Conception et programmation objet avancées

Java – rappels et compléments

Petru Valicov
petru.valicov@univ-amu.fr

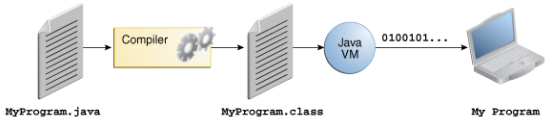
2017-2018

Institut Universitaire de Technologie
Aix-Marseille Université
Département Informatique, Aix

Java

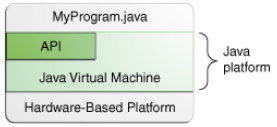
Java est un langage de programmation

- langage de haut niveau, objet, portable, ...
- langage pseudocompilé (*bytecode*) exécuté par la *JVM*



Java est aussi une plateforme

- plateforme portable (*Write once, run anywhere*)
- interface de programmation d'application (API)
  - Swing, AWT, JavaFX
  - JDBC
  - Réseau, HTTP, FTP
  - IO, math, config, etc.



Pourquoi Java dans ce cours ?

- Simple à prendre en main
  - l'API officielle très riche et assez fiable,
  - pas très verbeux
  - gestion automatique de la mémoire :
    utilisation implicite des pointeurs (références)
    le ramasse-miettes (*garbage collector*)
    } plus de sécurité
- Les APIs Java illustrent souvent des principes de la CPOA

Java : objet

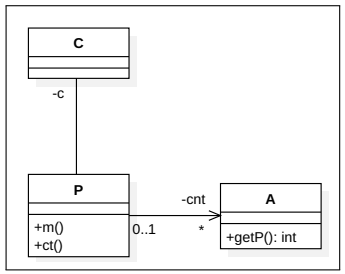
- Un objet est l'association d'un état et d'un comportement (très lié à l'objet d'UML)
- L'état est **stocké** dans des **champs** (*fields*)
- Le **comportement** est exposé via des **méthodes** (*methods*)
- Principe d'**encapsulation** (un des 3 piliers de la POO) : les méthodes manipulent l'état interne de l'objet et servent à la communication inter-objets
- les champs sont privés à un objet : **modularité**
  $\Longrightarrow$ réutilisation, debuggage plus facile, implémentation parallèle...
- En Java, hormis les types primitifs, tous les autres types sont des objets (héritent par défaut de la classe `Object`)

Écrire du code – conventions de nommage Java

Les conventions ne sont pas imposées par le compilateur, mais tout le monde s'attend à ce qu'elles soient suivies.

Règle d'or : *Donner des noms simples et descriptifs*

Qu'est-ce que c'est ?




```java
public class P{
  private C c;
  private List<A> cnt = new ArrayList<A>();

  public P(C c){
     this.c = c;
  }

  public void m(A a, int q){
     for (int i=0; i<q; i++){
        cnt.add(a);
     }
  }

  public int ct(){
     int t = 0;
     for (A a: cnt){
        t += a.getP();
     }
     return t;
  }
}
```

Écrire du code – conventions de nommage Java

Les conventions ne sont pas imposées par le compilateur, mais tout le monde s'attend à ce qu'elles soient suivies.

Règle d'or : *Donner des noms simples et descriptifs*

```java
public class P{
  private C c;
  private List<A> cnt = new ArrayList<A>();

  public P(C c){
     this.c = c;
  }

  public void m(A a, int q){
     for (int i=0; i<q; i++){
        cnt.add(a);
     }
  }

  public int ct(){
     int t = 0;
     for (A a: cnt){
        t += a.getP();
     }
     return t;
  }
}
```

```java
public class Panier{
  private Client client;
  private List<Article> contenu = new ArrayList<Article>();

  public Panier(Client client){
     this.client = client;
  }

  public void ajouter(Article a, int quantite){
     for (int i=0; i<quantite; i++){
        contenu.add(a);
     }
  }

  public int calculerTotal(){
     int total = 0;
     for (Article a: contenu){
        total += a.getPrix();
     }
     return total;
  }
}
```

Conventions de nommage Java - lexique

- package
  - tout en minuscule avec les mots collés
  - exemples : `java.util`, `javafx.scene`, `java.lang.reflect`, `fr.univamu.iut`
- champ `static final`
  - tout en majuscule avec des "_" entre les mots
- classe et interface :
  - 1ère lettre en majuscule
  - mots attachés, avec majuscule aux premières lettres de chaque
  - ex. : `MaPetiteClasse`, `BufferedReader`, `LinkedHashSet`
- champ, méthode et variable : comme pour une classe/interface mais 1ère lettre en minuscule

`Fenetre.expand()` vs `fenetre.expand()`

Conventions de nommage Java - sémantique

- classe :
  - groupe nominal au singulier
  - exemples : `Client`, `Reservation`, `CompteEpargne`
- interfaces :
  - même manière que les classes mais peuvent être des adjectifs
  - exemples : `List`, `Comparable`, `Serializable`
- champ et variable :
  - groupe nominal singulier ou pluriel
  - des noms très courts **uniquement** pour les variables à la durée de vie très courte (compteurs de boucle ou autre)
  - utiliser uniquement des caractères de l'alphabet [A-Z], [a-z], [0-9]
- méthode :
  - doit comporter un verbe (en général à l'infinitif)
  - `fermer()`, `ajouterElement()`, `calculerSomme()`, `get()`
- anglais ou français ? **À vous de choisir mais pas de mélange** !

