Conception et Programmation Objet Avancées
Introduction : rappels et compléments

Petru Valicov
petru.valicov@umontpellier.fr

https://github.com/IUTInfoMontp-M3105

2019-2020

Déroulement du cours

CMs en amphi :

1. Rappels notions d'objet, UML et Java (1-2 cours)
2. Quelques principes objet ($\approx$ 1 cours)
3. Modèles de conception ($\approx$ 3 cours)

21h de TD : une séance de 3h de TD par semaine en salle machine.

Enseignants :

sebastien.gagne@umontpellier.fr - TD
sophie.nabitz@univ-avignon.fr - TD
petru.valicov@umontpellier.fr - CM/TD

En TD c'est **VOUS** qui travaillez $\Longrightarrow$ Pas de correction à copier/coller $\Longrightarrow$ analyse/amélioration de **VOTRE** solution

Une question ? Posez là en amphi, en TD ou sur le **forum** dédié :

https://piazza.com/class/jzs4o7je7zm1a0

Déroulement du cours

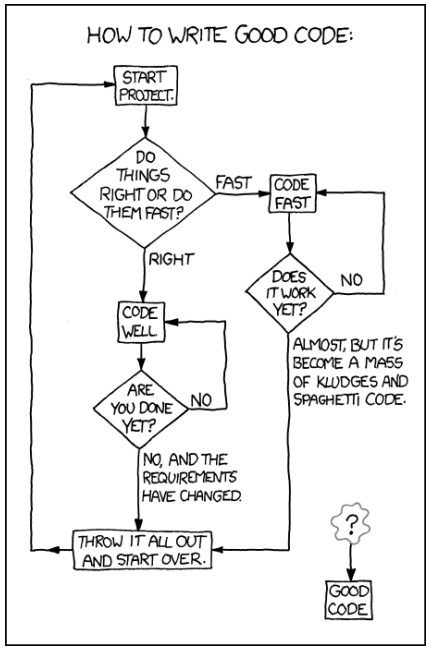
- **Contrôle des connaissances - examen**
  - octobre 2019
  - durée de 3 heures
  - questions de cours + exercices
  - documents autorisés : feuille A4 **manuscrite** recto-verso
  - **travail en TP** : pondération entre +2 et -2
- **Les outils** :
  - un langage de programmation : Java version $\geq$1.8
  - la notation UML (vue en Semestre 2)
  - un IDE : IntelliJ IDEA ou autre (Eclipse, NetBeans, etc.)
  - un outil de tests : JUnit (version 5)
  - un outil de versioning : Git et GitHub

Quelques références

1. E. Gamma, R. Helm, R. Johnson et J. Vlissides. **Design Patterns. Elements of Reusable Object Oriented Software**, édition Addison Wesley : 1995.
2. E. Freeman, E. Robson, B. Bates, K. Sierra. **Head First - Design Patterns**, édition O'Reilly : 2014.
3. J. Blosch. **Effective Java, 3rd Edition**, édition Addison-Wesley Professional : 2018.
4. R.C. Martin. **Clean Code - A Handbook of Agile Software Craftmanship**, édition Prentice Hall : 2008.

L'API officielle Java :
https://docs.oracle.com/javase/8/docs/api/

Une liste de bonnes pratiques et implémentations des patterns en Java : https://java-design-patterns.com/

