

M2103 (POO)



# Bases de la programmation orientée objet

**Responsable : Cristina Onete**

cristina.onete@gmail.com

<http://onete.net/#teaching>

# Les collections

The background features abstract, overlapping geometric shapes in various shades of green, ranging from light lime to dark forest green. These shapes are primarily located on the right side of the frame, creating a modern, layered effect. The rest of the background is plain white.

# Au delà des tableaux

- ▶ On a vu des tableaux :
  - ▶ homogènes : variables de même type
  - ▶ hétérogènes : en utilisant le polymorphisme
- ▶ Les tableaux sont très utiles !
- ▶ Mais quels peuvent être leurs inconvénients ?

**Il faut toujours déclarer la taille**

**Si on surestime la taille, cela coûte des ressources**

**Une fois la taille définie, elle ne peut jamais être modifiée**

**Pour ajouter un élément à un tableau il faut connaître sa position**

# Prenons un exemple

- ▶ On s'imagine un groupe d'objets dont la taille n'est pas connue
  - ▶ Et pour lequel la taille est difficile à estimer
  - ▶ Disons: les utilisateurs d'un réseau social
- ▶ Soit une classe NetUser qui modélise les utilisateurs du réseau
- ▶ Nous voulons être capable de :
  - ▶ Retenir dans un objet tous les objets du type NetUser
  - ▶ Ajouter et enlever des utilisateurs à volonté
  - ▶ Modifier de façon dynamique la taille du groupe d'objets

# Si nous utilisons un tableau

- ▶ Première question : quelle serait la taille d'un tel tableau ?
  - ▶ Disons 7 milliards ? (tous les usagers possibles ?)
  - ▶ Mais même Facebook n'a que 2.07 milliards d'utilisateurs actifs
  - ▶ Ceci étant dit, leur nombre augmente sans cesse
- ▶ Ajouter un utilisateur :
  - ▶ Trouver la première position non-occupée et la remplir
- ▶ Modifier la taille du tableau :
  - ▶ Initialiser un nouveau tableau de la taille souhaitée
  - ▶ Copier les éléments d'un tableau à l'autre

# Une meilleure alternative : ArrayList

- Une ArrayList est une collection d'éléments de même type;

```
ArrayList<NetUser> maListe = new ArrayList<NetUser>();
```

aucune taille en entrée !

Le constructeur de la classe ArrayList  
-- d'où les parenthèses

Le type d'élément contenu dans l'ArrayList  
Attention : <> au lieu de [] !

# Une meilleure alternative : ArrayList

- ▶ Une ArrayList est une collection d'éléments de même type;

```
ArrayList<NetUser> maListe = new ArrayList<NetUser>();
```

- ▶ Ajouter un élément :

```
maListe.add(new NetUser("Jean Dupont"));
```

la taille devient 1

Un nouveau objet de type NetUser

La méthode **add(Element)** ajoute un nouveau objet à l'ArrayList  
Le type d'élément ajouté doit coïncider avec le type collectionné par l'Arraylist

# Une meilleure alternative : ArrayList

- ▶ Une ArrayList est une collection d'éléments de même type;

```
ArrayList<NetUser> maListe = new ArrayList<NetUser>();
```

- ▶ Ajouter un élément :

```
maListe.add(new NetUser("Jean Dupont"));
```

```
maListe.add(1, new NetUser("Jean Dupont"));
```

La méthode **add(index, Element)** rajoute un nouveau élément à l'index indiqué  
Les autres éléments sont déplacés à droite  
La taille de l'ArrayList augmente par 1

# Une meilleure alternative : ArrayList

- ▶ Une ArrayList est une collection d'éléments de même type;

```
ArrayList<NetUser> maListe = new ArrayList<NetUser>();
```

- ▶ Ajouter un élément :

```
maListe.add(new NetUser("Jean Dupont"));
```

```
maListe.add(1, new NetUser("Jean Dupont"));
```

- ▶ Enlever un element :

```
maListe.remove(Utilisateur);
```

Un objet de type NetUser, déclaré en préalable

# Une meilleure alternative : ArrayList

- ▶ Une ArrayList est une collection d'éléments de même type;

```
ArrayList<NetUser> maListe = new ArrayList<NetUser>();
```

- ▶ Ajouter un élément :

```
maListe.add(new NetUser("Jean Dupont"));
```

```
maListe.add(1, new NetUser("Jean Dupont"));
```

- ▶ Enlever un élément :

```
maListe.remove(Utilisateur);
```

- ▶ On peut également trouver un élément dans la collection, ou cloner la liste, retourner un élément à une position donnée...