Qu'en pensez-vous ?

```java
import java.awt.Color;

public class Rectangle {
	private int longueur, largeur;
	private Color couleur;

	public Rectangle(int longueur, int largeur, Color couleur) {
		this.longueur = longueur;
		this.largeur = largeur;
		this.couleur = couleur;
	}

	public void changer(int x){
		longueur = x;
	}

	// on augmente suivant la proportion, mais on ne diminue jamais
	public void resize(int proportion){
		if (proportion > 1){
			longueur = longueur * proportion;
			largeur = largeur * proportion;
		}
	}

	public void changerCouleur(){
		couleur = Color.RED;
	}

	public Color coloration(){
		return couleur;
	}
}
```

Ça fonctionne mais...

Quelques conseils supplémentaires

"*Don't comment bad code - rewrite it !*"

B.W. Kernighan et P.J. Plaugher

Des fonctions courtes et simples (une seule action à la fois)

Des noms descriptifs pour les méthodes et les variables

Évitez la notation hongroise

- c'était utile à l'époque quand les compilateurs ne proposaient pas un bon système de types
- choisir des bons noms pour les méthodes et les variables suffit
- les IDEs d'aujourd'hui vous aident !
- "*gusing adjHungarian nnotation vmakes greading ncode adjdifficult*"

  Mark Stock

Héritage

- Relation "**est un**" entre deux types d'objets (attention : il faut qu'elle ait du sens !)
- Toutes les classes héritent d'une unique autre classe (par défaut `Object`)
- La sous-classe est liée par le "*contrat*" de sa classe parente
  - elle ne peut pas supprimer, ou ne pas hériter d'une méthode définie dans sa classe parente
  - elle ne peut pas restreindre la visibilité d'une méthode de la classe parente

Listing 1 : OK

```java
public class A {
   public void methode(){
     // corps
   }
}
```

Listing 2 : interdit

```java
public class B extends A{
   private void methode(){
     // corps
   }
}
```

Héritage : casting

```java
public class Animal{
    private String nom;
    public Animal(String nom){
        this.nom = nom;
    }

    public void dormir(){
      //corps
    }
}
```

```java
public class Chien extends Animal{

    public Chien(String nom){
        super(nom)
    }

    public void aboyer(){
      //corps
    }
}
```

```java
public class Chat extends Animal{

    public Chat(String nom){
        super(nom)
    }

    public void miauler(){
      //corps
    }
}
```

cast implicite *vs* cast explicite :

```java
Animal a = new Chien ("Bernny"); // OK ou pas ?
Chien c = (Chien) a; // OK ou pas ?
c.aboyer(); //OK ou pas?
((Chien)a).aboyer(); //OK ou pas?
Animal b = new Chat ("Tom"); //OK ou pas?
Chien c1 = (Chien) b; //OK ou pas ?
c1.aboyer(); //OK ou pas ?
```

L'opérateur `instanceof` permet de tester si un objet est d'un type (ou sous-type) donné.

De manière générale, les architectures bien réfléchies doivent pouvoir éviter son utilisation... **préférez le polymorphisme !**

Héritage : classe abstraite

- deux objets peuvent avoir *quelque chose* en commun, mais ce *quelque chose* ne peut pas exister de manière indépendante (l'exemple avec `Humain`)
- la classe abstraite ne peut pas être instanciée : erreur de compilation
- une classe abstraite peut être utilisée comme type apparent d'une référence

```
public abstract class Humain {
  public void manger(){
    // corps
  }
}
```

```
public class Femme extends Humain {
  // aucun attribut à ajouter
  // aucune méthode à déclarer
}
```

```
Humain h1 = null; // OK ou pas ?
Humain h2 = new Femme(); // OK ou pas ?
h2.manger(); // OK ou pas ?
Humain h3 = new Humain(); // OK ou pas ?
```

Héritage : méthodes abstraites

Une classe abstraite peut contenir des méthodes abstraites :

- pas d'implémentation fournie
- ses sous-classes doivent définir toutes les méthodes abstraites (ou être abstraites elles-mêmes)
- ainsi on garantit que toutes les classes filles "concrètes" *savent effectuer* ces opérations (on ne sait pas comment)

```
abstract class ObjetGraphique{
  int x, y;
  ...
  void deplacer(int nouveauX,
       int nouveauY){
    ...
  }

  abstract void dessiner();
  abstract void redimensionner();
}
```

```
class Bonhomme extends ObjetGraphique{

  void dessiner(){
    System.out.println("Je dessine une
        tête, 2 jambes et 2 mains");
  }
  void redimensionner(){
    System.out.println("Je change mes
        dimensions");
  }
}
```

Héritage : interface

Une interface est équivalente à une classe abstraite qui n'a **que** des méthodes abstraites et/ou des champs `static final`.

Est utilisée comme type apparent d'une référence (pas instanciée).