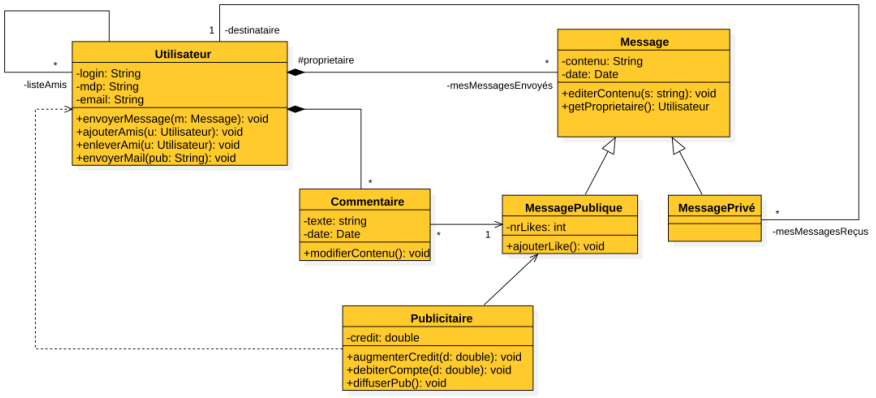


Bien coder

Savoir ce qu'on fait

Conception/Programmation Orientées Objet

- le programme/système est décomposé en **objets**
- chaque objet est responsable de son fonctionnement interne : **encapsulation**
- les objets communiquent en échangeant des **messages**



message = méthode = fonction/procédure

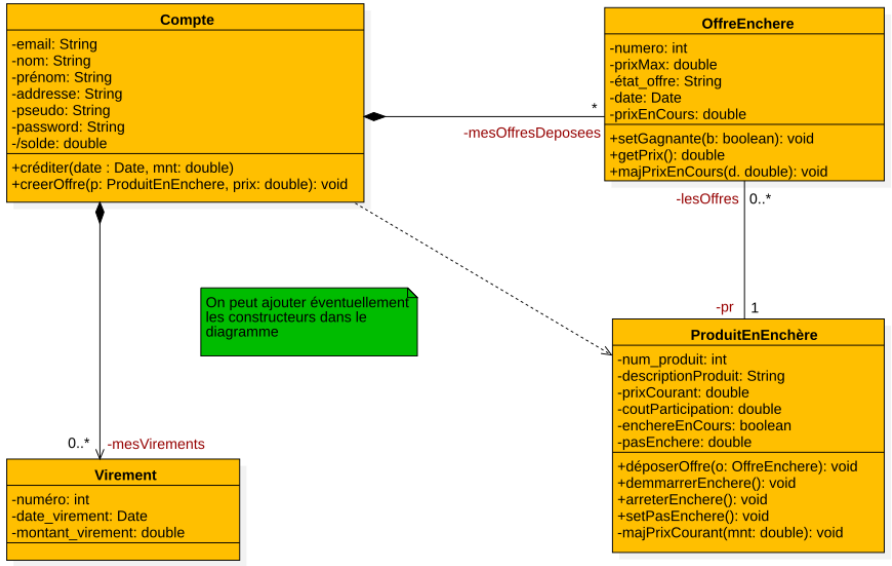
Quelques principes fondamentaux

1. **Encapsulation**
2. **Héritage**
3. **Polymorphisme**

} Les piliers magique de l'OO ! La base !

- **non-duplication du code** : **DRY** *vs* **WET**
- **couplage faible** : moins il y a de dépendances (liens) entre les classes, mieux on se porte
- **KISS** : plus c'est simple, plus c'est clair
- **YAGNI** : ajoutez du code quand vous en avez vraiment besoin

Un exemple d'application orientée objets



Définition

Objet = identité + état + comportement

- **identité** : `this` en `Java, C++, C#, PHP` ; `self` en `Python, Ruby, Perl` ; la clé primaire dans une BD...
- **état** : valeurs de l'ensemble d'*attributs* (ou données membres, ou champs)
- **comportement** : l'ensemble de *fonctions* membres ou *méthodes* d'instance (ou *messages* que l'objet peut accepter)

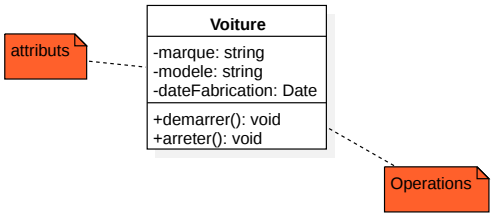
Classe *vs* Objets

Définition

Une classe est une description abstraite d'un ensemble d'objets "*de même nature*". Ces objets sont des **instances** de cette classe.

- La classe est un "**modèle**" pour la création de nouveaux objets.
- "**Contrat**" garantissant les compétences de ses objets.