# Les collections

- ▶ L'ArrayList est seulement un exemple de *Collection*
- ▶ *Java.util.Collection* est une interface en Java
  - ▶ Elle permet de faire des collections d'éléments
  - ▶ Les éléments doivent être impérativement des objets
    - ▶ Aucun type primitif (int, double) ne peut être collectionné
- ▶ Comme toute interface, Collection n'a que des méthodes abstraites
  - ▶ Ces méthodes sont personnalisées par les classes qui l'implémentent

# Java collection framework



# Les collections

- ▶ Java Collection Framework est un cadre de développement
  - ▶ Il contient toute une hiérarchie d'interfaces, donnant la possibilité d'opérer sur des collections de différents types
  - ▶ Les interfaces contiennent des méthodes abstraites, qui permettent la manipulation de la collection entière ou de ses éléments.
- ▶ Quelques avantages d'utiliser des collections :
  - ▶ Une collection a une taille modifiable
  - ▶ Des structures, méthodes et algorithmes déjà donnés
  - ▶ Plus d'efficacité dans la manipulation
  - ▶ Inter-opérable

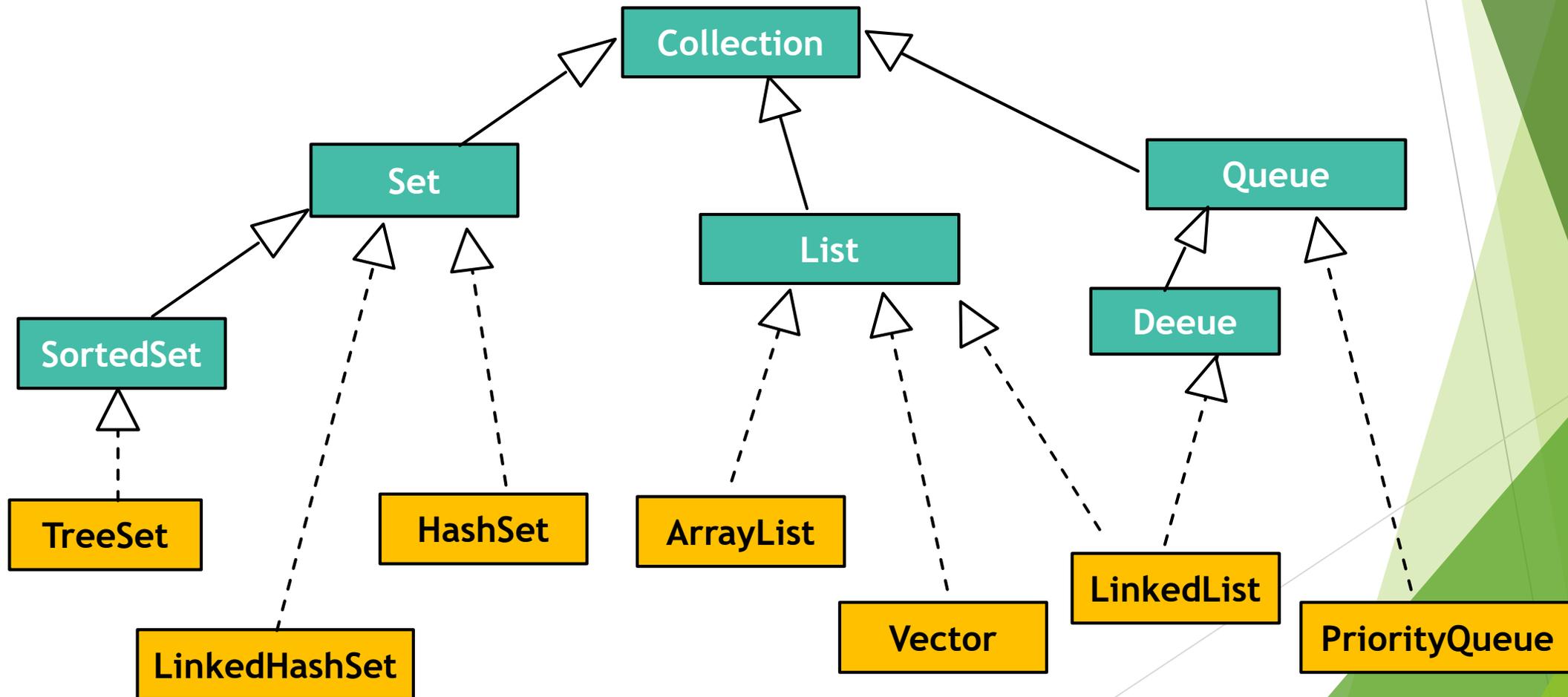
# Structure générique de Collection

- ▶ L'interface `Collection<E>` étend l'interface `Iterable<E>`  
L'interface `Iterable<E>` peut servir pour générer un itérateur

