class MethodGenTest {  
  
    @Test  
    public void testHasard() {  
        assertEquals(null, MethodGen.hasard(null) ) ;  
  
        assertEquals(null, MethodGen.hasard(new Integer[0]) ) ;  
        assertEquals(null, MethodGen.hasard(new Double[0]) ) ;  
  
        Integer[] tabi = {1, 5, 8, 4, 9} ;  
        String [] tabs = { "bonjour", "salut", "hello" } ;  
  
        // le "bon" type est choisi automatiquement  
        for(int i=0; i<10; ++i) System.out.println( MethodGen.hasard (tabi) ) ;  
  
        for(int i=0; i<10; ++i) System.out.println( MethodGen.hasard (tabs) ) ;  
  
        // On peut aussi imposer le type de la méthode générique:  
        MethodGen.<Object>hasard (tabs); // type effectif Object  
        MethodGen.<Number>hasard (tabi); // type effectif Number  
        MethodGen.<String>hasard (tabs); // type effectif String  
        MethodGen.<Double>hasard (tabs); // type effectif Double => rejeté à la compilation  
    }  
}
```

Allons plus loin

Les paramètres de type peuvent être limités

On peut imposer qu'un paramètre de type hérite d'une classe et/ou qu'il implémente une ou plusieurs interfaces.

```
class MethodGen {  
    public static <T extends Comparable<T> > T max(T[] valeurs) { ... }  
  
    public static <T extends Number> T hasard(T [] valeurs){ ... }  
}  
public class MethodGenTest {  
  
    @Test  
    public void testHasard() {  
        ...  
        Integer[] tabi = {1, 5, 8, 4, 9} ;  
        String [] tabs = { "bonjour", "salut", "hello" } ;  
  
        System.out.println( MethodGen.hasard (tabi) ) ; // OK car Integer hérite de Number  
  
        System.out.println( MethodGen.hasard (tabs) ) ; // ERREUR car String n'hérite pas de  
                                                // Number  
    }  
}
```

<http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html>

Allons plus loin

Les paramètres de type peuvent être limités

On peut imposer qu'un paramètre de type hérite d'une classe et/ou qu'il implémente une ou plusieurs interfaces.

```
class MethodGen {  
    public static <T extends Comparable<T> > T max(T[] valeurs) { ... }  
  
    public static <T extends Number> T hasard(T [] valeurs){ ... }  
}  
public class MethodGenTest {  
  
    @Test  
    public void testHasard() {  
        ...  
        Integer[] tabi = {1, 5, 8, 4, 9} ;  
        String [] tabs = { "bonjour", "salut", "hello" } ;  
  
        System.out.println( MethodGen.hasard (tabi) ) ; // OK car Integer hérite de Number  
  
        System.out.println( MethodGen.hasard (tabs) ) ; // ERREUR car String n'hérite pas de  
                                                // Number  
    }  
}
```

<http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html>

Allons plus loin

Les paramètres de type peuvent être limités

On utilisera toujours extends, même pour une interface, et une seule classe pourra être utilisée (obligatoirement en 1ère position) tandis qu'aucune contrainte n'existe sur le nombre d'interfaces à implémenter.

```
// Number est une classe tandis que Comparable et Cloneable sont des interfaces
public class Pair<T extends Number & Comparable & Cloneable> {
    ...
}
```

Cette possibilité sera rapidement utilisée, car par exemple pour trier un tableau d'objets avec un algorithme générique, il faut que les objets contenus dans le tableau soient comparables...

[Allons plus loin](#)

Les paramètres de type peuvent être limités

On utilisera toujours extends, même pour une interface, et une seule classe pourra être utilisée (obligatoirement en 1ère position) tandis qu'aucune contrainte n'existe sur le nombre d'interfaces à implémenter.