Représentation `UML` :



Code `Java` :

```java
class Voiture {
    private String marque;
    private String modele;
    private LocalDate dateFabrication;

    public void demarrer() {
        /* implémentation */
    }

    public void arreter() {
        /* implémentation */
    }
}
```

Construction : `Voiture renaultQuatreL = new Voiture(...);`

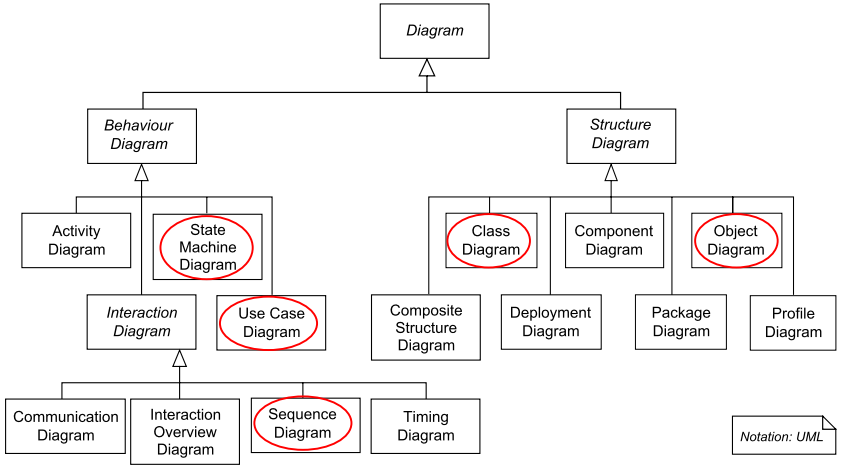
`UML` - Unified Modeling Language

Définition

`UML` est un **langage de modélisation** orienté objet qui permet de représenter (de manière graphique) et de communiquer les divers aspects d'un système informatique.

- Apparu au milieu des années '90
- Version actuelle : `UML 2.x` (standard ISO adopté par l'OMG)
- **langage de modélisation** $\neq$ **langage de programmation**
- C'est juste un ensemble de notations ayant comme base la notion d'*objet*

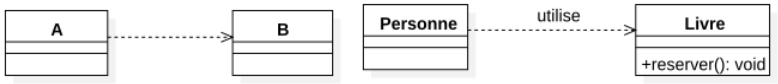
Les diagrammes `UML`



- Nous allons surtout nous intéresser aux diagrammes de classes.
- Révisez vos notes du cours de **Conception OO** du semestre dernier !

Dépendance

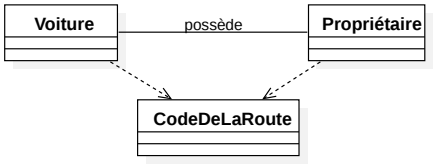
- Relation unidirectionnelle exprimant une dépendance sémantique entre deux classes
- Représenté par un trait discontinu orienté



- Généralement `A` dépend de `B` (on dit aussi `A` *utilise* `B`) si :
  - `A` utilise `B` comme argument dans la signature d'une méthode
  - `A` utilise `B` comme variable locale d'une méthode

Exemple : la modification du code de la route a un impact sur

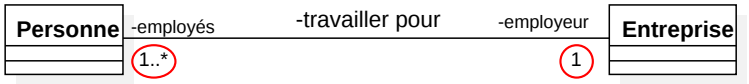
- l'attitude du conducteur
- des caractéristiques des voitures



Relation très générale : toutes les relations possibles entre les classes sont des dépendances

Relations entre classes : association

- relation sémantique entre les **objets** d'une classe
- possède un *rôle* à chaque extrémité
  - décrit comment une classe voit une autre classe à travers l'association
  - devient le nom d'un champ en `Java`
- multiplicité ou cardinalité

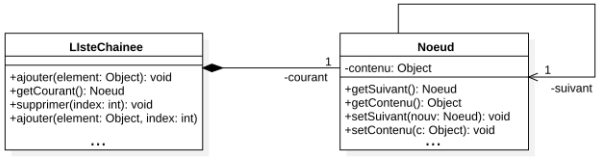


Une possible implémentation en `Java` :

```java
class Personne {
    private Entreprise employeur;
}
```

```java
class Entreprise {
    // un tableau d'employés
    private ArrayList<Personne> employés;
}
```