Collection
+ <code>add(E) : boolean</code> + <code>addAll(E) : boolean</code> + <code>clear() : void</code> + <code>contains(Object) : boolean</code> + <code>containsAll(Collection&lt;?&gt;) : boolean</code> + <code>equals(Object) : boolean</code> + <code>hashCode() : int</code> + <code>isEmpty() : boolean</code> + <code>iterator() : Iterator&lt;E&gt;</code> + <code>remove(Object) : boolean</code> + <code>removeAll(Collection&lt;?&gt;) : boolean</code> + <code>retainAll(Collection&lt;?&gt;) : Boolean</code> + <code>size() : int</code> + <code>toArray() : Object[]</code> + <code>toArray(T[]) : &lt;T&gt; T[]</code>

# La hiérarchie JCF

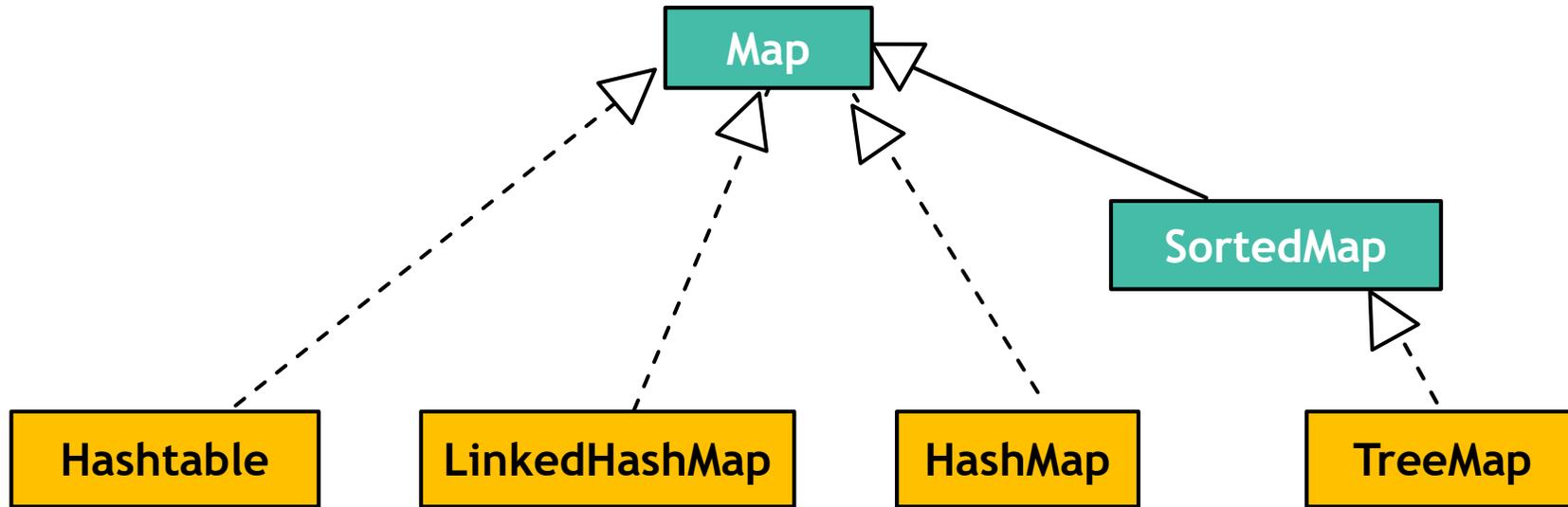
- ▶ Deux interfaces principales : **Collection** et **Map**



# L'interface Collection

- ▶ Une interface générale pour des collections des objets
- ▶ Des sous-interfaces spécialisées pour des collections différentes :
  - ▶ **Set (ensemble)** : collection dans laquelle un élément ne se répète jamais
  - ▶ **SortedSet (ensemble trié)** : ensemble avec éléments en ordre croissant
  - ▶ **List (liste)** : collection qui permet des éléments dupliqués  
les éléments sont indexés (on dit "ordonnés")
  - ▶ **Queue** : collection, stocke des éléments avant de les processor
  - ▶ **Deque (queue à double sens)** : permet l'insertion et utilisation des éléments aux deux bouts de la collection

# L'interface Map



- ▶ Les interfaces Map et SortedSet :
  - ▶ **Map** (fonction) : une association une-à-une entre des clés et des valeurs
  - ▶ **SortedMap** (fonction triée) : fonction triée par les clés

# Quelques collections en pratique

The background features abstract, overlapping geometric shapes in various shades of green, ranging from light lime to dark forest green. The shapes are primarily triangles and polygons, creating a dynamic, layered effect. The overall composition is clean and modern, with the text centered on a white background.