```
// Number est une classe tandis que Comparable et Cloneable sont des interfaces
public class Pair<T extends Number & Comparable & Cloneable> {
    ...
}
```

Cette possibilité sera rapidement utilisée, car par exemple pour trier un tableau d'objets avec un algorithme générique, il faut que les objets contenus dans le tableau soient comparables...

Plan

1 Les flux et les fichiers

2 La programmation générique

3 Les collections et les algorithmes

- Les structures de données à connaître
- Propriétés générales des collections
- Les itérateurs pour parcourir les éléments
- Fonctionnalités communes des collections
- Les List
- Les Set
- Les Map
- Les algorithmes
- Les bonnes pratiques

Le chapitre sur les collections et les algorithmes du livre "Programmer en Java" est disponible

http://www.eyrolles.com/Chapitres/9782212122329/Chap22_Delannoy.pdf

Les Collections en Java ça consiste en quoi ?

Les collections `java.util.Collection` rassemblent des *classes* et des *interfaces* qui implémentent les **structures de données les plus courantes** (tableaux dynamiques, listes chaînées, les arbres et les tables de hachage).

Les *algorithmes* de recherche d'un élément ou de tri d'une collection sont également fournis.

On peut parcourir les éléments d'une collection grâce à des *itérateurs*.

Les Collections en Java ça consiste en quoi ?

Les collections `java.util.Collection` rassemblent des *classes* et des *interfaces* qui implémentent les **structures de données les plus courantes** (tableaux dynamiques, listes chaînées, les arbres et les tables de hachage).

Les *algorithmes* de recherche d'un élément ou de tri d'une collection sont également fournis.

On peut parcourir les éléments d'une collection grâce à des *itérateurs*.

Les Collections en Java ça consiste en quoi ?

Les collections `java.util.Collection` rassemblent des *classes* et des *interfaces* qui implémentent les **structures de données les plus courantes** (tableaux dynamiques, listes chaînées, les arbres et les tables de hachage).

Les *algorithmes* de recherche d'un élément ou de tri d'une collection sont également fournis.

On peut parcourir les éléments d'une collection grâce à des *itérateurs*.

Les Collections en Java ça consiste en quoi ?

L'équivalent des Collections Java en C++ est la STL (*Standard Template Library*).

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; **une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments** (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; **une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments** (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables.*
- ➎ etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; **une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments** (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables.*
etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; **une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments** (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.

etc.

Les conteneurs dynamiques d'éléments de même type

- ➊ **Les tableaux ou vecteurs dynamiques** : leur taille peut évoluer au cours du temps.
- ➋ **Les listes chaînées** : doublement chaînée.
- ➌ **Les tableaux associatifs** : c'est une structure de données basée soit sur un *arbre binaire de recherche*, soit sur une *table de hachage* ; **une clé permet d'accéder à un élément, et c'est cette même clé qui permet d'organiser les éléments** (en fait des paires clé+valeur).
- ➍ **Les ensembles** : c'est une version particulière du tableau associatif dans laquelle la clé et la valeur sont confondues ; *les éléments d'un ensemble doivent donc être comparables*.
- ➎ etc.

En Java ces structures sont des collections

- Depuis le JDK 5.0, les collections sont manipulées par le biais de classes génériques implémentant l'interface `Collection<E>` où E représente le type des éléments de la collection. **Tous les éléments d'une même collection sont de même type E** (ou dérivé de E).
- Ex : Une liste `ArrayList<String>` ne pourra pas contenir d'objet de type autre que String ou dérivé de String.
- Ajouter ou supprimer un élément à une collection particulière revient à ajouter ou supprimer la référence sur l'objet en question (pas de copie profonde de l'objet).

En Java ces structures sont des collections

- Depuis le JDK 5.0, les collections sont manipulées par le biais de classes génériques implémentant l'interface `Collection<E>` où E représente le type des éléments de la collection. **Tous les éléments d'une même collection sont de même type E** (ou dérivé de E).
- Ex : Une liste `ArrayList<String>` ne pourra pas contenir d'objet de type autre que String ou dérivé de String.
- Ajouter ou supprimer un élément à une collection particulière revient à ajouter ou supprimer la référence sur l'objet en question (pas de copie profonde de l'objet).

En Java ces structures sont des collections

- Depuis le JDK 5.0, les collections sont manipulées par le biais de classes génériques implémentant l'interface `Collection<E>` où E représente le type des éléments de la collection. **Tous les éléments d'une même collection sont de même type E** (ou dérivé de E).
- Ex : Une liste `ArrayList<String>` ne pourra pas contenir d'objet de type autre que String ou dérivé de String.
- Ajouter ou supprimer un élément à une collection particulière revient à ajouter ou supprimer **la référence** sur l'objet en question (pas de copie profonde de l'objet).

Intérêt des collections ?

- **Remplacer les tableaux d'objets.**
- Des fonctionnalités en commun :
 - Parcourir les éléments de la collection.
 - Rechercher la présence (ou l'absence) d'un élément particulier dans la collection.
 - Ajouter et supprimer des éléments.
 - Et changer l'ordre de parcours des éléments.

Intérêt des collections ?

- **Remplacer les tableaux d'objets.**
- Des fonctionnalités en commun :
 - Parcourir les éléments de la collection.
 - Rechercher la présence (ou l'absence) d'un élément particulier dans la collection.
 - Ajouter et supprimer des éléments.
 - Et changer l'ordre de parcours des éléments.

Intérêt des collections ?

- **Remplacer les tableaux d'objets.**
- Des fonctionnalités en commun :
 - Parcourir les éléments de la collection.
 - Rechercher la présence (ou l'absence) d'un élément particulier dans la collection.
 - Ajouter et supprimer des éléments.
 - Et changer l'ordre de parcours des éléments.

Intérêt des collections ?

- **Remplacer les tableaux d'objets.**
- Des fonctionnalités en commun :
 - Parcourir les éléments de la collection.
 - Rechercher la présence (ou l'absence) d'un élément particulier dans la collection.
 - Ajouter et supprimer des éléments.
 - Et changer l'ordre de parcours des éléments.

Intérêt des collections ?

- **Remplacer les tableaux d'objets.**
- Des fonctionnalités en commun :
 - Parcourir les éléments de la collection.
 - Rechercher la présence (ou l'absence) d'un élément particulier dans la collection.
 - Ajouter et supprimer des éléments.
 - Et changer l'ordre de parcours des éléments.

Intérêt des collections ?

- **Remplacer les tableaux d'objets.**
- Des fonctionnalités en commun :
 - Parcourir les éléments de la collection.
 - Rechercher la présence (ou l'absence) d'un élément particulier dans la collection.
 - Ajouter et supprimer des éléments.
 - Et changer l'ordre de parcours des éléments.

Intérêt des collections ? Différentes structures de données et donc différentes performances.

On choisit la structure de données la plus adaptée à une problématique donnée (la + performante).

Un peu de lecture :

<http://bigocheatsheet.com/>

Intérêt des collections ? Différentes structures de données et donc différentes performances.

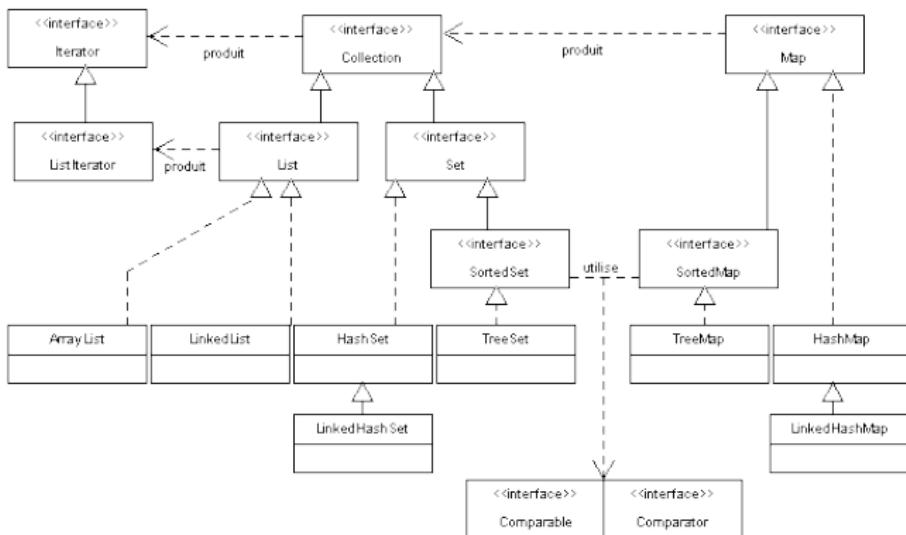
On choisit la structure de données la plus adaptée à une problématique donnée (la + performante).

Un peu de lecture :

<http://bigcheatsheet.com/>

Propriétés générales des collections

Hiérarchie des collections Java principales



<http://bioinfo.uqam.ca/inf7214/documents/semaine12/cours/Collections.html>

L'ordre des éléments

Toutes les collections Java sont munies d'un ordre (de parcours des éléments).

- **Les tableaux dynamiques et les listes chaînées conservent l'ordre d'insertion** : si un élément e est le i^{eme} élément ajouté à partir d'une collection vide, alors cet élément sera effectivement en i^{eme} position.
- Pour les autres collections l'ordre ne sera pas prévisible, il sera un ordre d'implantation.

L'ordre des éléments

Toutes les collections Java sont munies d'un ordre (de parcours des éléments).

- **Les tableaux dynamiques et les listes chaînées conservent l'ordre d'insertion** : si un élément e est le i^{eme} élément ajouté à partir d'une collection vide, alors cet élément sera effectivement en i^{eme} position.
- Pour les autres collections l'ordre ne sera pas prévisible, il sera un ordre d'implantation.

L'ordre des éléments

Toutes les collections Java sont munies d'un ordre (de parcours des éléments).

- **Les tableaux dynamiques et les listes chaînées conservent l'ordre d'insertion** : si un élément e est le i^{eme} élément ajouté à partir d'une collection vide, alors cet élément sera effectivement en i^{eme} position.
- Pour les autres collections l'ordre ne sera pas prévisible, il sera un ordre d'implantation.

La comparaison d'éléments au sein d'une collection

Pour la recherche de l'élément minimal, maximal ou pour trier une collection (ordre croissant ou ordre décroissant) :

- Les **tableaux dynamiques** et les **listes chaînées** supposeront que les éléments implémentent l'interface Comparable<E> et que la comparaison effective se réalise avec la méthode **public int compareTo(E o)** (seule méthode à implanter pour Comparable<E>).
- Les classes enveloppes (Integer, Double...) et la classe String implémentent l'interface Comparable<E>. Dans ce cas compareTo respecte l'ordre numérique pour les classes enveloppes et l'ordre lexicographique pour la classe String.

La comparaison d'éléments au sein d'une collection

Pour la recherche de l'élément minimal, maximal ou pour trier une collection (ordre croissant ou ordre décroissant) :

- Les **tableaux dynamiques** et les **listes chaînées** supposeront que les éléments implémentent l'interface Comparable<E> et que la comparaison effective se réalise avec la méthode **public int compareTo(E o)** (seule méthode à implanter pour Comparable<E>).
- Les classes enveloppes (Integer, Double...) et la classe String implémentent l'interface Comparable<E>.

Dans ce cas compareTo respecte l'ordre numérique pour les classes enveloppes et l'ordre lexicographique pour la classe String.

La comparaison d'éléments au sein d'une collection

Pour la recherche de l'élément minimal, maximal ou pour trier une collection (ordre croissant ou ordre décroissant) :

- Les **tableaux dynamiques** et les **listes chaînées** supposeront que les éléments implémentent l'interface Comparable<E> et que la comparaison effective se réalise avec la méthode **public int compareTo(E o)** (seule méthode à implanter pour Comparable<E>).
- Les classes enveloppes (Integer, Double...) et la classe String implémentent l'interface Comparable<E>. Dans ce cas compareTo respecte l'ordre numérique pour les classes enveloppes et l'ordre lexicographique pour la classe String.

Implémenter ou redéfinir la méthode compareTo

La méthode `compareTo` doit définir une relation d'ordre (pour pouvoir ordonner les éléments).

- Si l'objet courant est "inférieur" alors il faut retourner un entier négatif.
- Si l'objet courant est "égal" alors il faut retourner un entier nul.
- Si l'objet courant est "supérieur" alors il faut retourner un entier positif.
- **Relation d'ordre \leq** : Il faut garantir (tests unitaires) que
 - a $\leq o1$ (réflexivité).
 - si $o1 \leq o2$ et $o2 \leq o1$ alors $o1 = o2$ (antisymétrie).
 - si $o1 \leq o2$ et $o2 \leq o3$ alors $o1 \leq o3$ (transitivité).

Implémenter ou redéfinir la méthode compareTo

La méthode `compareTo` doit définir une relation d'ordre (pour pouvoir ordonner les éléments).

- Si l'objet courant est "inférieur" alors il faut retourner un entier négatif.
- Si l'objet courant est "égal" alors il faut retourner un entier nul.
- Si l'objet courant est "supérieur" alors il faut retourner un entier positif.
- **Relation d'ordre \leq** : Il faut garantir (tests unitaires) que
 - ➊ $o1 \leq o1$ (réflexivité).
 - ➋ si $o1 \leq o2$ et $o2 \leq o1$ alors $o1 = o2$ (antisymétrie).
 - ➌ si $o1 \leq o2$ et $o2 \leq o3$ alors $o1 \leq o3$ (transitivité).

Implémenter ou redéfinir la méthode compareTo

La méthode `compareTo` doit définir une relation d'ordre (pour pouvoir ordonner les éléments).

- Si l'objet courant est "inférieur" alors il faut retourner un entier négatif.
- Si l'objet courant est "égal" alors il faut retourner un entier nul.
- Si l'objet courant est "supérieur" alors il faut retourner un entier positif.
- **Relation d'ordre \leq** : Il faut garantir (tests unitaires) que
 - a $\leq o1$ (réflexivité).
 - si $o1 \leq o2$ et $o2 \leq o1$ alors $o1 = o2$ (antisymétrie).
 - si $o1 \leq o2$ et $o2 \leq o3$ alors $o1 \leq o3$ (transitivité).

Implémenter ou redéfinir la méthode compareTo

La méthode `compareTo` doit définir une relation d'ordre (pour pouvoir ordonner les éléments).

- Si l'objet courant est "inférieur" alors il faut retourner un entier négatif.
- Si l'objet courant est "égal" alors il faut retourner un entier nul.
- Si l'objet courant est "supérieur" alors il faut retourner un entier positif.
- **Relation d'ordre \leq** : Il faut garantir (tests unitaires) que
 - ➊ $o1 \leq o1$ (réflexivité).
 - ➋ si $o1 \leq o2$ et $o2 \leq o1$ alors $o1 = o2$ (antisymétrie).
 - ➌ si $o1 \leq o2$ et $o2 \leq o3$ alors $o1 \leq o3$ (transitivité).

Implémenter ou redéfinir la méthode compareTo

La méthode `compareTo` doit définir une relation d'ordre (pour pouvoir ordonner les éléments).

- Si l'objet courant est "inférieur" alors il faut retourner un entier négatif.
- Si l'objet courant est "égal" alors il faut retourner un entier nul.
- Si l'objet courant est "supérieur" alors il faut retourner un entier positif.
- **Relation d'ordre \leq** : Il faut garantir (tests unitaires) que
 - ➊ $o1 \leq o1$ (réflexivité).
 - ➋ si $o1 \leq o2$ et $o2 \leq o1$ alors $o1 = o2$ (antisymétrie).
 - ➌ si $o1 \leq o2$ et $o2 \leq o3$ alors $o1 \leq o3$ (transitivité).

Implémenter ou redéfinir la méthode compareTo

La méthode `compareTo` doit définir une relation d'ordre (pour pouvoir ordonner les éléments).

- Si l'objet courant est "inférieur" alors il faut retourner un entier négatif.
- Si l'objet courant est "égal" alors il faut retourner un entier nul.
- Si l'objet courant est "supérieur" alors il faut retourner un entier positif.
- **Relation d'ordre \leq** : Il faut garantir (tests unitaires) que
 - ➊ $o1 \leq o1$ (*réflexivité*).
 - ➋ si $o1 \leq o2$ et $o2 \leq o1$ alors $o1 = o2$ (*antisymétrie*).
 - ➌ si $o1 \leq o2$ et $o2 \leq o3$ alors $o1 \leq o3$ (*transitivité*).

Implémenter ou redéfinir la méthode compareTo

La méthode `compareTo` doit définir une relation d'ordre (pour pouvoir ordonner les éléments).

- Si l'objet courant est "inférieur" alors il faut retourner un entier négatif.
- Si l'objet courant est "égal" alors il faut retourner un entier nul.
- Si l'objet courant est "supérieur" alors il faut retourner un entier positif.
- **Relation d'ordre \leq** : Il faut garantir (tests unitaires) que
 - 1 $o1 \leq o1$ (*réflexivité*).
 - 2 si $o1 \leq o2$ et $o2 \leq o1$ alors $o1 = o2$ (*antisymétrie*).
 - 3 si $o1 \leq o2$ et $o2 \leq o3$ alors $o1 \leq o3$ (*transitivité*).

L'égalité au sein des collections

- **Toutes les collections supposent que l'on puisse tester l'égalité entre 2 éléments.** Cela est par exemple utile pour supprimer un élément d'une valeur donnée.
Il faudra donc redéfinir la méthode equals pour manipuler des collections de type perso, sauf pour les conteneurs basés sur un arbre binaire qui eux testent l'égalité avec la méthode compareTo (en implémentant l'interface Comparable<E>).
- Il est possible d'être amené à définir dans une même classe la méthode equals et la méthode compareTo. Dans ce cas il sera primordial de garantir que equals retourne true ssi compareTo retourne 0.

L'égalité au sein des collections

- **Toutes les collections supposent que l'on puisse tester l'égalité entre 2 éléments.** Cela est par exemple utile pour supprimer un élément d'une valeur donnée.
Il faudra donc redéfinir la méthode equals pour manipuler des collections de type perso, sauf pour les conteneurs basés sur un arbre binaire qui eux testent l'égalité avec la méthode compareTo (en implémentant l'interface Comparable<E>).
- Il est possible d'être amené à définir dans une même classe la méthode equals et la méthode compareTo. Dans ce cas il sera primordial de garantir que equals retourne true **ssi** compareTo retourne 0.

L'égalité au sein des collections

- **Toutes les collections supposent que l'on puisse tester l'égalité entre 2 éléments.** Cela est par exemple utile pour supprimer un élément d'une valeur donnée.
Il faudra donc **redéfinir la méthode equals** pour manipuler des collections de type perso, sauf pour les conteneurs basés sur un arbre binaire qui eux testent l'égalité avec la méthode compareTo (en implémentant l'interface Comparable<E>).
- Il est possible d'être amené à définir dans une même classe la méthode equals et la méthode compareTo. Dans ce cas il sera primordial de garantir que equals retourne true **ssi** compareTo retourne 0.

L'égalité au sein des collections

- **Toutes les collections supposent que l'on puisse tester l'égalité entre 2 éléments.** Cela est par exemple utile pour supprimer un élément d'une valeur donnée.
Il faudra donc redéfinir la méthode equals pour manipuler des collections de type perso, sauf pour les conteneurs basés sur un arbre binaire qui eux testent l'égalité avec la méthode compareTo (en implémentant l'interface Comparable<E>).
- Il est possible d'être amené à définir dans une même classe la méthode equals et la méthode compareTo. Dans ce cas il sera primordial de garantir que equals retourne true **ssi** compareTo retourne 0.

L'égalité au sein des collections

- **Toutes les collections supposent que l'on puisse tester l'égalité entre 2 éléments.** Cela est par exemple utile pour supprimer un élément d'une valeur donnée.
Il faudra donc **redéfinir la méthode equals pour manipuler des collections de type perso**, sauf pour les conteneurs basés sur un arbre binaire qui eux testent l'égalité avec la méthode compareTo (en implémentant l'interface Comparable<E>).
- **Il est possible d'être amené à définir dans une même classe la méthode equals et la méthode compareTo.** Dans ce cas il sera primordial de garantir que equals retourne true **ssi** compareTo retourne 0.

L'égalité au sein des collections

- **Toutes les collections supposent que l'on puisse tester l'égalité entre 2 éléments.** Cela est par exemple utile pour supprimer un élément d'une valeur donnée.
Il faudra donc **redéfinir la méthode equals** pour manipuler des collections de type perso, sauf pour les conteneurs basés sur un arbre binaire qui eux testent l'égalité avec la méthode compareTo (en implémentant l'interface Comparable<E>).
- **Il est possible d'être amené à définir dans une même classe la méthode equals et la méthode compareTo.** Dans ce cas il sera primordial de garantir que equals retourne true **ssi** compareTo retourne 0.

L'égalité au sein des collections

Les méthodes de Collections suivantes utilisent la méthode public boolean equals(Object object) définie pour le type des éléments contenus :

- boolean contains(Object o) (sauf les TreeSet).
- boolean remove(Object o).

Attention : pour un HashSet il faudra redéfinir en plus d>equals, hashCode.

L'égalité au sein des collections

Les méthodes de Collections suivantes utilisent la méthode `public boolean equals(Object object)` définie pour le type des éléments contenus :

- `boolean contains(Object o)` (sauf les `TreeSet`).
- `boolean remove(Object o)`.

Attention : pour un `HashSet` il faudra redéfinir en plus d'`equals`, `hashCode`.

L'égalité au sein des collections

Les méthodes de Collections suivantes utilisent la méthode public boolean equals(Object object) définie pour le type des éléments contenus :

- boolean contains(Object o) (sauf les TreeSet).
- boolean remove(Object o).

Attention : pour un HashSet il faudra redéfinir en plus d>equals, hashCode.

Les itérateurs pour parcourir les éléments d'une collection

Les types d'itérateur

- **Les itérateurs monodirectionnels** : le parcours se fait d'un début vers une fin, on ne passe qu'une seule fois sur un élément particulier de la collection.
- **Les itérateurs bidirectionnels** : le parcours peut se faire dans les 2 sens, on peut avancer dans un sens, puis reculer.

Les itérateurs pour parcourir les éléments d'une collection

Les types d'itérateur

- **Les itérateurs monodirectionnels** : le parcours se fait d'un début vers une fin, on ne passe qu'une seule fois sur un élément particulier de la collection.
- **Les itérateurs bidirectionnels** : le parcours peut se faire dans les 2 sens, on peut avancer dans un sens, puis reculer.

Les itérateurs pour parcourir les éléments d'une collection

Les types d'itérateur

- **Les itérateurs monodirectionnels** : le parcours se fait d'un début vers une fin, on ne passe qu'une seule fois sur un élément particulier de la collection.
- **Les itérateurs bidirectionnels** : le parcours peut se faire dans les 2 sens, on peut avancer dans un sens, puis reculer.

Les itérateurs pour parcourir les éléments d'une collection

Les types d'itérateur

- **Les itérateurs monodirectionnels** : le parcours se fait d'un début vers une fin, on ne passe qu'une seule fois sur un élément particulier de la collection.
- **Les itérateurs bidirectionnels** : le parcours peut se faire dans les 2 sens, on peut avancer dans un sens, puis reculer.

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs monodirectionnels : l'interface Iterator<E>

Toutes les collections Java (pas les Map<E>) disposent d'une méthode iterator fournissant un itérateur monodirectionnel, i.e. un objet implémentant l'interface Iterator<E> (depuis le JDK 5.0).