Relations entre classes : association




```java
public class ListeChainee {
   private Noeud courant;

   public void ajouter(Object element) {
       Noeud nouveau = new Noeud(element);
       if (courant == null)
           courant = nouveau;
       else {
           Noeud tmp = courant;
           while (tmp.getSuivant() != null)
               tmp = tmp.getSuivant();
           tmp.setSuivant(nouveau);
       }
   }

   public Noeud getCourant() { return courant; }

   public void supprimer(int index) { ... }

   public void ajouter(Object element, int index) { ... }

   /* d'autres attributs et méthodes */
}
```

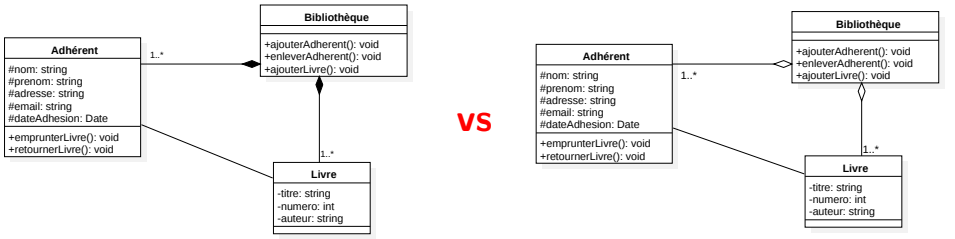
```java
public class Noeud {
   private Noeud suivant;
   private Object contenu;

   // constructeur
   public Noeud(Object contenu) {
      this.contenu = contenu;
   }

   public Noeud getSuivant() { return suivant; }

   public Object getContenu() { return contenu; }

   public void setSuivant(Noeud nouv) {
       suivant = nouv;
   }

   public void setContenu(Object c) {
       contenu = c;
   }

   /* d'autres attributs et méthodes */
}
```


Association spéciales : composition vs agrégation

- Dès que il y a la notion de contenance on utilise une agrégation ou une composition
- La composition est aussi dite **agrégation forte**

Comment décider entre la composition et l'agrégation ?

Si les composants ont une autonomie vis-à-vis du composite alors préférez l'agrégation. Mais tout dépend de l'application que vous développez...



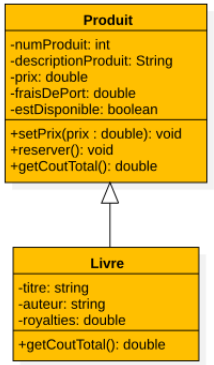
Relation d'héritage

- "Héritage" des propriétés des classes parents
  - La classe **enfant** est la classe spécialisée (ici *Livre*)
  - La classe **parent** est la classe générale (ici *Produit*)

- La classe enfant n'a pas accès aux propriétés privées

- La classe enfant peut redéfinir des méthodes de la classe mère : **polymorphisme**

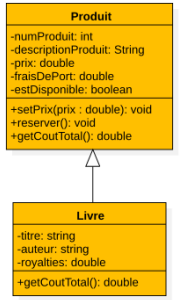
- **Principe de substitution** - toute opération acceptant un objet de type `Produit` doit accepter un objet de type `Livre`.



Attention : TOUTES les propriétés (publiques/privés/protégées) sont héritées dans les classes enfants.