# Utiliser un ensemble

- ▶ Un ensemble sans doublon
- ▶ Personnalise toutes les méthodes de l'interface Collection, avec la restriction sur les doublons
- ▶ Classes qui implémentent Set :
  - ▶ HashSet : éléments stockés dans un tableau haché (opérations de base en temps constant)
  - ▶ TreeSet : éléments stockés en ordre dans un arbre
  - ▶ LinkedHashSet : éléments stockés en ordre selon leurs index d'insertion

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

public class MonSet{
    public static void main(String[] args){
        final Set<String> monEnsemble = new HashSet<String>();
        monEnsemble.add("Jean");
        monEnsemble.add("Gabrielle");
        monEnsemble.add("Jean"); // non ajouté car dupliqué
        System.out.println(monEnsemble); // affiche {Jean, Gabrielle}
        ou {Gabrielle, Jean}
        System.out.println(monEnsemble.size()); // affiche 2
    }
}
```

# Les itérateurs et l'affichage

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class MonSet{
    public static void main(String[] args){
        final Set<String> monEnsemble = new HashSet<String>();
        monEnsemble.add("Jean");
        monEnsemble.add("Gabrielle");

        Iterator<String> monIterateur = monEnsemble.iterator();
        while (monIterateur.hasNext()){
            String objetActuel = monIterateur.next();
            System.out.println(objetActuel);
            if (objetActuel.equals("Jean"))
                System.out.println("C'est Jean");
        }
    }
}
```

Les itérateurs sont utiles pour parcourir une collection non-ordonnée

Cet itérateur est propre à la collection monEnsemble

La méthode hasNext() vérifie si l'itérateur a encore des positions à parcourir

La méthode next() récupère l'objet à la prochaine position de l'itérateur

Comparer des éléments avec une valeur

# D'autres manipulations

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Set;

public class MonSet{
    public static void main(String[] args){
        final Set<String> monEnsemble = new HashSet<String>;
        monEnsemble.add("Jean");
        monEnsemble.add("Gabrielle");
        System.out.println(monEnsemble.size()); // affiche 2

        monEnsemble.remove("Gabrielle");
        System.out.println(monEnsemble); // affiche Jean
        monEnsemble.add("Amelie");
        if (monEnsemble.contains("Gabrielle"))
            System.out.println("Gabrielle est encore la.");
    }
}
```

← enlever un élément

← vérifier si la collection  
contient un certain élément  
pas besoin de parcourir la  
collection !

# Utiliser des Listes

- ▶ Les listes sont indexées
- ▶ Des doublons possibles
- ▶ Des méthodes additionnelles :
  - ▶ Pour manipuler les éléments par leur index dans la liste
  - ▶ Pour partitionner une liste en sous-listes
  - ▶ pour trier, mélanger, inverser, copier ou rechercher des éléments
- ▶ Des types de listes :
  - ▶ ArrayList : tableau redimensionné
  - ▶ LinkedList : tableau qu'on peut manipuler par ses deux extrémités

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;

public class MonSet{
    public static void main(String[] args){
        final List<Integer> maListe = new ArrayList<Integer>();
        maListe.add(Integer.valueOf(10));
        maListe.add(Integer.valueOf(5));
        maListe.add(Integer.valueOf(8));
        maListe.add(Integer.valueOf(10)); // ajouté
        System.out.println(maListe); // affiche {10,5,8,10}
        System.out.println(maListe.get(2)); // affiche 8
    }
}
```

**Integer** : un int encapsulé dans un objet  
Rappel : **collections** collectent **objets** !

# Plus de fonctionnalités

```
import java.util.Collection;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class MonSet{
    public static void main(String[] args){
        final List<Integer> maListe = new ArrayList<Integer>();
        maListe.add(Integer.valueOf(10));
        maListe.add(Integer.valueOf(5));
        maListe.add(Integer.valueOf(8));
        maListe.add(Integer.valueOf(10)); //ajouté

        System.out.println(maListe.indexOf(10)); // affiche 0
        System.out.println(maListe.lastIndexOf(10)); // affiche 3
        Collections.sort(maListe);
        System.out.println(maListe); //affiche {5,8,10,10}
    }
}
```

Fait partie de la JCF  
Beaucoup de méthodes abstraites, y compris le triage

Trouver la première position d'un élément

Trouver la dernière position d'un élément

Méthode qui s'applique à quelques types de listes  
Y compris des Integers et des Strings