Il y a un avantage par rapport à une classe abstraite... **Lequel ?**

Un *"petit"* changement à partir de `Java 8` – les interfaces peuvent contenir des méthodes par défaut

```
public interface MonInterface {
   public void faireQuelqueChose();

   default public void faireAutreChose(){
      System.out.println("Implémentation par défaut");
   }
}
```

```
public class MaClasse implements MonInterface {
   public void faireQuelqueChose(){
      System.out.println("Mon implémentation");
    }
}
```

```
public class ClasseDeTest {
   public static void main(String[] args) {
      MonInterface obj = new MaClasse();
      obj.faireQuelqueChose(); // classique
      obj.faireAutreChose(); // default
   }
}
```

Héritage de **comportement**

L'utilisation des méthodes par défaut devrait être justifiée (et a priori interdite pour vous) !

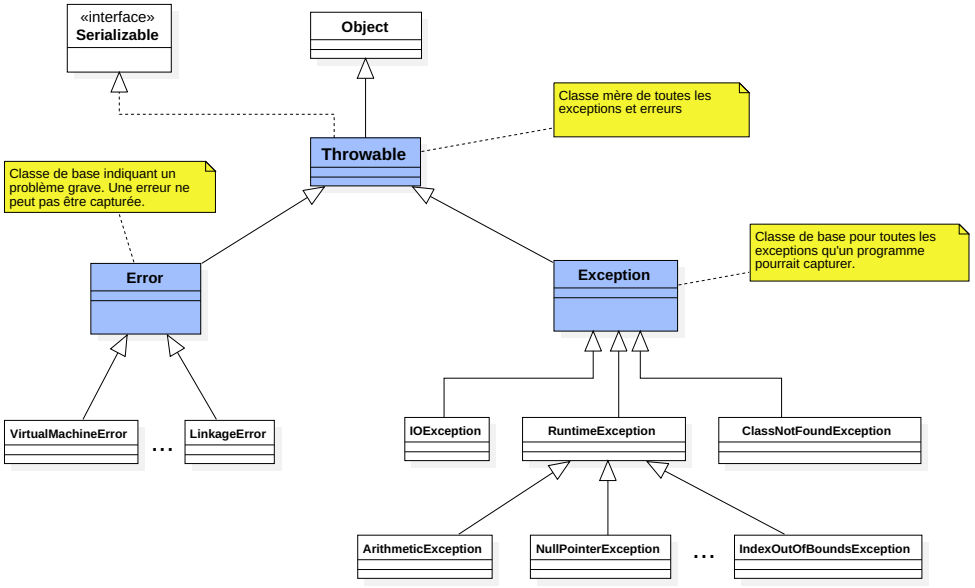
Les exceptions

- contraire à l'exécution classique du code (du début à la fin)
- une exception **ne doit pas** être considérée comme un bug
- quand une exception se produit :
  1. l'exécution normale du code s'arrête
  2. un objet de type `Exception` (ou une de ses sous-classes) est créé
  3. cet objet est transféré au runtime

  } **levée** d'exception (*throw*)
  4. le runtime Java cherche un bloc de traitement d'exception (*exception handler*) dans la pile d'appel } *catch*

```
System.out.println("On commence");
try{
  // du code susceptible de poser problème (i.e. lever une exception)
} catch (NullPointerException e){
  // traitement... lequel ?
} catch (IndexOutOfBoundException e){
  // traitement... lequel ?
} finally{
          //---Ici code2---
}
          //---Ici code3---
```

Hiérarchie des classes d'exceptions



Les exceptions : capture

- le runtime Java cherche dans la pile des appels le bloc `catch` le plus profond pour ce type d'exception

  le bloc `catch` définit le traitement approprié à l'exception

  `e.getMessage()`
  `e.printStackTrace()`
  } les actions standards (minimales)

- l'exécution reprend ensuite après ce bloc
- le fonctionnement est implicite pour les exceptions de sous-type `RuntimeException`
- très pratique pour le debuggage

Les exceptions : transfert

Pour les autres types d'exceptions, il faut indiquer au compilateur que vous acceptez les "risques" que l'exception soit *transférée* :

```
public void lire() throws IOException{
    System.in.read();
}
```

Du coup, effet boule de neige : toutes le méthodes appelant `lire()` doivent soit inclure un `try/catch` soit avoir une clause `throws`

```
public void parser() throws IOException{
    lire();
}
```

ou

```
public void parser(){
    try{
        lire();
     }catch(IOException e){
        System.out.println("Quelque chose de pas très cool vient de se passer !");
        System.out.println(e.getMessage()); //On affiche au moins le message détaillé
        System.out.println("Au secours !!!");
     }
}
```

```
public class MauvaisNombreException extends Exception {
    public MauvaisNombreException(String s){ /* construction */}

    public String recupererMessage(){
        return "Erreur de nombre";
    }
}
```

```
public class ExemplesExceptions {
    public int diviser(int divise, int diviseur) throws MauvaisNombreException{
        if (diviseur == 0){
            throw new MauvaisNombreException("Ne peut pas diviser par 0");
        }
        return divise / diviseur;
    }

    public void appelDiviseur(){
        try {
            int resultat = diviser(2,1);
            System.out.println(resultat);
            resultat = diviser(2,0);
            System.out.println(resultat); //pas exécuté
        } catch (MauvaisNombreException e) {
            //traitement approprié
            System.out.println(e.getMessage());
        }
        System.out.println("Essai de la division terminé");
    }

    public void appelDiviseur2() throws MauvaisNombreException{
        int resultat = diviser(2,1);
        System.out.println(resultat);
        resultat = diviser(2,0);
        System.out.println(resultat); //pas exécuté
    }
}
```

Qu'est-ce qu'on en fait ?