Relation d'héritage : exemple

```java
public class Produit {
    private int numProduit;
    private double prix;
    private boolean estDispo = false;

    public Produit(int numProduit, double prix) {
        this.numProduit = numProduit; this.prix = prix;
    }

    public void setPrix(double prix) { this.prix = prix; }

    public void reserver() { estDispo = false; }

    public double getCoutTotal() { return prix; }
}
```

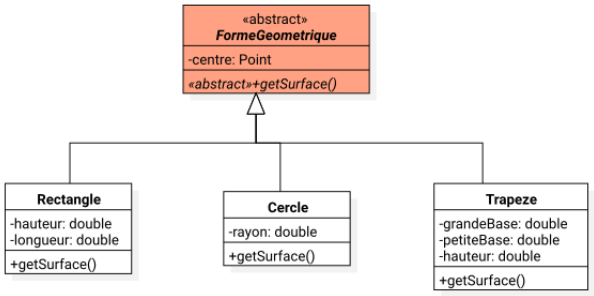



```java
public class Livre extends Produit {
    private String titre, auteur;
    private double royalties;

    public Livre(int numero, double prix, String titre, String auteur, double royalties) {
        super(numero, prix); // appel au constructeur de la classe mère (obligatoire)

        this.titre = titre; this.auteur = auteur; this.royalties = royalties;
    }

    @Override
    public double getCoutTotal() { // redéfinition de la méthode de la classe mère
        return super.getCoutTotal() + royalties;
    }
}
```


Héritage avec des classes abstraites

On peut factoriser les attributs et les méthodes dans une classe-mère sans définir son comportement intégralement :

1. parce qu'on n'en est pas capable
2. parce qu'on veut imposer cette tâche aux sous-classes



Héritage avec des classes abstraites

```java
public abstract class FormeGeometrique {
    private double centre;

    public abstract double getSurface();
}
```

```java
public class Rectangle extends FormeGeometrique {
    private double hauteur, largeur;

    public double getSurface() {
        return hauteur * largeur;
    }
}
```

```java
public class Cercle extends FormeGeometrique {
    private double rayon;

    public double getSurface() {
        return Math.PI * rayon * rayon;
    }
}
```

```java
public class Trapeze extends FormeGeometrique {
    private double grandeBase;
    private double petiteBase;
    private double hauteur;

    public double getSurface() {
        return (grandeBase + petiteBase) * hauteur / 2;
    }
}
```

```java
public class ClasseCliente {
    public static void main(String[] args) {
        FormeGeometrique forme = new Cercle();
        System.out.println( forme.getSurface() ); // la méthode de calcul de Cercle

        forme = new Rectangle();
        System.out.println( forme.getSurface() ); // la méthode de calcul de Rectangle
    }
}
```

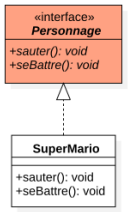
Intérêt des classes abstraites

Respect du *contrat* de la classe-mère : toutes les classes filles "concrètes" *savent effectuer* les opérations

- méthode abstraite = méthode "**promise**"
  - on s'attend à pouvoir calculer la surface de toute forme géométrique
  - ... sans vraiment savoir comment le faire dans le cas général
- méthode concrète (ordinaire) :
  - toutes les instances vont hériter son implémentation...
  - les sous-classes devront **s'y conformer** ou la **redéfinir**

Interfaces

Une **interface** est une classe abstraite qui n'a pas d'attributs et où **toutes** les méthodes sont abstraites.




```java
public interface Personnage {

   public void sauter();

   public void seBattre();

}
```

```java
public class SuperMario implements Personnage {

   public int sauter() {
      // implémentation concrète ici
   }

   public int seBattre() {
      // implémentation concrète ici
   }
}
```

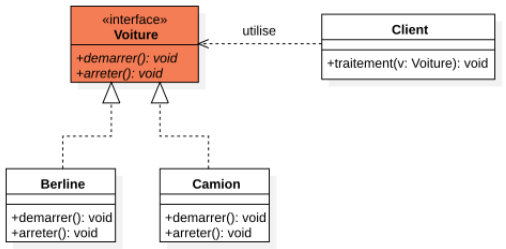
Ici `SuperMario` *implémente* l'interface `Personnage` (ou est une *réalisation* de l'interface `Personnage`)

À quoi sert une interface si c'est juste une classe abstraite ? ? ?

Indice : il n'y a pas d'*héritage d'état* contrairement aux classes abstraites.