# Utiliser une Map

- ▶ Une map associe des clés uniques aux éléments d'une collection
  - ▶ Clé = numéro d'étudiant, etc.
  - ▶ Une autre façon d'indexer
- ▶ Méthodes permettant la manipulation des éléments, clés, éléments par clés
- ▶ Types de maps :
  - ▶ HashMap : clés/éléments stockés dans table de hachage (efficacité)
  - ▶ TreeMap : éléments stockés dans un arbre (ordonné en ordre d'insertion)
  - ▶ LinkedHashMap : compromis entre HashMap (efficacité) et TreeMap (moins efficace mais ordonnée)

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

public class MonSet{
    public static void main(String[] args){
        final Map<String, String> Agents00 = new HashMap<>();
        Agents00.put("002", "Bill Fairbanks");
        Agents00.put("006", "Alec Trevelyan");
        Agents00.put("007", "James Bond");
        Agents00.put("002", "Bill Fairbanks"); // non ajouté, clé dupliqué

        System.out.println(Agents00); // affiche {002=Bill Fairbanks,
006=Alec Trevelyan, 007=James Bond}
    }
}
```

# D'autres opérations avec les maps

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

public class MonSet{
    public static void main(String[] args){
        final Map<String, String> Agents00 = new HashMap<>();
        Agents00.put("002", "Bill Fairbanks");
        Agents00.put("006", "Alec Trevelyan");
        Agents00.put("007", "James Bond");
        System.out.println(Agents00.get("007"));
        System.out.println(Agents00.size());
        if (Agents00.containsKey("001")){
            System.out.println(Agents00.get("001"));
        }
        else
            System.out.println("Position vacante");
        System.out.println(Agents00.keySet());
        System.out.println(Agents00.values());
    }
}
```

Trouver la valeur couplée à la clé 007

Cherche parmi les clés

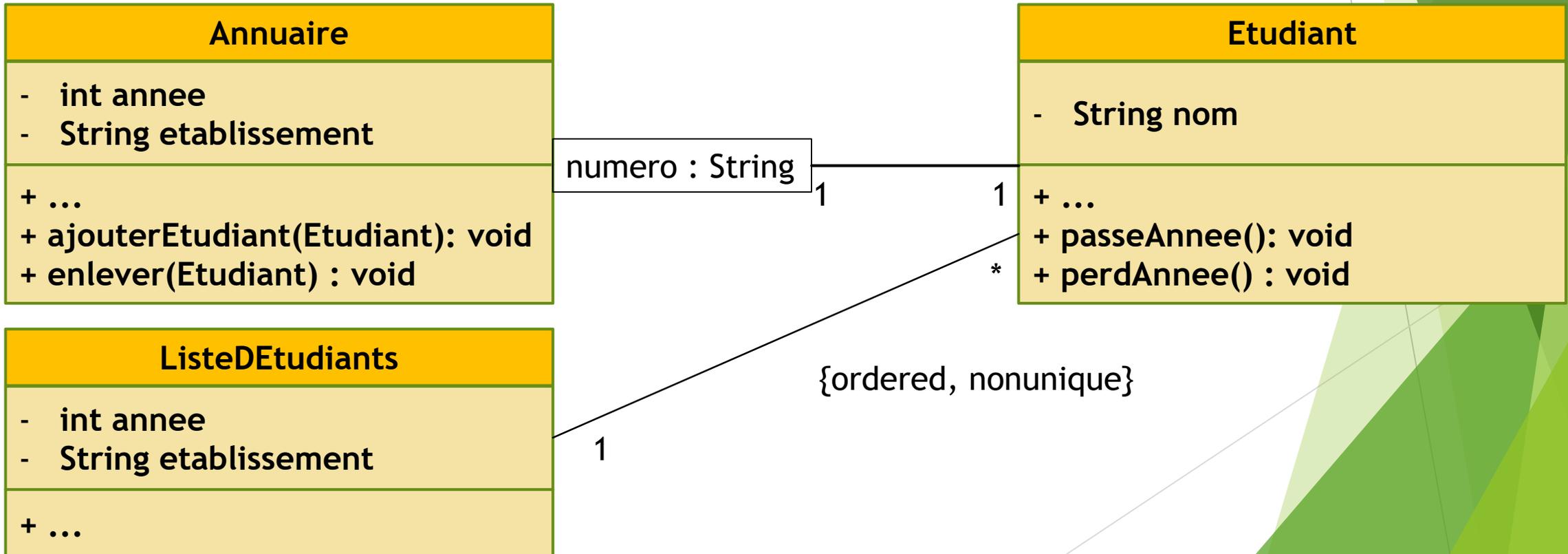
Pour chercher parmi les valeurs on a **contains()**