- Traiter les exceptions à l'endroit utile :
  - annuler l'opération en cours
  - afficher un message à l'utilisateur
  - rectifier la situation et relancer le traitement
- Ne **jamais** lever une exception et ne rien faire !
- Spécifier le type le plus précis possible :

  pas de `catch (Exception e)`

  pas de `throws Exception`
- N'hésitez pas à définir des exceptions personnalisées
- Utiliser le bloc "*try with ressources*" pour libérer les ressources (fichiers, connexion réseau, ...) ouvertes dans le `try` correspondant
- Une bonne gestion des exceptions indique généralement un projet solide et bien structuré

Généricité

Pour quoi faire ? Des exemples ?

- Abstraction du type d'un ou des objets sur lesquels on travaille
- La généricité participe à la *type safety*
- Élimination des "casts"
- Syntaxe : `TypeGenerique<T1,T2,...>`
- Utilisée partout où un type est nécessaire (type de variable/champ, constructeur, ...)

Généricité - exemple

```
public class Entry<K,V>{
    private final K key;
    private final V value;

    public Entry(K k, V v){
        key = k;
        value = v;
    }

    public K getKey(){
        return key;
    }

    public V getValue(){
        return value;
    }

    public String toString(){
        return "(" + key + ","+value+")";
    }
}
```

```
Entry<String, String> grade = new Entry<String, String>("Albert", "B");
Entry<String, Integer> marks = new Entry<String, Integer>("Albert", 16);
System.out.println("grade: " + grade);
System.out.println("marks: " + marks);
// Un djoker peut être utilisé pour plus de flexibilité :
Entry<?, ?> randomNotation = new Entry<String, Double>("Albert", 0.7);
```

Autoboxing

- Les types génériques ne peuvent pas être des types primitifs :

  `List<int> liste = new LinkedList<int>(); // interdit !`
- Il existe une classe équivalente pour chaque type primitif

  ex. `Integer` pour `int`, `Boolean` pour `boolean`
- Pour ajouter des `int` dans une collection (ici `List`), il faut utiliser la classe d'emballage `Integer` :

```
List<Integer> liste = new LinkedList<Integer>();
int nombre = 5;
liste.add(new Integer(nombre)); //emballage (boxing)
liste.add(nombre); //emballage automatique (autoboxing)
int v1 = liste.get(0); // déballage automatique (auto-unboxing)
int v2 = liste.get(1); // déballage automatique (auto-unboxing)
Integer v3 = liste.get(0);
Integer v4 = liste.get(1);
int v5 = v4.intValue(); // déballage
```

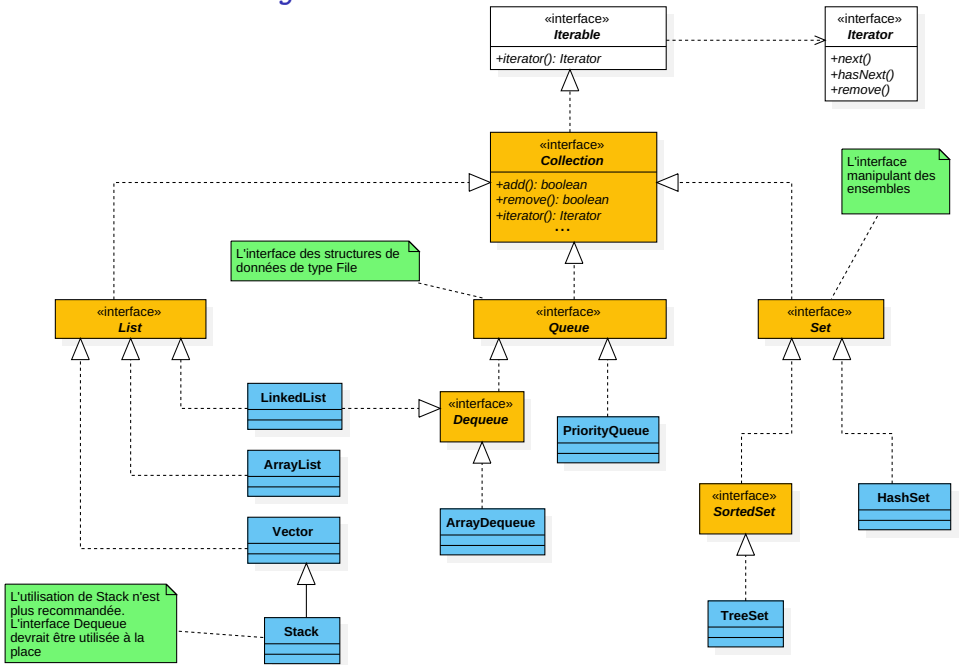
Java Collections Framework

- Contient différentes APIs de *collections*
- Systèmes d'interfaces, d'implémentations et d'algorithmes
- Hiérarchie des types
- Permet une meilleure interopérabilité entre d'autres APIs du langage

Organisation :

- Dans le package `java.util`
- Deux interfaces de base :
  - `Collection`
  - `Map`
- Une interface pour itérer sur les collections : `Iterator`

`java.util.Collection`



API de collections

Interface `Collection` :