```
import java.util.LinkedList ;  
import java.util.Iterator ;  
...  
// Depuis JDK 5.0  
LinkedList<String> coll = new LinkedList<String>() ;  
coll.add("Hello") ;  
coll.add("Salut") ;  
  
Iterator<String> iter = coll.iterator() ; // Obtenir un itérateur monodirectionnel init  
// avec la 1ère position (début selon ordre)  
// si elle existe  
while( iter.hasNext() ) // Est-ce que la position courante dispose d'une position suivante?  
{  
    String o = iter.next() ; // Obtenir l'objet associé suivant et avancer  
    // l'itérateur d'une position  
    System.out.println(o) ; // utilisation de l'élément courant o  
}  
...
```

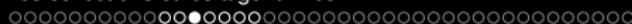
Les itérateurs pour parcourir les éléments d'une collection

Itérateurs monodirectionnels : l'interface Iterator<E>

Toutes les collections Java (pas les Map<E>) disposent d'une méthode iterator fournissant un itérateur monodirectionnel, i.e. un objet implémentant l'interface Iterator<E> (depuis le JDK 5.0).

```
import java.util.LinkedList ;
import java.util.Iterator ;
...
// Depuis JDK 5.0
LinkedList<String> coll = new LinkedList<String>() ;
coll.add("Hello") ;
coll.add("Salut") ;

Iterator<String> iter = coll.iterator() ; // Obtenir un itérateur monodirectionnel init
                                         // avec la 1ère position (début selon ordre)
                                         // si elle existe
while( iter.hasNext() ) // Est-ce que la position courante dispose d'une position suivante?
{
    String o = iter.next() ; // Obtenir l'objet associé suivant et avancer
                           // l'itérateur d'une position
    System.out.println(o) ; // utilisation de l'élément courant o
}
...
```



Les itérateurs pour parcourir les éléments d'une collection

Itérateurs monodirectionnels : faire 2 parcours

```
import java.util.LinkedList ;
import java.util.Iterator ;
...
// Depuis JDK 5.0
LinkedList<String> coll = new LinkedList<String>() ;
coll.add("Hello") ;
coll.add("Salut") ;

// Premier parcours
Iterator<String> iter = coll.iterator() ;
while( iter.hasNext() ){
    String o = iter.next() ;

    System.out.println(o) ;
}

// Deuxième parcours
iter = coll.iterator() ; // on rappelle la méthode iterator
while( iter.hasNext() ){
    String o = iter.next() ;

    System.out.println(o) ;
}
...
}
```



Les itérateurs pour parcourir les éléments d'une collection

Itérateurs monodirectionnels : la méthode remove

Jusqu'à présent nous avons parlé des méthodes `hasNext` et `next` de `Iterator<E>`. Il en existe une troisième, `remove`.

`remove` supprime de la collection le dernier objet renvoyé par `next`. Un appel à `remove` sans appel préalable à `next` engendrera une exception `IllegalStateException`.

```
import java.util.LinkedList ;
import java.util.Iterator ;

...
// Depuis JDK 5.0
LinkedList<String> coll = new LinkedList<String>();
coll.add("Hello") ;
coll.add("Salut") ;

// Premier parcours
Iterator<String> iter = coll.iterator() ;
while( iter.hasNext() ){
    String o = iter.next() ;

    if( o.equals("Salut") ) iter.remove() ; // supprime l'occurrence o de la collection
}
...
```

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs monodirectionnels : la méthode remove

Jusqu'à présent nous avons parlé des méthodes `hasNext` et `next` de `Iterator<E>`. Il en existe une troisième, `remove`.

remove supprime de la collection le dernier objet renvoyé par next. *Un appel à remove sans appel préalable à next engendrera une exception IllegalStateException.*

```
import java.util.LinkedList ;  
import java.util.Iterator ;  
  
...  
// Depuis JDK 5.0  
LinkedList<String> coll = new LinkedList<String>();  
coll.add("Hello") ;  
coll.add("Salut") ;  
  
// Premier parcours  
Iterator<String> iter = coll.iterator() ;  
while( iter.hasNext() ){  
    String o = iter.next() ;  
  
    if( o.equals("Salut") ) iter.remove() ; // supprime l'occurrence o de la collection  
}  
...
```

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs monodirectionnels : la méthode remove

Jusqu'à présent nous avons parlé des méthodes `hasNext` et `next` de `Iterator<E>`. Il en existe une troisième, `remove`.
remove supprime de la collection le dernier objet renvoyé par next. *Un appel à remove sans appel préalable à next engendrera une exception IllegalStateException.*

```
import java.util.LinkedList ;
import java.util.Iterator ;

...
// Depuis JDK 5.0
LinkedList<String> coll = new LinkedList<String>() ;
coll.add("Hello") ;
coll.add("Salut") ;

// Premier parcours
Iterator<String> iter = coll.iterator() ;
while( iter.hasNext() ){
    String o = iter.next() ;

    if( o.equals("Salut") ) iter.remove() ; // supprime l'occurrence o de la collection
}
...
```

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs bidirectionnels : l'interface ListIterator<E>

Les listes chaînées et les vecteurs dynamiques Java disposent d'une méthode listIterator fournissant un itérateur bidirectionnel, i.e. un objet implémentant l'interface ListIterator<E> (depuis le JDK 5.0), dérivée de Iterator<E>.

Les méthodes supplémentaires à hasNext, next et remove sont :

- hasPrevious, previous ;
- add entre la position précédente et la position courante de l'itérateur ; un appel à add déplace la position courante après l'élément que l'on vient d'ajouter ;
- set de l'élément courant (dernier élément obtenu via next ou previous).

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs bidirectionnels : l'interface ListIterator<E>

Les listes chaînées et les vecteurs dynamiques Java disposent d'une méthode listIterator fournissant un itérateur bidirectionnel, i.e. un objet implémentant l'interface ListIterator<E> (depuis le JDK 5.0), dérivée de Iterator<E>.

Les méthodes supplémentaires à hasNext, next et remove sont :

- hasPrevious, previous ;
- add entre la position précédente et la position courante de l'itérateur ; un appel à add déplace la position courante après l'élément que l'on vient d'ajouter ;
- set de l'élément courant (dernier élément obtenu via next ou previous).

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs bidirectionnels : l'interface ListIterator<E>

Les listes chaînées et les vecteurs dynamiques Java disposent d'une méthode listIterator fournissant un itérateur bidirectionnel, i.e. un objet implémentant l'interface ListIterator<E> (depuis le JDK 5.0), dérivée de Iterator<E>.

Les méthodes supplémentaires à hasNext, next et remove sont :

- hasPrevious, previous ;
- add entre la position précédente et la position courante de l'itérateur ; un appel à add déplace la position courante après l'élément que l'on vient d'ajouter ;
- set de l'élément courant (dernier élément obtenu via next ou previous).

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs bidirectionnels : l'interface ListIterator<E>

Les listes chaînées et les vecteurs dynamiques Java disposent d'une méthode listIterator fournissant un itérateur bidirectionnel, i.e. un objet implémentant l'interface ListIterator<E> (depuis le JDK 5.0), dérivée de Iterator<E>.

Les méthodes supplémentaires à hasNext, next et remove sont :

- hasPrevious, previous ;
- add entre la position précédente et la position courante de l'itérateur ; un appel à add déplace la position courante après l'élément que l'on vient d'ajouter ;
- set de l'élément courant (dernier élément obtenu via next ou previous).

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs bidirectionnels : l'interface ListIterator<E>

Les listes chaînées et les vecteurs dynamiques Java disposent d'une méthode listIterator fournissant un itérateur bidirectionnel, i.e. un objet implémentant l'interface ListIterator<E> (depuis le JDK 5.0), dérivée de Iterator<E>.

Les méthodes supplémentaires à hasNext, next et remove sont :

- hasPrevious, previous ;
- add entre la position précédente et la position courante de l'itérateur ; un appel à add déplace la position courante après l'élément que l'on vient d'ajouter ;
- set de l'élément courant (dernier élément obtenu via next ou previous).

Les itérateurs pour parcourir les éléments d'une collection

Itérateurs bidirectionnels : l'interface ListIterator<E>

Parcourir une liste à l'envers :