Classe cliente d'une interface

- Quand une classe dépend d'une interface pour réaliser ses opérations, elle est dite **classe cliente de l'interface**
- On utilise une relation de dépendance entre la classe cliente et l'interface requise

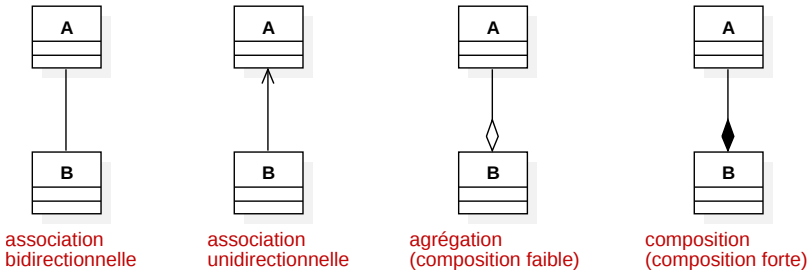


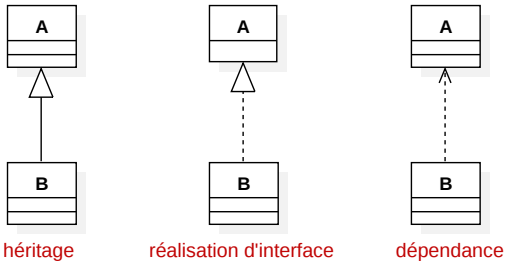

```java
public class Client {
    public void traitement(Voiture v) {
        v.demarrer();
        v.arreter();
    }
}
```

```java
public class Main {
    public static void main (String args[]) {
        Client objet = new Client();
        Voiture camion = new Camion();
        Voiture berline = new Berline();
        objet.traitement(camion);
        objet.traitement(berline);
    }
}
```

- Comme pour l'héritage, l'implémentation concrète de l'interface est masquée au client - **principe de substitution**

Les relations entre les classes

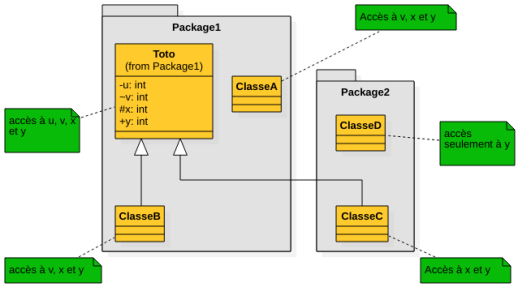


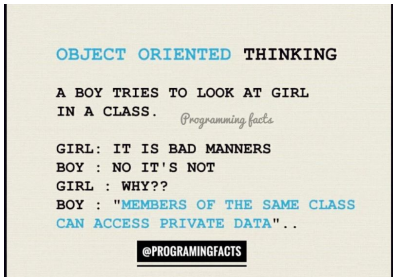


Visibilité

En Java, il y a 4 niveaux de visibilité :

```
class Toto {
    private int u; // seules les méthodes de la classe Toto y ont accès

    int v; // visibilité par défaut : accessible aux membres de la classe Toto
           // et de toutes les classes situées dans le même paquetage

    protected int x; // accessible aux membres de la classe Toto, de ses sous-classes
                     //et de toutes les classes situées dans le même paquetage

    public int y; // accessible aux membres de toutes les classes
}
```





Typage en Java

Java est un langage fortement typé :

- types *primitifs*
- types objets ou *références* (valeur par défaut `null`)

type primitif	taille en octets	valeur par défaut
boolean	1 bit	false
byte	1	0
short	2	0
long	8 bit	0
float	4	0.0
double	8	0.0
char	2	'\u0000'

Accès **par valeur** pour les types primitifs :

```
int x, y;
x = 2;
y = 3;

// on compare la valeur de x à la valeur de y
x == y;
```

L'accès aux types objets se fait **toujours par référence** :

```
Voiture x, y; // deux objets de type Voiture
x = new Voiture();
y = new Voiture();
x == y; // false : on compare l'adresse mémoire de x à l'adresse mémoire de y

Voiture v;
System.out.println(v); // affiche "null" - la valeur par défaut
```


Passage de paramètres en Java

En Java, le passage de paramètres se fait toujours **par valeur**.