- `Set` - notion d'ensemble mathématique
- `List` - vous la connaissez
- `Queue` - stockage temporaire (file FIFO)
- `Deque` (double ended queue) - file à deux bout

Interface `Map` – généricité sur deux types

Interface Collection

```
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(TypeGenerique element);
boolean remove(Object element);
Iterator<TypeGenerique> iterator();
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends TypeGenerique> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
void clear();
Object[] toArray();
```

Un moyen simple d'itérer

```
for (Type var : collection) {
    //traitement
}
```

- Utilise en interne la fonction `iterator()`
- Pour supprimer des éléments ou itérer sur plusieurs collections, il faut utiliser `iterator()` manuellement.

```
Collection<TypeGenerique> c = ... ;
for (Iterator<TypeGenerique> it = c.iterator(); it.hasNext();) {
    TypeGenerique o = it.next();
    if (o.getVal() > 20)
       it.remove(); //Supprime o de c
}
```

- Consigne : ne pas utiliser le `for` classique !

API de collections

- On utilise en général comme type apparent une des interfaces (`List`, `Set`)
- On choisit l'implémentation correspondante

```
List<Integer> liste = new LinkedList<Integer>();
//List<Integer> liste = new ArrayList<Integer>(); - à utiliser plus tard

// du code utilisant la liste
int nouveauNombre = 42;
liste.add(nouveauNombre);

int moyenne = 0;
for (Integer e : liste){
    System.out.println("Valeur : " + e);
    moyenne += e;
}
if (liste.size() > 0)
    moyenne /= liste.size();
System.out.println("Moyenne = " + moyenne);
```

Interface Set

- Mêmes méthodes que `Collection`
- Unicité des éléments (`add(element)` retourne `false` si l'élément est déjà présent)
- Comparaison des ensembles (avec `equals`)
- Trois implémentations :
  - `HashSet` - ordre aléatoire (performant)
  - `TreeSet` - ordonnée en fonction des valeurs du set
  - `LinkedHashSet` - ordre d'ajout
- L'ordre est celui de parcours avec `foreach`
- Les opérations mathématiques avec les ensembles : `containsAll`, `addAll`, `retainAll`, `removeAll`

Interface Set - exemples

Que font-ils ?

```
import java.util.Set;
import java.util.TreeSet;

public class TestSet {

    public static void main(String[] args) {
        Set<String> s = new TreeSet<String>();

        for (String a : args)
            if (!s.add(a))
                System.out.println("Mot "+a);

        System.out.println(s.size()+" mots");
        System.out.println("La liste des mots : "+s);
    }

}
```

```
Collection<TypeG> c = new ArrayList<TypeG>();
//ajout des éléments dans c
Collection<TypeG> c2 = new HashSet<TypeG>(c);
```

Interface List

- Collection ordonnée
- Méthodes d'accès à un élément avec son index :

  `get(int i)` et `set(int i, TypeG element)`
- Recherche l'index d'un élément : `indexOf`, `lastIndexOf`
- Sous-liste : `List<TypeG> subList(int from, int to)`
- Méthodes d'ajout et de suppression :

```
boolean add(E element)
void add(int index, E element)
E remove(int index)
boolean addAll(int index,Collection<? extends E> c)
```

- avantages algorithmiques ?

Interface List

Plusieurs implémentations. On s'intéresse à trois :

- `LinkedList` - liste chaînée
- `ArrayList` - tableau dynamique
- `Vector` - tableau dynamique, supporte le multithreading

Quels sont les avantages et les inconvénients ?

Itération sur List

Un itérateur spécifique `ListIterator` (implémente `Iterator`) :

- parcours à l'envers : `hasPrevious()`, `previous()`
- accès à l'index : `nextIndex()`, `previousIndex()`
- modification de la liste (à utiliser avec précaution) :

  `add(E e)`, `remove()`, `set(E e)`

```
List<Integer> liste = new LinkedList<Integer>();
ListIterator<Integer> monIterateur = liste.listIterator();

while(monIterateur.hasNext()){
   System.out.println(monIterateur.next());
}

while(monIterateur.hasPrevious()){
   System.out.println(monIterateur.previous());
}
```

La classe Stack - exemple de conception erronée

Étend la classe `Vector`

- Principe de LIFO
- Quatre opérations de base :

  `empty()`, `peek()`, `push(E e)`, `pop()`
- Opération supplémentaire : `search(Object o)`

Quelques problèmes avec `Stack` :

- Hérite de `Vector` - donc par défaut c'est un tableau
- N'est pas une interface
- Les opérations de pile ne sont pas très optimales

À éviter (utiliser plutôt `Deque`)

Interface Queue

- Stockage temporaire d'éléments en attente de traitement
- On ne choisit pas l'endroit d'insertion
- Uniquement le premier élément (la tête) de la queue est accessible
- Différentes implémentations
  - `LinkedList` : queue FIFO
  - `PriorityQueue` : les éléments sont traités par ordre de priorité
  - `Deque` : une queue utilisable dans les deux sens (LIFO)
- Les méthodes ont deux versions :

`add(e)`	`remove()`	`element`	lèvent une exception si problème
`offer(e)`	`poll()`	`peek()`	retournent une valeur spéciale

Comment ordonner les objets ?