```
import java.util.LinkedList ;
import java.util.ListIterator ;

...
// Depuis JDK 5.0
LinkedList<String> coll = new LinkedList<String>() ;
coll.add("Hello") ;
coll.add("Salut") ;

ListIterator<String> iter = coll.listIterator(coll.size()) ; // coll.size()=la position de départ

while( iter.hasPrevious() )
{
    String o = iter.previous() ; // Obtenir l'objet associé à la position précédent
                                // la position courante et reculer l'itérateur d'une position
    System.out.println(o) ; // utilisation de l'élément courant o
}
...
```

Les itérateurs pour parcourir les éléments d'une collection

Comment ajouter un élément à une collection ne possédant qu'un itérateur monodirectionnel ?

En utilisant la méthode add de l'interface Collection<E>.

Attention : il ne faut jamais modifier une collection via une méthode de l'interface Collection<E> lorsqu'on est en train de parcourir la collection à l'aide d'un itérateur. En effet, dans ce cas là le comportement de l'itérateur est indéfini.

Les itérateurs pour parcourir les éléments d'une collection

Comment ajouter un élément à une collection ne possédant qu'un itérateur monodirectionnel ?

En utilisant la méthode add de l'interface Collection<E>.

Attention : il ne faut jamais modifier une collection via une méthode de l'interface Collection<E> lorsqu'on est en train de parcourir la collection à l'aide d'un itérateur. *En effet, dans ce cas là le comportement de l'itérateur est indéfini.*

Les itérateurs pour parcourir les éléments d'une collection

Comment ajouter un élément à une collection ne possédant qu'un itérateur monodirectionnel ?

En utilisant la méthode add de l'interface Collection<E>.

Attention : il ne faut jamais modifier une collection via une méthode de l'interface Collection<E> lorsqu'on est en train de parcourir la collection à l'aide d'un itérateur. **En effet, dans ce cas là le comportement de l'itérateur est indéfini.**

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- Constructeur sans argument ;
- Constructeur avec Collection< ? extends E > en argument ;
- Tout ce qui est en lien avec les itérateurs monodirectionnel : iterator, puis sur l'objet *iter* obtenu : hasNext, next et remove ;
- Toute collection dispose d'une méthode public boolean add(E element) ;
- Toute collection dispose d'une méthode public boolean remove(E element) ;
- Les opérations collectives : addAll(c), removeAll(c) et retainAll(c).

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- Constructeur sans argument ;
- Constructeur avec Collection< ? extends E > en argument ;
- Tout ce qui est en lien avec les itérateurs monodirectionnel : iterator, puis sur l'objet *iter* obtenu : hasNext, next et remove ;
- Toute collection dispose d'une méthode public boolean add(E element) ;
- Toute collection dispose d'une méthode public boolean remove(E element) ;
- Les opérations collectives : addAll(c), removeAll(c) et retainAll(c).

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- Constructeur sans argument ;
- Constructeur avec Collection< ? extends E > en argument ;
- Tout ce qui est en lien avec les itérateurs monodirectionnel : iterator, puis sur l'objet *iter* obtenu : hasNext, next et remove ;
- Toute collection dispose d'une méthode public boolean add(E element) ;
- Toute collection dispose d'une méthode public boolean remove(E element) ;
- Les opérations collectives : addAll(c), removeAll(c) et retainAll(c).

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- Constructeur sans argument ;
- Constructeur avec Collection< ? extends E > en argument ;
- Tout ce qui est en lien avec les itérateurs monodirectionnel : iterator, puis sur l'objet *iter* obtenu : hasNext, next et remove ;
- Toute collection dispose d'une méthode public boolean add(E element) ;
- Toute collection dispose d'une méthode public boolean remove(E element) ;
- Les opérations collectives : addAll(c), removeAll(c) et retainAll(c).

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- Constructeur sans argument ;
- Constructeur avec Collection< ? extends E > en argument ;
- Tout ce qui est en lien avec les itérateurs monodirectionnel : iterator, puis sur l'objet *iter* obtenu : hasNext, next et remove ;
- Toute collection dispose d'une méthode public boolean add(E element) ;
- Toute collection dispose d'une méthode public boolean remove(E element) ;
- Les opérations collectives : addAll(c), removeAll(c) et retainAll(c).

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- Constructeur sans argument ;
- Constructeur avec Collection< ? extends E > en argument ;
- Tout ce qui est en lien avec les itérateurs monodirectionnel : iterator, puis sur l'objet *iter* obtenu : hasNext, next et remove ;
- Toute collection dispose d'une méthode public boolean add(E element) ;
- Toute collection dispose d'une méthode public boolean remove(E element) ;
- Les opérations collectives : addAll(c), removeAll(c) et retainAll(c).

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- `size` fournit la taille d'une collection ;
- `isEmpty` teste si une collection est vide ou pas ;
- `clear` vide une collection ;
- `contains(E elem)` permet de savoir si la collection contient un élément de valeur `elem` ;
- `toArray` pour créer un tableau usuel d'objets à partir de la collection.

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- `size` fournit la taille d'une collection ;
- `isEmpty` teste si une collection est vide ou pas ;
- `clear` vide une collection ;
- `contains(E elem)` permet de savoir si la collection contient un élément de valeur `elem` ;
- `toArray` pour créer un tableau usuel d'objets à partir de la collection.

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- size fournit la taille d'une collection ;
- isEmpty teste si une collection est vide ou pas ;
- clear vide une collection ;
- contains(E elem) permet de savoir si la collection contient un élément de valeur elem ;
- toArray pour créer un tableau usuel d'objets à partir de la collection.

Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- size fournit la taille d'une collection ;
- isEmpty teste si une collection est vide ou pas ;
- clear vide une collection ;
- contains(E elem) permet de savoir si la collection contient un élément de valeur elem ;
- toArray pour créer un tableau usuel d'objets à partir de la collection.

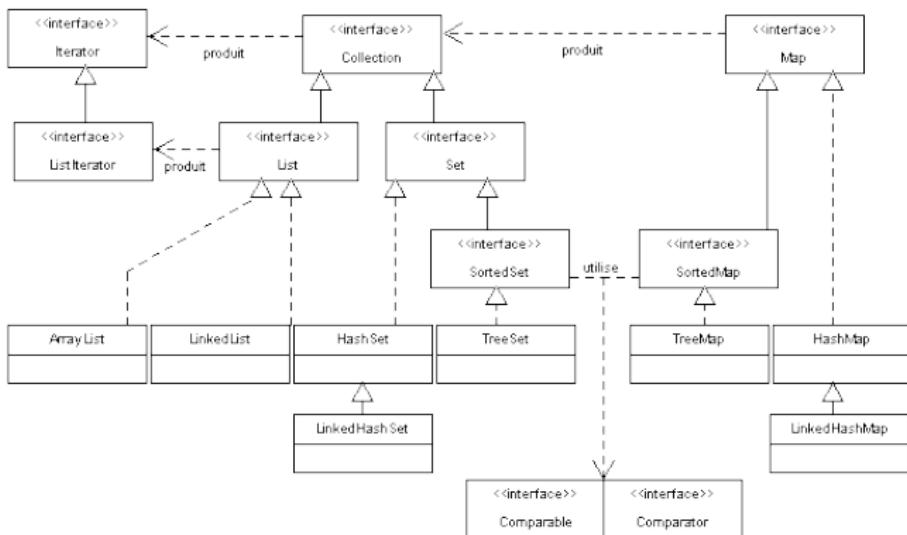
Bilan sur ce qu'on peut faire avec les collections Java

Les opérations communes à toutes les collections implémentant Collection<E>

- size fournit la taille d'une collection ;
- isEmpty teste si une collection est vide ou pas ;
- clear vide une collection ;
- contains(E elem) permet de savoir si la collection contient un élément de valeur elem ;
- toArray pour créer un tableau usuel d'objets à partir de la collection.

Bilan sur ce qu'on peut faire avec les collections Java

Hiérarchie des collections Java principales



<http://bioinfo.uqam.ca/inf7214/documents/semaine12/cours/Collections.html>

L'interface List<E>

L'interface List est implantée par les ArrayList et les LinkedList.

- Doublons possibles.
- Maintient l'ordre d'insertion.
- Les sous-listes ne sont que des vues.
- 2 listes sont égales si elles contiennent les mêmes éléments dans le même ordre.

L'interface List<E>

L'interface List est implantée par les ArrayList et les LinkedList.

- Doublons possibles.
- Maintient l'ordre d'insertion.
- Les sous-listes ne sont que des vues.
- 2 listes sont égales si elles contiennent les mêmes éléments dans le même ordre.

L'interface List<E>

L'interface List est implantée par les ArrayList et les LinkedList.

- Doublons possibles.
- Maintient l'ordre d'insertion.
- Les sous-listes ne sont que des vues.
- 2 listes sont égales si elles contiennent les mêmes éléments dans le même ordre.

L'interface List<E>

L'interface List est implantée par les ArrayList et les LinkedList.

- Doublons possibles.
- Maintient l'ordre d'insertion.
- Les sous-listes ne sont que des vues.
- 2 listes sont égales si elles contiennent les mêmes éléments dans le même ordre.

Les ArrayList

- Les ArrayList utilisent un tableau.
- L'intérêt principal des ArrayList réside dans les méthodes d'accès direct get(i) et set(i, val). L'ajout se fait par défaut en fin de tableau en temps constant.
- **Dans pratiquement tous les cas, ArrayList est un meilleur choix que LinkedList.** Les LinkedList sont meilleures uniquement pour l'insertion et la suppression à la position courante (même complexité si insertion ou suppression à une position i).

Les ArrayList

- Les ArrayList utilisent un tableau.
- L'intérêt principal des ArrayList réside dans les méthodes d'accès direct get(i) et set(i, val). L'ajout se fait par défaut en fin de tableau en temps constant.
- **Dans pratiquement tous les cas, ArrayList est un meilleur choix que LinkedList.** Les LinkedList sont meilleures uniquement pour l'insertion et la suppression à la position courante (même complexité si insertion ou suppression à une position i).

Les ArrayList

- Les ArrayList utilisent un tableau.
- L'intérêt principal des ArrayList réside dans les méthodes d'accès direct get(i) et set(i, val). L'ajout se fait par défaut en fin de tableau en temps constant.
- **Dans pratiquement tous les cas, ArrayList est un meilleur choix que LinkedList.** Les LinkedList sont meilleures uniquement pour l'insertion et la suppression à la position courante (même complexité si insertion ou suppression à une position i).

Les LinkedList

- Les LinkedList utilisent une liste doublement chaînée.

<http://docs.oracle.com/javase/tutorial/collections/implementations/list.html>

Les LinkedList

- Les LinkedList utilisent une liste doublement chaînée.

<http://docs.oracle.com/javase/tutorial/collections/implementlist.html>

L'interface Set<E>

L'interface Set est implantée par les HashSet et les TreeSet.

- **Doublons impossibles** au sein de la collection. Il faut se préoccuper des méthodes equals (HashSet) et compareTo (TreeSet).
- LinkedHashSet (dérivé de HashSet) maintient l'ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 ensembles sont égaux s'ils contiennent les mêmes éléments.
- **Les opérations ensemblistes** : e1.addAll(e2) (*union de e1 et e2*), e1.retainAll(e2) (*intersection de e1 et e2*) et e1.removeAll(e2) (*complémentaire de e2 par rapport à e1*).

L'interface Set<E>

L'interface Set est implantée par les HashSet et les TreeSet.

- **Doublons impossibles** au sein de la collection. Il faut se préoccuper des méthodes equals (HashSet) et compareTo (TreeSet).
- HashSet (dérivé de Collection) maintient l'ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 ensembles sont égaux s'ils contiennent les mêmes éléments.
- **Les opérations ensemblistes** : e1.addAll(e2) (*union de e1 et e2*), e1.retainAll(e2) (*intersection de e1 et e2*) et e1.removeAll(e2) (*complémentaire de e2 par rapport à e1*).

L'interface Set<E>

L'interface Set est implantée par les HashSet et les TreeSet.

- **Doublons impossibles** au sein de la collection. Il faut se préoccuper des méthodes equals (HashSet) et compareTo (TreeSet).
- HashSet (dérivé de List) maintient l'ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 ensembles sont égaux s'ils contiennent les mêmes éléments.
- **Les opérations ensemblistes** : e1.addAll(e2) (*union de e1 et e2*), e1.retainAll(e2) (*intersection de e1 et e2*) et e1.removeAll(e2) (*complémentaire de e2 par rapport à e1*).

L'interface Set<E>

L'interface Set est implantée par les HashSet et les TreeSet.

- **Doublons impossibles** au sein de la collection. Il faut se préoccuper des méthodes equals (HashSet) et compareTo (TreeSet).
- LinkedHashSet (dérivé de HashSet) maintient l'ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 ensembles sont égaux s'ils contiennent les mêmes éléments.
- **Les opérations ensemblistes** : e1.addAll(e2) (*union de e1 et e2*), e1.retainAll(e2) (*intersection de e1 et e2*) et e1.removeAll(e2) (*complémentaire de e2 par rapport à e1*).

L'interface Set<E>

L'interface Set est implantée par les HashSet et les TreeSet.

- **Doublons impossibles** au sein de la collection. Il faut se préoccuper des méthodes equals (HashSet) et compareTo (TreeSet).
- LinkedHashSet (dérivé de HashSet) maintient l'ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 ensembles sont égaux s'ils contiennent les mêmes éléments.
- **Les opérations ensemblistes** : e1.addAll(e2) (*union de e1 et e2*), e1.retainAll(e2) (*intersection de e1 et e2*) et e1.removeAll(e2) (*complémentaire de e2 par rapport à e1*).

Les HashSet

Se base sur **hashCode()** (*ordonnancer les éléments dans une table de hachage*) et **equals()** (*test d'appartenance*).

Les opérations (insertion, suppression et appartenance) ont une complexité O(1) sauf très mauvais hashCode().

Il faut définir hashCode() et equals() pour un type perso.

Attention : 2 objets égaux pour equals() doivent absolument fournir le même hashCode().

Les HashSet

Se base sur `hashCode()` (*ordonnancer les éléments dans une table de hachage*) et `equals()` (*test d'appartenance*).

Les opérations (insertion, suppression et appartenance) ont une complexité O(1) sauf très mauvais `hashCode()`.

Il faut définir `hashCode()` et `equals()` pour un type perso.

Attention : 2 objets égaux pour `equals()` doivent absolument fournir le même `hashCode()`.

Les HashSet

Se base sur `hashCode()` (*ordonnancer les éléments dans une table de hachage*) et `equals()` (*test d'appartenance*).

Les opérations (insertion, suppression et appartenance) ont une complexité O(1) sauf très mauvais `hashCode()`.

Il faut définir `hashCode()` et `equals()` pour un type perso.

Attention : 2 objets égaux pour `equals()` doivent absolument fournir le même `hashCode()`.