Affiche un ensemble de clés : **{"002", "006", "007"}**

Affiche l'ensemble de valeurs

# Diagrammes de classes

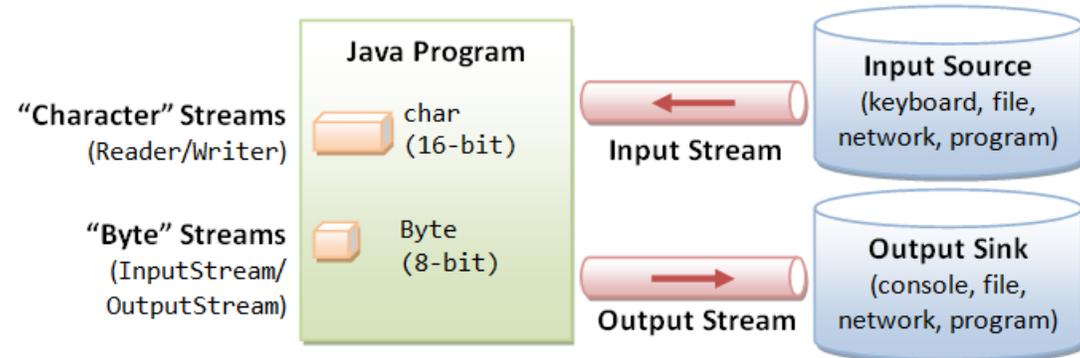
- ▶ Les collections sont en général indiquées par des associations
  - ▶ Notation non-standard : indiquées par des attributs



# Flux d'entrée et sortie

# Le package java.io

- ▶ Le flux d'entrée/sortie est essentiel en Java
  - ▶ Il est unidirectionnel
  - ▶ Diverses sources d'entrée/destinations de sortie sont possibles
  - ▶ Sources d'entrée : entrée au clavier, fichier, réseau, autre programme
  - ▶ Destinations de sortie : console Java, fichier, programme, réseau
- ▶ Des flux (streams) binaires ou de caractères



#### Internal Data Formats:

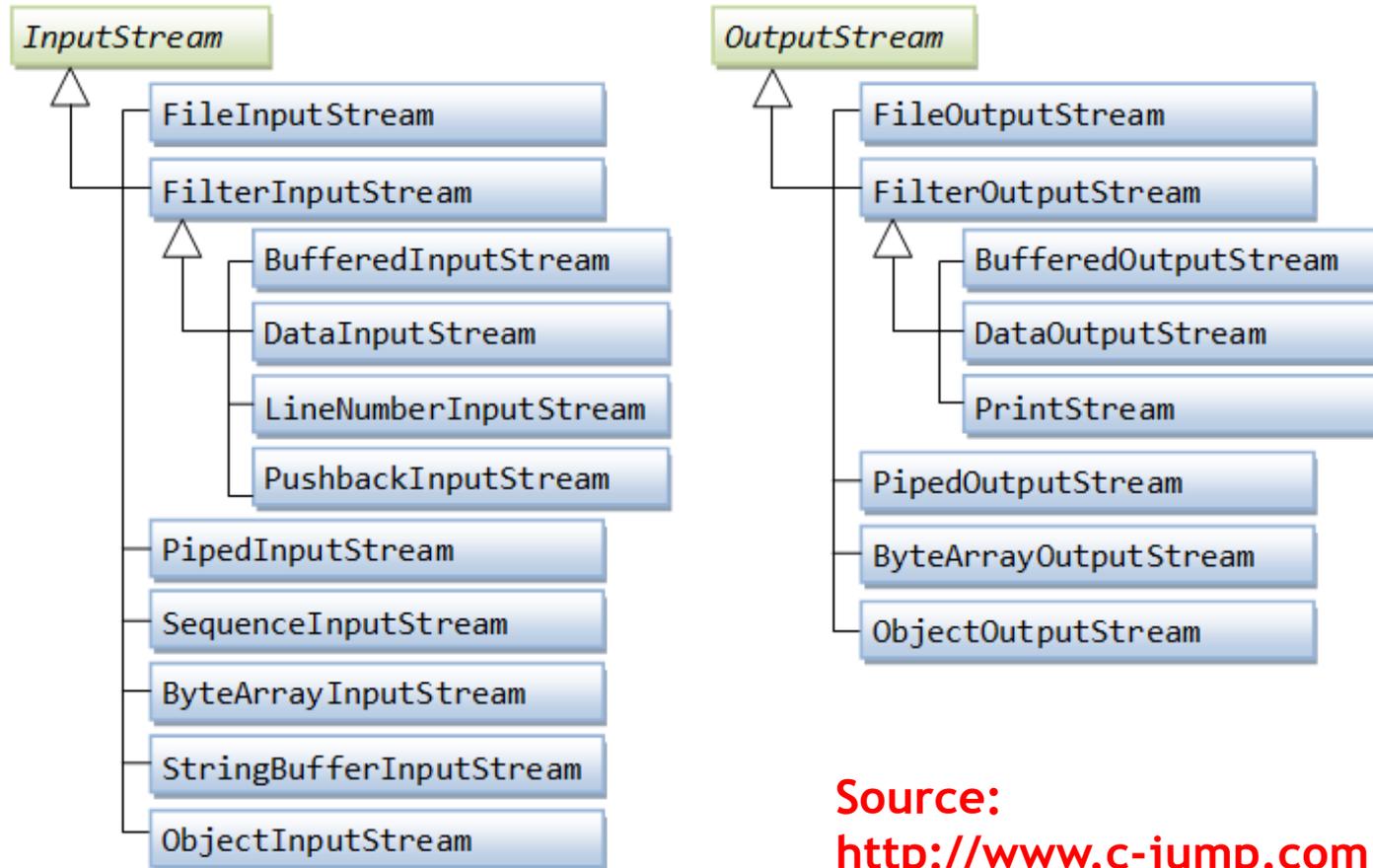
- Text (char): UCS-2
- int, float, double, etc.