- Certaines implémentations de collections ont un ordre : `TreeSet`, `PriorityQueue`, `TreeMap`.
- L'ordre est défini par l'interface `Comparable`
- Les classes Java standard ont déjà un ordre naturel (`Integer`, `String`)
- **Avantage ?**

Pour utiliser l'interface `Comparable` :

1. implémenter l'interface :
`public class MaClasse implements Comparable<MaClasse>`
2. définir la méthode `compareTo(MaClasse o)`
3. redéfinir `equals(Object o)`

Interface Comparable<E>

```java
public class Ville implements Comparable<Ville>{
    String nom;
    Integer codePostal;
    Integer population;

    public Ville(String n, int cp, int p){
        nom = n;
        codePostal = cp;
        population = p;
    }

    public int compareTo(Ville o) {
        return population.compareTo(o.population);
    }
}
```

```java
public interface Comparable<T>{
   public int compareTo(T o);
}
```

```java
public class TestComparable {
    public static void main(String[] args) {
        Ville marseille = new Ville("Marseille", 13000, 850636);
        Ville aix = new Ville ("Aix-en-Provence", 13080, 140684);

        if (marseille.compareTo(aix) > 0)
            System.out.println("Marseille plus grand");
        else if (marseille.compareTo(aix) < 0)
            System.out.println("Aix plus grand");
        else System.out.println("ils ont la meme population !");
    }
}
```

Interface Comparator

- Imaginons qu'on ne souhaite pas modifier les objets.
- Par exemple : d'abord ordonner les employés suivant leur salaire et quelques jours plus tard suivant leur date d'embauche.
- L'interface `Comparator` permet de définir un ordre sur les objets sans que cet ordre devienne une propriété de l'objet.
- On passe un `Comparator` en paramètre du constructeur de la collection qui doit utiliser un ordre particulier :

```java
public interface Comparator<T>{
   public int compare(T o1, T o2); //on compare o1 à o2
}
```

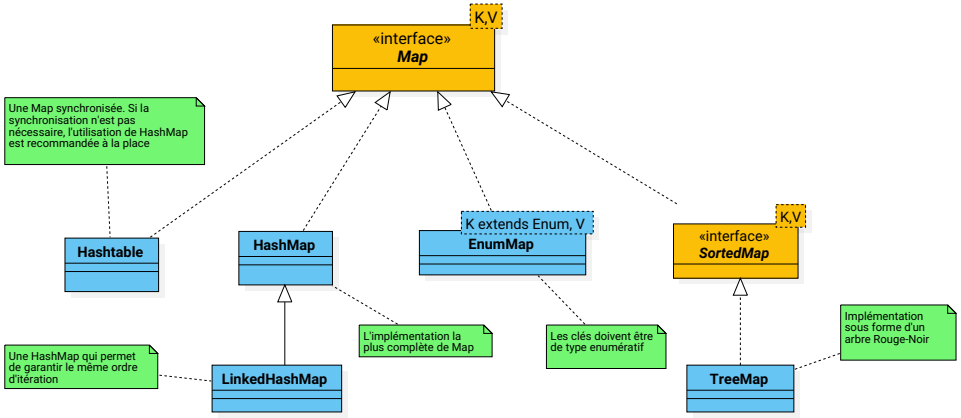
Interface Comparator - exemple

```java
import java.util.Comparator;

public class VilleOrdreCP implements Comparator<Ville>{
     public int compare(Ville v1, Ville v2){
          //comparaison en fonction du code postal
          return v2.getCodePostal().compareTo(v1.getCodePostal());
     }
}
```

```java
import java.util.SortedSet;
import java.util.TreeSet;

public class TestComparator {
     public static void main(String[] args) {
          Ville marseille = new Ville("Marseille", 13000, 850636);
          Ville aix = new Ville ("Aix-en-Provence", 13080, 140684);

          SortedSet<Ville> ensemble = new TreeSet<Ville>(new VilleOrdreCP());

          ensemble.add(aix);
          ensemble.add(marseille);

          for(Ville v : ensemble){
               System.out.println(v.getCodePostal());
          }
     }
}
```

Interface java.util.Map

- Fait correspondre une clef unique de type arbitraire à une valeur
- La clef et la valeur sont de types génériques



Une `Map` *c'est comme un tableau sauf qu'indexé par des objets.*

Interface Map

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

public class HashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> villes = new HashMap<String, Integer>();

        villes.put("Marseille", 13000);
        villes.put("Aix", 13080);
        villes.put("Lyon", 69000);
        villes.put("Paris", 75000);

        String clefDeRecherche = "Lyon";
        if(villes.containsKey(clefDeRecherche))
            System.out.println("Code postal: " + villes.get(clefDeRecherche));
    }
}
```

Interface Map

- `get(clef)`, `remove(clef)` - `null` si pas présent
- `put(clef, valeur)` - efface la valeur existante
- `containsKey(clef)`, `containsValue(val)`
- `size()`, `isEmpty()`, `clear()`
- `Set<K> keySet()`, `Collection<V> values()`

Il est possible d'associer plusieurs valeurs à une clef :

`Map<E, List<E2>>`

Interface Map

Mêmes types d'implémentations que pour `Set` :

- `HashMap` - performance
- `TreeMap` - ordre
- `LinkedHashMap` - facilité

Itération sur une `Map` :