Les HashSet

Se base sur `hashCode()` (*ordonnancer les éléments dans une table de hachage*) et `equals()` (*test d'appartenance*).

Les opérations (insertion, suppression et appartenance) ont une complexité O(1) sauf très mauvais `hashCode()`.

Il faut définir `hashCode()` et `equals()` pour un type perso.

Attention : 2 objets égaux pour `equals()` doivent absolument fournir le même `hashCode()`.

Les LinkedHashSet

Idem HashSet mais retient en plus l'ordre d'insertion.

Les TreeSet

Ne se base pas sur hashCode() et equals() mais sur une comparaison : **compareTo()**. Les éléments sont automatiquement triés selon l'ordre de comparaison (*arbres binaires de recherche*).

Les opérations (insertion, suppression et appartenance) ont une complexité $O(\log N)$ (*donc mieux que pour les listes chaînées et les tableaux dynamiques pour lesquels ces opérations sont en $O(N)$*).

Il faut définir compareTo() pour un type perso.

Les TreeSet

Ne se base pas sur `hashCode()` et `equals()` mais sur une comparaison : `compareTo()`. Les éléments sont automatiquement triés selon l'ordre de comparaison (*arbres binaires de recherche*).

Les opérations (insertion, suppression et appartenance) ont une complexité $O(\log N)$ (*donc mieux que pour les listes chaînées et les tableaux dynamiques pour lesquels ces opérations sont en $O(N)$*).

Il faut définir `compareTo()` pour un type perso.

Les TreeSet

Ne se base pas sur `hashCode()` et `equals()` mais sur une comparaison : `compareTo()`. Les éléments sont automatiquement triés selon l'ordre de comparaison (*arbres binaires de recherche*).

Les opérations (insertion, suppression et appartenance) ont une complexité $O(\log N)$ (*donc mieux que pour les listes chaînées et les tableaux dynamiques pour lesquels ces opérations sont en $O(N)$*).

Il faut définir `compareTo()` pour un type perso.

HashSet ou TreeSet ?

HashSet est beaucoup plus rapide que TreeSet (temps-constant VS temps sous-linéaire).

TreeSet offre des garanties d'ensemble totalement ordonné (+un accès direct au min et au max de l'ensemble), ce que ne permet pas HashSet. Par contre cela suppose que l'on puisse comparer 2 objets (compareTo()), ce qui n'est pas toujours le cas.

HashSet ou TreeSet ?

HashSet est beaucoup plus rapide que TreeSet (temps-constant VS temps sous-linéaire).

TreeSet offre des garanties d'ensemble totalement ordonné (+un accès direct au min et au max de l'ensemble), ce que ne permet pas HashSet. Par contre cela suppose que l'on puisse comparer 2 objets (`compareTo()`), ce qui n'est pas toujours le cas.

HashSet ou TreeSet ?

HashSet est beaucoup plus rapide que TreeSet (temps-constant VS temps sous-linéaire).

TreeSet offre des garanties d'ensemble totalement ordonné (+un accès direct au min et au max de l'ensemble), ce que ne permet pas HashSet. Par contre cela suppose que l'on puisse comparer 2 objets (`compareTo()`), ce qui n'est pas toujours le cas.

HashSet ou TreeSet ?

Pour des objets de type perso, TreeSet impose de définir compareTo(), tandis que HashSet nécessite à la définition de hashCode() et equals(). Une bonne fonction de hachage, qui statistiquement repartisse bien les éléments, n'est pas toujours évidente à trouver.

Si vous ne connaissez pas la taille maximale de votre ensemble (ou si cette dernière varie brutalement et fréquemment), alors en termes de coût amorti le TreeSet est souvent meilleur (l'agrandissement d'une table de hachage est très coûteux).

<http://docs.oracle.com/javase/tutorial/collections/implementations/set.html>

HashSet ou TreeSet ?

Pour des objets de type perso, TreeSet impose de définir compareTo(), tandis que HashSet nécessite à la définition de hashCode() et equals(). **Une bonne fonction de hachage, qui statistiquement répartisse bien les éléments, n'est pas toujours évidente à trouver.**

Si vous ne connaissez pas la taille maximale de votre ensemble (ou si cette dernière varie brutalement et fréquemment), alors en termes de coût amorti le TreeSet est souvent meilleur (l'agrandissement d'une table de hachage est très coûteux).

<http://docs.oracle.com/javase/tutorial/collections/implementations/set.html>

HashSet ou TreeSet ?

Pour des objets de type perso, TreeSet impose de définir compareTo(), tandis que HashSet nécessite à la définition de hashCode() et equals(). **Une bonne fonction de hachage, qui statistiquement répartisse bien les éléments, n'est pas toujours évidente à trouver.**

Si vous ne connaissez pas la taille maximale de votre ensemble (ou si cette dernière varie brutalement et fréquemment), alors en termes de coût amorti le TreeSet est souvent meilleur (l'agrandissement d'une table de hachage est très coûteux).

<http://docs.oracle.com/javase/tutorial/collections/implementing/set.html>

HashSet ou TreeSet ?

Pour des objets de type perso, TreeSet impose de définir compareTo(), tandis que HashSet nécessite à la définition de hashCode() et equals(). **Une bonne fonction de hachage, qui statistiquement répartisse bien les éléments, n'est pas toujours évidente à trouver.**

Si vous ne connaissez pas la taille maximale de votre ensemble (ou si cette dernière varie brutalement et fréquemment), alors en termes de coût amorti le TreeSet est souvent meilleur (l'agrandissement d'une table de hachage est très coûteux).

<http://docs.oracle.com/javase/tutorial/collections/implementing/set.html>

L'interface Map<E>

Une **table associative** conserve des informations sous la forme d'une paire clé-valeur : *dictionnaire, annuaire, étudiants-notes.*

L'interface Map est implantée par les HashMap et les TreeMap.

- **Idem Set**, mais cette fois ça sera la clé qui sera utilisée pour les opérations (insertion, suppression, consultation) et pour l'ordre total des TreeMap. A chaque objet clé un seul objet valeur pourra être associé.
- Un Map n'hérite pas de Collection, car dans un Map on manipule des paires.
- LinkedHashMap maintient ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 tables associatives sont égales si elles contiennent les mêmes paires.

L'interface Map<E>

Une **table associative** conserve des informations sous la forme d'une paire clé-valeur : *dictionnaire*, *annuaire*, *étudiants-notes*.

L'interface Map est implantée par les `HashMap` et les `TreeMap`.

- **Idem Set**, mais cette fois ça sera la clé qui sera utilisée pour les opérations (insertion, suppression, consultation) et pour l'ordre total des `TreeMap`. A chaque objet clé un seul objet valeur pourra être associé.
- Un Map n'hérite pas de Collection, car dans un Map on manipule des paires.
- `LinkedHashMap` maintient ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 tables associatives sont égales si elles contiennent les mêmes paires.

L'interface Map<E>

Une **table associative** conserve des informations sous la forme d'une paire clé-valeur : *dictionnaire*, *annuaire*, *étudiants-notes*.

L'interface Map est implantée par les `HashMap` **et les** `TreeMap`.

- **Idem Set, mais cette fois ça sera la clé qui sera utilisée pour les opérations** (insertion, suppression, consultation) **et pour l'ordre total des TreeMap.** A chaque objet clé un seul objet valeur pourra être associé.
- Un Map n'hérite pas de Collection, car dans un Map on manipule des paires.
- `LinkedHashMap` maintient ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 tables associatives sont égales si elles contiennent les mêmes paires.

L'interface Map<E>

Une **table associative** conserve des informations sous la forme d'une paire clé-valeur : *dictionnaire*, *annuaire*, *étudiants-notes*.

L'interface Map est implantée par les `HashMap` **et les** `TreeMap`.

- **Idem Set, mais cette fois ça sera la clé qui sera utilisée pour les opérations (insertion, suppression, consultation) et pour l'ordre total des TreeMap. A chaque objet clé un seul objet valeur pourra être associé.**
- Un Map n'hérite pas de Collection, car dans un Map on manipule des paires.
- LinkedHashMap maintient ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 tables associatives sont égales si elles contiennent les mêmes paires.

L'interface Map<E>

Une **table associative** conserve des informations sous la forme d'une paire clé-valeur : *dictionnaire*, *annuaire*, *étudiants-notes*.

L'interface Map est implantée par les `HashMap` **et les** `TreeMap`.

- **Idem Set, mais cette fois ça sera la clé qui sera utilisée pour les opérations** (insertion, suppression, consultation) **et pour l'ordre total des TreeMap**. **A chaque objet clé un seul objet valeur pourra être associé.**
- Un Map n'hérite pas de Collection, car dans un Map on manipule des paires.
- `LinkedHashMap` maintient ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 tables associatives sont égales si elles contiennent les mêmes paires.

L'interface Map<E>

Une **table associative** conserve des informations sous la forme d'une paire clé-valeur : *dictionnaire*, *annuaire*, *étudiants-notes*.

L'interface Map est implantée par les `HashMap` **et les** `TreeMap`.

- **Idem Set, mais cette fois ça sera la clé qui sera utilisée pour les opérations** (insertion, suppression, consultation) **et pour l'ordre total des TreeMap**. **A chaque objet clé un seul objet valeur pourra être associé.**
- Un Map n'hérite pas de Collection, car dans un Map on manipule des paires.
- `LinkedHashMap` maintient ordre d'insertion, sinon pas de maintien de l'ordre d'insertion.
- **2 tables associatives sont égales si elles contiennent les mêmes paires.**

L'interface Map<E>

Une **table associative** conserve des informations sous la forme d'une paire clé-valeur : *dictionnaire*, *annuaire*, *étudiants-notes*.

L'interface Map est implantée par les `HashMap` **et les** `TreeMap`.

- **Idem Set, mais cette fois ça sera la clé qui sera utilisée pour les opérations** (insertion, suppression, consultation) **et pour l'ordre total des TreeMap**. **A chaque objet clé un seul objet valeur pourra être associé.**
- Un Map n'hérite pas de Collection, car dans un Map on manipule des paires.
- `LinkedHashMap` maintient ordre d'insertion, sinon pas de maintient de l'ordre d'insertion.
- 2 tables associatives sont égales si elles contiennent les mêmes paires.

L'ajout, la recherche et la suppression d'information

- **Ajout** : `m.put(keyObject, valueObject)`. Si la clé `keyObject` existait déjà, alors l'ancienne valeur associée est remplacée par `valueObject`. `put` retourne la valeur de l'ancien objet-valeur (null s'il n'existe pas).
- **Recherche** : `m.get(keyObject)`. Retourne la valeur associée à cette clé ou null si aucune valeur n'est associée à la clé `keyObject`.
- **Suppression** : `m.remove(keyObject)`. Retourne l'ancienne valeur associée à cette clé ou null si `keyObject` n'est pas présente dans la table.

L'ajout, la recherche et la suppression d'information

- **Ajout** : `m.put(keyObject, valueObject)`. Si la clé `keyObject` existait déjà, alors l'ancienne valeur associée est remplacée par `valueObject`. `put` retourne la valeur de l'ancien objet-valeur (null s'il n'existe pas).
- **Recherche** : `m.get(keyObject)`. Retourne la valeur associée à cette clé ou null si aucune valeur n'est associée à la clé `keyObject`.
- **Suppression** : `m.remove(keyObject)`. Retourne l'ancienne valeur associée à cette clé ou null si `keyObject` n'est pas présente dans la table.

L'ajout, la recherche et la suppression d'information

- **Ajout** : `m.put(keyObject, valueObject)`. Si la clé `keyObject` existait déjà, alors l'ancienne valeur associée est remplacée par `valueObject`. `put` retourne la valeur de l'ancien objet-valeur (null s'il n'existe pas).
- **Recherche** : `m.get(keyObject)`. Retourne la valeur associée à cette clé ou null si aucune valeur n'est associée à la clé `keyObject`.
- **Suppression** : `m.remove(keyObject)`. Retourne l'ancienne valeur associée à cette clé ou null si `keyObject` n'est pas présente dans la table.

L'ajout, la recherche et la suppression d'information

- **Ajout** : `m.put(keyObject, valueObject)`. Si la clé `keyObject` existait déjà, alors l'ancienne valeur associée est remplacée par `valueObject`. `put` retourne la valeur de l'ancien objet-valeur (null s'il n'existe pas).
- **Recherche** : `m.get(keyObject)`. Retourne la valeur associée à cette clé ou null si aucune valeur n'est associée à la clé `keyObject`.
- **Suppression** : `m.remove(keyObject)`. Retourne l'ancienne valeur associée à cette clé ou null si `keyObject` n'est pas présente dans la table.

L'ajout, la recherche et la suppression d'information

- **Ajout** : `m.put(keyObject, valueObject)`. Si la clé `keyObject` existait déjà, alors l'ancienne valeur associée est remplacée par `valueObject`. `put` retourne la valeur de l'ancien objet-valeur (null s'il n'existe pas).
- **Recherche** : `m.get(keyObject)`. Retourne la valeur associée à cette clé ou null si aucune valeur n'est associée à la clé `keyObject`.
- **Suppression** : `m.remove(keyObject)`. Retourne l'ancienne valeur associée à cette clé ou null si `keyObject` n'est pas présente dans la table.

L'ajout, la recherche et la suppression d'information

- **Ajout** : `m.put(keyObject, valueObject)`. Si la clé `keyObject` existait déjà, alors l'ancienne valeur associée est remplacée par `valueObject`. `put` retourne la valeur de l'ancien objet-valeur (null s'il n'existe pas).
- **Recherche** : `m.get(keyObject)`. Retourne la valeur associée à cette clé ou null si aucune valeur n'est associée à la clé `keyObject`.
- **Suppression** : `m.remove(keyObject)`. Retourne l'ancienne valeur associée à cette clé ou null si `keyObject` n'est pas présente dans la table.

Parcours une table associative

On doit passer par un ensemble de paires obtenu grâce à la méthode entrySet().