#### External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

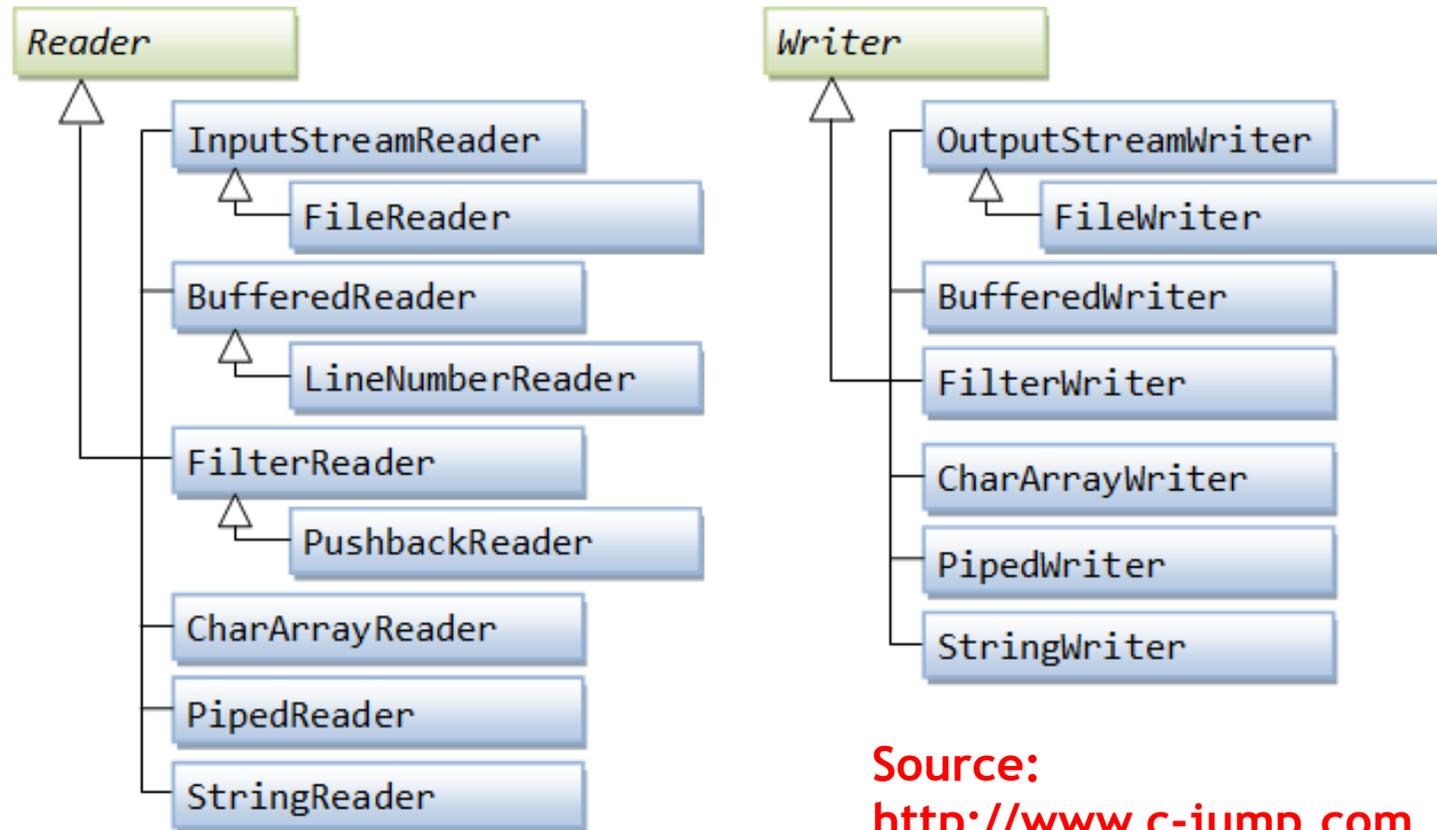
Source : <https://www.ntu.edu.sg>

# Binary Streams in Java



Source:  
<http://www.c-jump.com>

# Character streams



**Source:**  
<http://www.c-jump.com>

# Lire un fichier texte

- ▶ Le traitement des fichiers en Java implique :
  - ▶ L'ouverture correcte des fichiers
    - ▶ Avec un traitement correct des exceptions possibles y compris FileNotFoundException
  - ▶ Le traitement (par exemple lecture et/ou écriture)
  - ▶ La fermeture correcte des fichiers
    - ▶ Attention : un fichier doit toujours être fermé après l'utilisation
    - ▶ Ceci veut dire : même si une erreur se produit, même si une exception est levée, le fichier DOIT impérativement être fermé à la fin d'un programme !
  
- ▶ Nous allons voir ces étapes une par une

# Ouvrir et lire un fichier

- ▶ Un fichier peut être lu avec un lecteur de fichiers (fileReader)

- ▶ le constructeur de FileReader qu'on va utiliser à la signature

```
public FileReader(String nomFichier)
```

```
throws FileNotFoundException
```

- ▶ Les méthodes dans lesquelles on instancie un lecteur de fichiers doivent prendre en compte les exceptions !

- ▶ Des méthodes utiles dans la classe FileReader :

- ▶ `public int read()` : lit un seul caractère, lève une IOException

- ▶ `public int read(char[] buffer)` : lit des caractères dans un tableau de caractères plus efficace, lève une IOException

# Fermer un fichier

- ▶ Il faut toujours, impérativement fermer les fichiers en utilisation
- ▶ 2 façons de réaliser cela :
  - ▶ le bloc finally d'un bloc try-catch-finally s'exécute toujours
  - ▶ Depuis Java 7 l'interface `java.lang.AutoCloseable` : try-catch suffit
    - ▶ Interface implémentée par la plupart des lecteurs/écrivains et flux I/O.

```
import java.io.*;

public class Lire{
    public static void main(String[] args){
        try (FileReader lecteur = new FileReader("C:/Documents/File.txt")) {
            // lire par caractere
            int caract;