```
for (Map.Entry<TClef, TVal> entry : map.entrySet()){
  System.out.println(entry.getKey() + "/" + entry.getValue());
}
```

`Map` n'implémentent pas `Iterable` (contrairement à `Collection`).

Algorithmes

- Définis dans la classe `Collections` (**et pas** `Collection`).
- La classe contient uniquement des méthodes statiques qui manipulent des collections (des objets de type `Collection`) :

  `Collections.reverse(List l)`

  `Collections.swap(List l, int index1, int index2)`

  `Collections.rotate(List list, distance)`

  `Collections.shuffle(List l)`

  `Collections.addAll(Collection c, elements)`

  `Collections.sort(List l)`

  `synchronizedCollection(Collection c)`

  etc...

Classes Internes Anonymes

- Permettent de définir des sous-classes à "usage unique"
- Peuvent implémenter les interfaces directement
- Pratique pour les classes courtes à utiliser à un seul endroit

```
interface Comparator {
  public int compare(Object o1, Object o2);
  public boolean equals(Object obj);
}
```

```
List<Employe> employes = new LinkedList<Employe>();
Collections.sort(employes, new Comparator<Employe>() {//debut du code
    public int compare(Employe e1, Employe e2){
        return e1.getSalaire().compareTo(e2.getSalaire());
    }
} //Fin de classe anonyme
       ); //Fin de l'appel de sort
```

Classes Internes Anonymes

Souvent utilisées pour attacher de manière rapide des écouteurs dans les interfaces graphiques :

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

```
public class Fenetre {
    //...
    public Fenetre() {
        //...
        Button bouton = new Button("monBouton");
        bouton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                // traitement associé au clic de bouton
            }
        });
        //...
    }
}
```

Classes Anonymes - amélioration en Java 8

Lorsque la classe anonyme est très simple (une seule méthode par ex. `ActionListener`, `Comparator`), la syntaxe est un peu lourde

Les *lambda expressions* permettent de pallier ce problème :

```
public class Fenetre{
    public Fenetre(){
        //...
        Button bouton = new Button("monBouton");
        bouton.addActionListener(event -> //Code correspondant);
    }
}
```

```
List<Employe> employes = new LinkedList<Employe>();
Collections.sort(employes, (Employe e1, Employe e2) ->
     e1.getSalaire().compareTo(e2.getSalaire()));
```

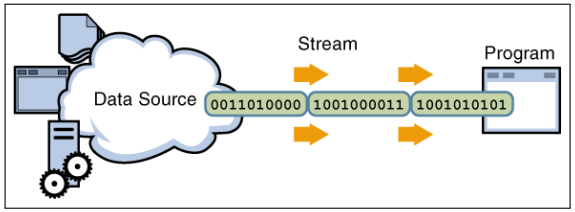
Les interfaces avec une seule méthode abstraite sont des *interfaces fonctionnelles*

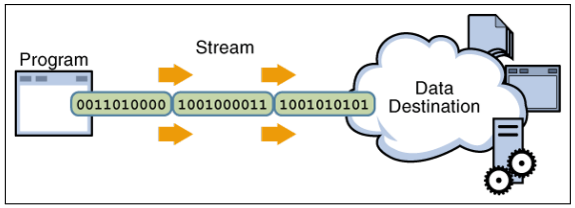
Les Flux

- Principe similaire à celui de C++
- Concepts équivalents pour lire/écrire dans des fichiers et même sur des sockets
- Package `java.io`
- Tout passe à travers un mécanisme de gestion des exceptions

InputStream et OutputStream

- Un flux représente une source d'entrée ou une destination de sortie.
- On lit/écrit un élément à la fois, dans l'ordre





I/OStream

- Différents types de données : simples octets, types primitifs de données, caractères, objets en général.
- On peut juste transférer les données ou les transformer et les manipuler de manière convenable
- La source (ou la destination) peut être un fichier, un périphérique, un programme informatique, une socket, etc.
- Deux types : flux d'octets et flux de caractères Unicode

Type des éléments

- Octets (bytes)
  - type le plus basique (très limité donc)
  - codés sur 8 bits
  - un octet + une table de caractères = un caractère
- Caractères (UTF-16)
  - représentation universelle des caractères
  - les plus utilisés

Quand vous lisez des octets, ce ne sont pas des caractères ! ! !

La Console

- `System.in` - une `InputStream` (avec Exceptions). On lit des octets et pas de caractères... Pour lire des caractères :

```
InputStreamReader cin = new InputStreamReader(System.in)
```

- `System.out` - une `PrintStream` (sous-classe de `OutputStream` qui ne lève jamais d'exception)
- `System.err` - mêmes opérations que pour `System.out`, pour afficher les messages d'erreurs

BufferedReader

- Associé à un `InputStreamReader` (pas un héritage !)
- Utilise un tampon (buffer) pour lire le `InputStreamReader` avant que le programme appelle `read()`
- Utilisation de la méthode `readLine()`

```
public static void main(String[] args) throws IOException{
    InputStreamReader cin = new InputStreamReader(System.in);
    BufferedReader bin = new BufferedReader(cin);

    String s;
    while(true){
        s = bin.readLine();
        System.out.println("Je lis : "+s);
    }
}
```

Scanner

- Package `java.util`
- Associé à un `BufferedReader`
- Fonctions `next()`, `nextInt()`, `nextDouble()`, ...
- Fonctions `hasNext()`, `hasNextInt()`, ...
- La chaîne est divisée en *tokens* suivant les espaces

```
public static void main(String[] args) throws IOException{
    InputStreamReader cin = new InputStreamReader(System.in);
    BufferedReader bin = new BufferedReader(cin);

    Scanner sin = new Scanner(bin);

    while(true){
        if (sin.hasNextInt())
            System.out.println("afficher entier : "+sin.nextInt());
        else if (sin.hasNext())
            System.out.println("afficher une string : "+sin.next());
    }
}
```

Fichiers

- Ouverture en mode caractères

```
new FileWriter("fichierAEcrire.txt");
new FileReader("fichierALire.txt");
```

- Ouverture en mode octets - fichiers binaires (images, fichiers .doc, etc.)