```
import java.util.Map;
import java.util.Set;
import java.util.Iterator ;

public class MethodGen {
    public static <K, V> void afficheMap(Map<K,V> m)
    {
        Set< Map.Entry<K, V> > entrees = m.entrySet() ;

        Iterator < Map.Entry<K, V> > iter = entrees.iterator() ;
        while( iter.hasNext() ){
            Map.Entry<K, V> entree = iter.next() ;

            K cle = entree.getKey() ;
            V valeur = entree.getValue() ;

            System.out.println("Clé = "+cle+" et valeur = "+valeur);
        }
    }
}
```

Les Map (les tables associatives)

Parcours une table associative

```
import java.util.Map;
import java.util.HashMap;

public class TestMethodGen {
    public static void main(String [] args)
    {
        Map< String, Double > notes = new HashMap< String, Double >() ;
        notes.put("Bob", 12.5) ;
        notes.put("Alfredo", 9.5) ;
        notes.put("Mireille", 17.0) ;

        MethodGen.afficheMap(notes) ;
    }
}
```

● Collections : méthodes utilitaires sur les collections :

- Recherche du minimum ou du maximum pour des objets comparables (ordre interne via compareTo ou alors définir un ordre externe via un Comparator, cf. page 655 du livre Prog. en Java).
- Pour des collections implémentant l'interface List : **Tri** (sort en anglais) pour des objets comparables et mélange aléatoire (*shuffle* en anglais).
- Arrays : méthodes utilitaires sur les tableaux (e.g. Arrays.asList(tab)).

- Collections : méthodes utilitaires sur les collections :
 - **Recherche du minimum ou du maximum** pour des objets comparables (ordre interne via compareTo ou alors définir un ordre externe via un Comparator, cf. page 655 du livre Prog. en Java).
 - Pour des collections implémentant l'interface List : **Tri** (sort en anglais) pour des objets comparables et mélange aléatoire (*shuffle* en anglais).
- Arrays : méthodes utilitaires sur les tableaux (e.g. Arrays.asList(tab)).

- Collections : méthodes utilitaires sur les collections :
 - **Recherche du minimum ou du maximum** pour des objets comparables (ordre interne via compareTo ou alors définir un ordre externe via un Comparator, cf. page 655 du livre Prog. en Java).
 - Pour des collections implémentant l'interface List : **Tri** (*sort* en anglais) pour des objets comparables **et mélange aléatoire** (*shuffle* en anglais).
- Arrays : méthodes utilitaires sur les tableaux (e.g. Arrays.asList(tab)).

- Collections : méthodes utilitaires sur les collections :
 - **Recherche du minimum ou du maximum** pour des objets comparables (ordre interne via compareTo ou alors définir un ordre externe via un Comparator, cf. page 655 du livre Prog. en Java).
 - Pour des collections implémentant l'interface List : **Tri** (*sort* en anglais) pour des objets comparables **et mélange aléatoire** (*shuffle* en anglais).
- Arrays : méthodes utilitaires sur les tableaux (e.g. Arrays.asList(tab)).

Méthodes à coder pour une classe

- `public boolean equals(Object o)` : pour les tests d'appartenance : `List` et `HashSet` ;
- `public int compareTo(E o)` : pour les comparaisons (min, max et tri) : `List` et `TreeSet` ;
- `public int hashCode()` : pour les tests d'appartenance au sein d'une table de hachage : `HashSet`.

Attention : pour un `TreeSet`, il est primordial que `equals` et `compareTo` soient cohérentes.

Méthodes à coder pour une classe

- public boolean equals(Object o) : pour les tests d'appartenance : List et HashSet ;
- public int compareTo(E o) : pour les comparaisons (min, max et tri) : List et TreeSet ;
- public int hashCode() : pour les tests d'appartenance au sein d'une table de hachage : HashSet.

Attention : pour un TreeSet, il est primordial que equals et compareTo soient cohérentes.

Méthodes à coder pour une classe

- public boolean equals(Object o) : pour les tests d'appartenance : List et HashSet ;
- public int compareTo(E o) : pour les comparaisons (min, max et tri) : List et TreeSet ;
- public int hashCode() : pour les tests d'appartenance au sein d'une table de hachage : HashSet.

Attention : pour un TreeSet, il est primordial que equals et compareTo soient cohérentes.

Méthodes à coder pour une classe

- public boolean equals(Object o) : pour les tests d'appartenance : List et HashSet ;
- public int compareTo(E o) : pour les comparaisons (min, max et tri) : List et TreeSet ;
- public int hashCode() : pour les tests d'appartenance au sein d'une table de hachage : HashSet.

Attention : pour un TreeSet, il est primordial que equals et compareTo soient cohérentes.

Puisque toutes les collections implémentent au moins une interface, **coder sur les interfaces**.

Eviter :