            while ( (caract = lecteur.read()) != -1 )
                System.out.print((char)caract);
        }
        catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Optimiser la lecture

- ▶ La lecture caractère par caractère est inefficace
- ▶ Pour être plus efficace on peut lire par tableau de caractères
  - ▶ De taille variable
- ▶ Pour des gros fichiers, cela peut faire toute la différence

```
import java.io.*;

public class Lire{
    public static void main(String[] args){
        try{
            FileReader lecteur = new FileReader("C:/Documents/File.txt");
            char[] monContenu = new char[128];
            while ( (lecteur.read(monContenu)) != -1 ){
                for (int i=0; i<128; i++){
                    System.out.print(monContenu[i]);
                }
            }
        }
        catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

# Ecrire dans un fichier

- ▶ Utilisation d'un scribe de fichiers : `fileWriter`
- ▶ Deux constructeurs très utiles :
  - ▶ `public FileWriter(String filename)` : lève `IOException`
  - ▶ `public FileWriter(String filename, boolean append)` : lève `IOException`
    - ▶ si `append == true`, alors les données en entrée sont écrites à la fin du fichier, plutôt qu'au début
    - ▶ très utile dans un log, ou dans un fichier où l'ordre doit être préservée
- ▶ Des retours de lignes possibles avec `"\n"`.

# Ecrivons dans notre fichier

```
import java.io.*;

public class Lire{
    public static void main(String[] args){
        try{
            FileWriter ecriteur = new FileWriter("C:/Documents/File.txt", true);
            ecriteur.write(" Je veux continuer ce fichier, alors voici\n le reste de la
phrase que j'ai ecrite.");
        }
        catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

# D'autres opérations intéressantes

- ▶ La classe `BufferedReader` optimise la lecture d'un fichier
- ▶ La classe `File` (fichier)
  
- ▶ Plus d'infos :
  - ▶ <https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>
  - ▶ <https://docs.oracle.com/javase/7/docs/api/java/io/File.html>