```
new FileOutputStream ("fichierAEcrire");
new FileInputStream("fichierALire");
```

Pour les fichiers binaires il y a une description de l'organisation des données binaires dans le fichier.

Le plus souvent il existe une librairie externe pour traduire le fichier en objets Java.

Mode caractères

Copie caractère par caractère :

```
public void copierCaracteres() throws IOException {
   try ( FileReader fluxEntree = new FileReader("fichier.txt");
         FileWriter fluxSortie = new FileWriter("fichier_sortie.txt") )
     {
         int c;
         // -1 indique la fin de la chaine
         while ((c = fluxEntree.read()) != -1) {
             fluxSortie.write(c);
         }
     }
}
```

Utilisation de l'instruction *try-with-resources* pour fermer correctement les flux après leur utilisation.

Mode caractères

Copie ligne par ligne en utilisant BufferedReader et PrintWriter :

```
public static void copierLignes() throws IOException {
    try (
    BufferedReader fluxEntree =
       new BufferedReader(new FileReader("fichier.txt"));
    PrintWriter fluxSortie =
       new PrintWriter(new FileWriter("fichier_sorti.txt"))
    )
    {
        String ligne;
        while ((ligne = fluxEntree.readLine()) != null) {
            fluxSortie.println(ligne);
        }
    }
}
```

Utilisation de l'instruction *try-with-resources* pour fermer correctement les flux après leur utilisation.

Fichiers de type Data

- Contiennent des types primitifs Java et des String
- Fichiers binaires (pas éditables à la main)
- Doivent être créés par un programme Java
- Compatibles multi-platforme
- Il faut faire attention à l'ordre de lecture et écriture (doit être le même)
- Les classes qui les manipulent : DataInputStream et DataOutputStream

Exemple

```java
public class Personne {
    String nom;
    String prenom;
    int age;
    double taille;
    boolean auChomage;

    public void sauvegarder(DataOutputStream str) throws IOException{
        str.writeUTF(nom);
        str.writeUTF(prenom);
        str.writeInt(age);
        str.writeDouble(taille);
        str.writeBoolean(auChomage);
    }

    // on lit dans le même ordre qu'on a sauvegardé
    public void lire(DataInputStream str) throws IOException{
        nom = str.readUTF();
        prenom = str.readUTF();
        age = str.readInt();
        taille = str.readDouble();
        auChomage = str.readBoolean();
    }

}
```

```java
private List<Personne> listePersonnes = new ArrayList<Personne>();

public void ecrireFichier(String nom){
    DataOutputStream sortie = null;
    try{
        sortie = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(nom)));
        for (Personne p : listePersonnes){
            p.sauvegarder(sortie);
        }
        sortie.close();
    }catch(IOException e){
        e.printStackTrace();
        new File(nom).delete(); //On détruit le fichier aleteré
    }
}

public void lireFichier(String nom){
    DataInputStream entree = null;
    try{
        entree = new DataInputStream(new BufferedInputStream(new FileInputStream(nom)));
        try{
            while(true){
                Personne p = new Personne();
                p.lire(entree);
                listePersonnes.add(p);
            }
        }catch(EOFException e){
            System.out.println("On a recuperé "+listePersonnes.size()+" personnes");
        }
        entree.close();
    }catch(IOException e){
        System.out.println("Erreur de lecture sur l'élément "+(listePersonnes.size()+1));
        e.printStackTrace();
    }
}
```

Au-delà des types primitifs

- En Java il est possible de lire et écrire des objets dans des fichiers.
- Utiliser `ObjectInputStream` et `ObjectOutputStream`
- Ils doivent implémenter l'interface `Serializable` :
  - Ajouter `implements Serializable` à la définition de la classe et tout sera possible à sauvegarder
  - Ajouter `transient` à la déclaration des champs qu'on ne souhaite pas sauvegarder (un peu plus compliqué)

Au-delà de ces transparents

`http://docs.oracle.com/javase/8/docs/api/`

`http://docs.oracle.com/javase/specs/`

`http://stackoverflow.com/`

Java Generics and Collections
M. Naftalin et P. Wadler ; O'REILLY, 2006.

Java in a Nutshell, 6th Edition, A Desktop Quick Reference
B.J. Evans et D. Flanagan ; O'REILLY, 2014.
disponible sur SAFARI pour les étudiants AMU :
`http://proquestcombo.safaribooksonline.com.lama.univ-amu.fr/`

JavaFX 8, Introduction by Example, second edition
C. Dea, M. Heckler, G. Grunwald, J. Pereda et S.M. Phillips ;
APRESS, 2014.