```
public HashMap<Long, User> methode(){  
    HashMap<Long, User> result = new HashMap<Long, User>();  
    ...  
    return result;  
}
```

Promouvoir :

```
public Map<Long, User> methode(){  
    Map<Long, User> result = new HashMap<Long, User>();  
    ...  
    return result;  
}
```

Puisque toutes les collections implémentent au moins une interface, **coder sur les interfaces**.

Eviter :

```
public HashMap<Long, User> methode(){
    HashMap<Long, User> result = new HashMap<Long, User>() ;
    ...
    return result ;
}
```

Promouvoir :

```
public Map<Long, User> methode(){
    Map<Long, User> result = new HashMap<Long, User>() ;
    ...
    return result ;
}
```

Préférer les collections aux tableaux.

- Plus clair ;
- Plus fonctionnel ;
- Plus sûr (concurrence possible etc.).
- On peut transformer un tableau en liste :

```
String[] array = ...;
```

```
List<String> fixedSizeList = Arrays.asList(array);
```

```
List<String> resizableList = new
```

```
ArrayList<>(Arrays.asList(array));
```

- On peut toujours revenir à un tableau si nécessaire :

```
List<String> list = ...;
```

```
String[] array = list.toArray(new String[list.size()]);
```

Préférer les collections aux tableaux.

- Plus clair ;
- Plus fonctionnel ;
- Plus sûr (concurrence possible etc.).
- On peut transformer un tableau en liste :

```
String[] array = ...;
```

```
List<String> fixedSizeList = Arrays.asList(array);
```

```
List<String> resizableList = new
```

```
ArrayList<>(Arrays.asList(array));
```

- On peut toujours revenir à un tableau si nécessaire :

```
List<String> list = ...;
```

```
String[] array = list.toArray(new String[list.size()]);
```

Préférer les collections aux tableaux.

- Plus clair ;
- Plus fonctionnel ;
- Plus sûr (concurrence possible etc.).
- On peut transformer un tableau en liste :

```
String[] array = ...;
List<String> fixedSizeList = Arrays.asList(array);
List<String> resizableList = new
ArrayList<>(Arrays.asList(array));
```
- On peut toujours revenir à un tableau si nécessaire :

```
List<String> list = ...;
String[] array = list.toArray(new String[list.size()]);
```

Préférer les collections aux tableaux.

- Plus clair ;
- Plus fonctionnel ;
- Plus sûr (concurrence possible etc.).
- On peut transformer un tableau en liste :

```
String[] array = ...;
List<String> fixedSizeList = Arrays.asList(array);
List<String> resizableList = new
ArrayList<>(Arrays.asList(array));
```
- On peut toujours revenir à un tableau si nécessaire :

```
List<String> list = ...;
String[] array = list.toArray(new String[list.size()]);
```

Préférer les collections aux tableaux.

- Plus clair ;
- Plus fonctionnel ;
- Plus sûr (concurrence possible etc.).
- On peut transformer un tableau en liste :

```
String[] array = ... ;
```

```
List<String> fixedSizeList = Arrays.asList(array) ;
```

```
List<String> resizableList = new
```

```
ArrayList<>(Arrays.asList(array)) ;
```

- On peut toujours revenir à un tableau si nécessaire :

```
List<String> list = ... ;
```

```
String[] array = list.toArray(new String[list.size()]) ;
```

Preférer les collections aux tableaux.

- Plus clair ;
- Plus fonctionnel ;
- Plus sûr (concurrence possible etc.).
- On peut transformer un tableau en liste :

```
String[] array = ... ;
```

```
List<String> fixedSizeList = Arrays.asList(array) ;
```

```
List<String> resizableList = new
```

```
ArrayList<>(Arrays.asList(array)) ;
```

- On peut toujours revenir à un tableau si nécessaire :

```
List<String> list = ... ;
```

```
String[] array = list.toArray(new String[list.size()]) ;
```

Toujours retourner une collection vide plutôt que null.

Collections.emptyList(), Collections.emptyMap(),
Collections.emptySet().

Toujours retourner une collection vide plutôt que null.

Collections.emptyList(), Collections.emptyMap(),
Collections.emptySet().

Pour garantir les invariants des objets, empêcher la modification directe des collections.

Collections.unmodifiableSet(Set),
Collections.unmodifiableMap(Map),
Collections.unmodifiableList(List)

Pour garantir les invariants des objets, empêcher la modification directe des collections.

Collections.unmodifiableSet(Set),
Collections.unmodifiableMap(Map),
Collections.unmodifiableList(List)

- apache-commons CompareToBuilder.
- Guava ComparisonChain.

- apache-commons CompareToBuilder.
- Guava ComparisonChain.

Implémenter un Comparator

```
/**  
 * Sorts tweets by author, then date descending, then subject, then ID  
 */  
public class TweetComparator implements Comparator<Tweet> {  
    @Override  
    public int compare(Tweet t1, Tweet t2) {  
        return ComparisonChain.start()  
            .compare(t1.getAuthor(), t2.getAuthor())  
            .compare(t1.getDate(), t2.getDate(), Ordering.natural().reverse())  
            .compare(t1.getSubject(), t2.getSubject())  
            .compare(t1.getId(), t2.getId())  
            .result();  
    }  
}
```

Implémenter une méthode hashCode

Vous pouvez vous aider des méthodes Objects.hashCode(int i),
Objects.hashCode(String s) etc.

A partir de Java 7, vous pouvez utiliser Objects.hash(attribut1,
attribut2, ..., attributn).

Exemple : <http://www.javaworld.com/article/2074417/core-java/guava-s-objects-class--equals--hashcode--and-tostring.html>.

Implémenter une méthode hashCode

Vous pouvez vous aider des méthodes Objects.hashCode(int i),
Objects.hashCode(String s) etc.

A partir de Java 7, vous pouvez utiliser Objects.hash(attribut1,
attribut2, ..., attributn).

Exemple : <http://www.javaworld.com/article/2074417/core-java/guava-s-objects-class--equals--hashcode--and-tostring.html>.

Implémenter une méthode hashCode

Vous pouvez vous aider des méthodes Objects.hashCode(int i),
Objects.hashCode(String s) etc.

A partir de Java 7, vous pouvez utiliser Objects.hash(attribut1,
attribut2, ..., attributn).

Exemple : <http://www.javaworld.com/article/2074417/core-java/guava-s-objects-class--equals--hashcode--and-tostring.html>.

Plan

- 1 Les flux et les fichiers
- 2 La programmation générique
- 3 Les collections et les algorithmes
- 4 Java 8 : Expressions lambda, streams et l'API java.time
 - Les expressions lambda
 - Les streams
 - API java.time

Définition

Expression lambda ou lambda

Une expression lambda est une méthode sans nom qui explicite un calcul/une expression à partir d'argument(s).

Syntaxes :

liste_d_arguments -> expression_utilisant_arguments

liste_d_arguments -> {suite_d_instructions}

Exemples : $x \rightarrow x * x$, $(x, y) \rightarrow x * y$,

$(\text{String first}, \text{String second}) \rightarrow \text{Integer.compare(first.length(), second.length())}$

Exemples

Le type des arguments n'est pas obligatoire si déductible :
`(first, second) -> Integer.compare(first.length(), second.length())`

Pas d'argument, "()" obligatoire :
`() -> System.out.println("Hello")`

Plusieurs instructions ou instruction terminée par un ';' :
`x -> { int i = 2*x ; return x - i ; }`

Pourquoi les lambdas ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Les expressions lambda sont convertibles/compatibles avec les *interfaces fonctionnelles* (e.g. Iterable, Comparable, Runnable mais aussi IntUnaryOperator...), plus besoin d'utiliser des classes anonymes.
Les **interfaces fonctionnelles** ont une seule méthode abstraite.
- Ces "morceaux" de code peuvent être exécutés plus tard ou en parallèle.

Le paquetage java.util.function contient beaucoup d'interfaces fonctionnelles (fonctions, prédictats...).

Pourquoi les lambdas ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Les expressions lambda sont convertibles/compatibles avec les *interfaces fonctionnelles* (e.g. Iterable, Comparable, Runnable mais aussi IntUnaryOperator...), plus besoin d'utiliser des classes anonymes.

Les *interfaces fonctionnelles* ont une seule méthode abstraite.

- Ces "morceaux" de code peuvent être exécutés plus tard ou en parallèle.

Le paquetage java.util.function contient beaucoup d'interfaces fonctionnelles (fonctions, prédictats...).

Pourquoi les lambdas ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Les expressions lambda sont convertibles/compatibles avec les *interfaces fonctionnelles* (e.g. Iterable, Comparable, Runnable mais aussi IntUnaryOperator...), plus besoin d'utiliser des classes anonymes.

Les **interfaces fonctionnelles** ont une seule méthode abstraite.

- Ces "morceaux" de code peuvent être exécutés plus tard ou en parallèle.

Le paquetage java.util.function contient beaucoup d'interfaces fonctionnelles (fonctions, prédictats...).

Pourquoi les lambdas ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Les expressions lambda sont convertibles/compatibles avec les *interfaces fonctionnelles* (e.g. Iterable, Comparable, Runnable mais aussi IntUnaryOperator...), plus besoin d'utiliser des classes anonymes.
Les **interfaces fonctionnelles** ont une seule méthode abstraite.
- Ces "morceaux" de code peuvent être exécutés plus tard ou en parallèle.

Le paquetage java.util.function contient beaucoup d'interfaces fonctionnelles (fonctions, prédictats...).

Pourquoi les lambdas ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Les expressions lambda sont convertibles/compatibles avec les *interfaces fonctionnelles* (e.g. Iterable, Comparable, Runnable mais aussi IntUnaryOperator...), plus besoin d'utiliser des classes anonymes.
Les **interfaces fonctionnelles** ont une seule méthode abstraite.
- Ces "morceaux" de code peuvent être exécutés plus tard ou en parallèle.

Le paquetage `java.util.function` contient beaucoup d'interfaces fonctionnelles (fonctions, prédictats...).

Interface fonctionnelle : exemple

```
public interface Calculateur {  
    public int calcul(int n) ;  
}  
  
...  
Calculateur c = x -> 2*x ;  
int i = c.calcul(7) ; // i vaut 14  
  
...  
Calculateur c1 = (x, y) -> x+y ; // erreur de compilation  
[pas bon nb d'arg]  
Calculateur c2 = x -> 2.5*x ; // erreur de compilation  
[type de retour pas compatible]  
Calculateur c3 = (float x) -> 2*x ; // erreur de  
compilation [type arg pas compatible]
```

Définition

Référence à une méthode

Une référence à une méthode est le nom d'une méthode existante que l'on va spécifier là où une *interface fonctionnelle* est attendue.

Syntaxes :

NomClasse: :nomMethode

nomObjet: :nomMethode

Intérêt ?

A chaque endroit où une expression lambda peut être utilisée, si une méthode existante a la bonne signature, alors on pourra l'utiliser.

Définition

Référence à une méthode

Une référence à une méthode est le nom d'une méthode existante que l'on va spécifier là où une *interface fonctionnelle* est attendue.

Syntaxes :

NomClasse: :nomMethode

nomObjet: :nomMethode

Intérêt ?

A chaque endroit où une expression lambda peut être utilisée, si une méthode existante a la bonne signature, alors on pourra l'utiliser.

Référence à une méthode : remarques

- Si une classe C dispose d'une méthode de classe mStatic, alors tous les arguments de la méthode sont explicites.
- Si une classe C dispose d'une méthode d'objet mInstance, alors il ne faut pas oublier que le premier argument est implicite, il est de type C (l'objet courant).
- La syntaxe NomClasse : :nomMethode est équivalente à l'expression lambda (other1, other2) -> other1.nomMethode(other) si nomMethode est une méthode d'objet et l'expression lambda (other1, other2) -> nomMethode(other1, other2) si nomMethode est une méthode de classe.
- La syntaxe nomObjet: :nomMethode est équivalente à l'expression lambda other -> nomObjet.nomMethode(other)

Référence à une méthode : remarques

- Si une classe C dispose d'une méthode de classe mStatic, alors tous les arguments de la méthode sont explicites.
- Si une classe C dispose d'une méthode d'objet mInstance, alors il ne faut pas oublier que le premier argument est implicite, il est de type C (l'objet courant).
- La syntaxe NomClasse : :nomMethode est équivalente à l'expression lambda (other1, other2) -> other1.nomMethode(other) si nomMethode est une méthode d'objet et l'expression lambda (other1, other2) -> nomMethode(other1, other2) si nomMethode est une méthode de classe.
- La syntaxe nomObjet: :nomMethode est équivalente à l'expression lambda other -> nomObjet.nomMethode(other)

Référence à une méthode : remarques

- Si une classe C dispose d'une méthode de classe mStatic, alors tous les arguments de la méthode sont explicites.
- Si une classe C dispose d'une méthode d'objet mInstance, alors il ne faut pas oublier que le premier argument est implicite, il est de type C (l'objet courant).
- La syntaxe NomClasse : :nomMethode est équivalente à l'expression lambda (other1, other2) -> other1.nomMethode(other) si nomMethode est une méthode d'objet et l'expression lambda (other1, other2) -> nomMethode(other1, other2) si nomMethode est une méthode de classe.
- La syntaxe nomObjet: :nomMethode est équivalente à l'expression lambda other -> nomObjet.nomMethode(other)

Référence à une méthode : remarques

- Si une classe C dispose d'une méthode de classe mStatic, alors tous les arguments de la méthode sont explicites.
- Si une classe C dispose d'une méthode d'objet mInstance, alors il ne faut pas oublier que le premier argument est implicite, il est de type C (l'objet courant).
- La syntaxe NomClasse : :nomMethode est équivalente à l'expression lambda (other1, other2) -> other1.nomMethode(other) si nomMethode est une méthode d'objet et l'expression lambda (other1, other2) -> nomMethode(other1, other2) si nomMethode est une méthode de classe.
- La syntaxe nomObjet: :nomMethode est équivalente à l'expression lambda other -> nomObjet.nomMethode(other)

Définition

Stream

Un stream est une séquence d'éléments issus d'une source qui est compatible avec les opérations d'agglomération filter, map, reduce, find, match, sorted.

Un stream ne stocke pas d'élément, mais il consomme une ressource à la demande.

Définition

Stream

Un stream est une séquence d'éléments issus d'une source qui est compatible avec les opérations d'agglomération filter, map, reduce, find, match, sorted.

Un stream ne stocke pas d'élément, mais il consomme une ressource à la demande.

Pourquoi les streams ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Pipeline de traitement.
- Traitement feignant.
- Transformer une Collection c en Stream séquentiel : c.stream() ; préserve l'ordre initial d'un conteneur séquentiel.
- Transformer une Collection c en Stream parallèle (automatisation du calcul parallèle transparente à l'utilisateur) : c.parallelStream() ; ne préserve pas l'ordre initial.

Pourquoi les streams ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Pipeline de traitement.
- Traitement feignant.
- Transformer une Collection c en Stream séquentiel : c.stream() ; préserve l'ordre initial d'un conteneur séquentiel.
- Transformer une Collection c en Stream parallèle (automatisation du calcul parallèle transparente à l'utilisateur) : c.parallelStream() ; ne préserve pas l'ordre initial.

Pourquoi les streams ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Pipeline de traitement.
- Traitement feignant.
- Transformer une Collection c en Stream séquentiel : c.stream() ; préserve l'ordre initial d'un conteneur séquentiel.
- Transformer une Collection c en Stream parallèle (automatisation du calcul parallèle transparente à l'utilisateur) : c.parallelStream() ; ne préserve pas l'ordre initial.

Pourquoi les streams ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Pipeline de traitement.
- Traitement feignant.
- Transformer une Collection c en Stream séquentiel : c.stream() ; préserve l'ordre initial d'un conteneur séquentiel.
- Transformer une Collection c en Stream parallèle (automatisation du calcul parallèle transparente à l'utilisateur) : c.parallelStream() ; ne préserve pas l'ordre initial.

Pourquoi les streams ?

Offrir des possibilités réservées habituellement aux langages de programmation fonctionnelle.

- Pipeline de traitement.
- Traitement feignant.
- Transformer une Collection c en Stream séquentiel : c.stream() ; préserve l'ordre initial d'un conteneur séquentiel.
- Transformer une Collection c en Stream parallèle (automatisation du calcul parallèle transparente à l'utilisateur) : c.parallelStream() ; ne préserve pas l'ordre initial.

Streams : opérations possibles

- **Filtrer** (intermédiaire) : *filter* (prédicat unaire à vérifier : IF), *distinct* (éléments distincts selon equals), *limit(n)* (stream de longueur max n), *skip(n)* (sauter les n 1ers éléments) ;
- Savoir si un ou tous les éléments vérifient un prédicat : *anyMatch*, *allMatch*, et *noneMatch* ; retourne un booléen ;
- Trouver un élément vérifiant un prédicat : *findFirst* et *findAny* ; retourne un objet optionnel (qui peut être null) ;
- Réduire : *reduce(valeur initial, opérateur binaire)* ; utile pour calculer le min, le max, la somme, le produit...

Streams : opérations possibles

- **Filtrer** (intermédiaire) : *filter* (prédicat unaire à vérifier : IF), *distinct* (éléments distincts selon equals), *limit(n)* (stream de longueur max n), *skip(n)* (sauter les n 1ers éléments) ;
- **Savoir si** un ou tous les éléments vérifient un prédicat : *anyMatch*, *allMatch*, et *noneMatch* ; retourne un booléen ;
- Trouver un élément vérifiant un prédicat : *findFirst* et *findAny* ; retourne un objet optionnel (qui peut être null) ;
- **Réduire** : *reduce(valeur initial, opérateur binaire)* ; utile pour calculer le min, le max, la somme, le produit...

Streams : opérations possibles

- **Filtrer** (intermédiaire) : *filter* (prédicat unaire à vérifier : IF), *distinct* (éléments distincts selon equals), *limit(n)* (stream de longueur max n), *skip(n)* (sauter les n 1ers éléments) ;
- **Savoir si** un ou tous les éléments vérifient un prédicat : *anyMatch*, *allMatch*, et *noneMatch* ; retourne un booléen ;
- **Trouver un élément** vérifiant un prédicat : *findFirst* et *findAny* ; retourne un objet optionnel (qui peut être null) ;
- **Réduire** : *reduce(valeur initial, opérateur binaire)* ; utile pour calculer le min, le max, la somme, le produit...

Streams : opérations possibles

- **Filtrer** (intermédiaire) : *filter* (prédicat unaire à vérifier : IF), *distinct* (éléments distincts selon equals), *limit(n)* (stream de longueur max n), *skip(n)* (sauter les n 1ers éléments) ;
- **Savoir si** un ou tous les éléments vérifient un prédicat : *anyMatch*, *allMatch*, et *noneMatch* ; retourne un booléen ;
- **Trouver un élément** vérifiant un prédicat : *findFirst* et *findAny* ; retourne un objet optionnel (qui peut être null) ;
- **Réduire** : *reduce(valeur initial, opérateur binaire)* ; utile pour calculer le min, le max, la somme, le produit...

Streams : opérations possibles

- **Trier** (comparateur : IF) : `sorted` ;
- **Appliquer une fonction** (IF unaire) : `forEach (final)`, `map` (intermédiaire) ;
- **Récupérer le résultat final** :
`collect (e.g. collect(Collectors.toList()))`.
- ...

Streams : opérations possibles

- **Trier** (comparateur : IF) : *sorted* ;
- **Appliquer une fonction** (IF unaire) : *forEach* (final), *map* (intermédiaire) ;
- **Récupérer le résultat final** :
collect (e.g. *collect(Collectors.toList())*).
- ...

Streams : opérations possibles

- **Trier** (comparateur : IF) : *sorted* ;
- **Appliquer une fonction** (IF unaire) : *forEach* (final), *map* (intermédiaire) ;
- **Récupérer le résultat final** :
collect (e.g. *collect(Collectors.toList())*).

...
...

Streams : opérations possibles

- **Trier** (comparateur : IF) : *sorted* ;
- **Appliquer une fonction** (IF unaire) : *forEach* (final), *map* (intermédiaire) ;
- **Récupérer le résultat final** :
collect (e.g. *collect(Collectors.toList())*).
- ...

Exemple sur le filtrage et le tri de points 2D

Streams : classe générique et classes spécialisées

- Pour les streams génériques (jusqu'à présent) : Stream<T> ;
- Pour les streams numériques : IntStream, LongStream et DoubleStream ; ils sont plus efficaces et munis d'opérations particulières comme max, min, sum ou range(début, fin) pour les streams entiers.

Streams : classe générique et classes spécialisées

- Pour les streams génériques (jusqu'à présent) : Stream<T> ;
- Pour les streams numériques : IntStream, LongStream et DoubleStream ; ils sont plus efficaces et munis d'opérations particulières comme max, min, sum ou range(début, fin) pour les streams entiers.

Instants et durées

```
import java.time.* ;
public class CalculDuree{
    public static void main(String [] args){
        Instant deb = Instant.now() ;
        ... // un traitement
        Instant fin = Instant.now() ;
        Duration duree = Duration.between(deb, fin) ;
        System.out.println("Ce traitement a duré " +
duree.getSeconds() + " seconds") ;
        Instant autreFin = fin.plus(duree) ;
    }
}
```

Les classes LocalDate et Period

```
import java.time.* ;
public class LocDate{
    public static void main(String [] args){
        LocalDate aujourd'hui = LocalDate.now() ;
        LocalDate demain = aujourd'hui.plusDays(1) ;
        LocalDate jourParti = LocalDate.of(1945, 5, 8) ;
        Period ecart = jourParti.until(aujourd'hui) ;
        LocalDate uneAnneePlusTard =
            aujourd'hui.plus(Period.of(1, 0, 0)) ;
    }
}
```

La classe LocalTime

```
import java.time.* ;
public class LocHeure{
    public static void main(String [] args){
        LocalTime maintenant = LocalTime.now() ;
        LocalTime dans2heures = maintenant.plusHours(2) ;
        Duration duree = Duration.between(maintenant,
        dans2heures) ;
    }
}
```

La classe LocalDateTime

- Regroupe les fonctionnalités de LocalDate et LocalTime.
- Ne gère pas le changement d'heure.

La classe LocalDateTime

- Regroupe les fonctionnalités de LocalDate et LocalTime.
- Ne gère pas le changement d'heure.

Formatage de dates

Utilisez la classe `DateTimeFormatter` du paquetage `java.time.format`.

```
import java.time.* ;
import java.time.format.* ;
public class FormatDate{
    public static void main(String [] args){
        LocalDate aujourd'hui = LocalDate.now() ;
        String dateBasic = DateTimeFormatter.BASIC_ISO_DATE.format(aujourd'hui) ;
        String dateComp = DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).format(aujourd'hui);

        LocalDateTime maintenant = LocalDateTime.now() ;
        String dateHeureComp =
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL).format(maintenant) ;
        ...
    }
}
```