Autrement dit, la méthode manipule une copie du contenu de la variable passée en paramètre.

1. si la variable est de type primitif : c'est une copie de sa valeur
2. si la variable est de type référence : c'est une copie de l'adresse indiquant où est situé l'objet concerné.

```
public static void main(String args[]) {
   int x = 10;
   methodeInutile(x);
   System.out.println(x); // 10

   Voiture v = new Voiture("berline");
   methodeInutileBis(v);
   System.out.println(v); // berline
}
```

```
public void methodeInutile(int v) {
   v = 30;
}
```

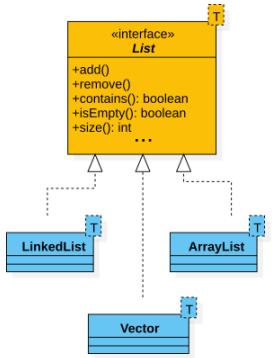
```
public void methodeInutileBis(Voiture v) {
   v = new Voiture("cabriolet");
}
```

L'objet référencé reste accessible et *modifiable* à travers ses méthodes.

Généricité

La **généricité** est un paradigme de programmation où les algorithmes sont conçus sur des *types inconnus* à l'avance.

- Abstraction du type des objets sur lesquels on travaille.
- On dit que la classe est **paramétrée**. Le type générique est spécifié à l'instanciation d'un objet de la classe.
- Utilisée partout où un type est nécessaire (type de paramètre/champ, constructeur).
- La généricité participe à la *type safety* (élimination des casts ...).



Exemple : `java.util.List`

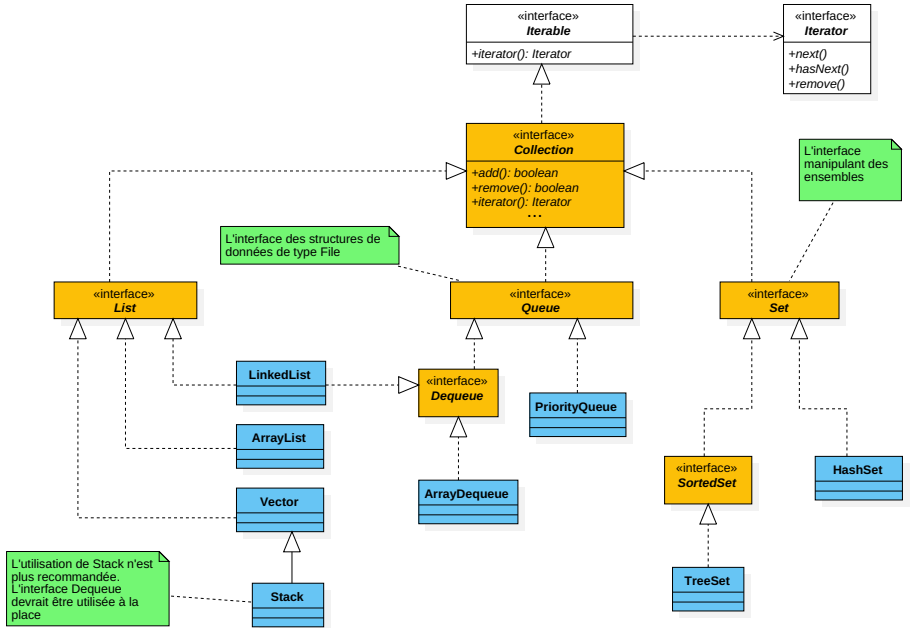
Java Collections Framework

- Contient différentes APIs de structures de données classiques (appelées "*collections*")
- Hiérarchie d'interfaces, d'implémentations et d'algorithmes
- Permet une meilleure interopérabilité avec les autres APIs du langage

Organisation :

- Dans le package `java.util`
- Deux interfaces de base :
  - `java.util.Collection` - structures *itérables*
  - `java.util.Map` - structures de la forme *tableau associatif*
- Une interface pour itérer sur les collections :
  - `java.util.Iterator`

java.util.Collection

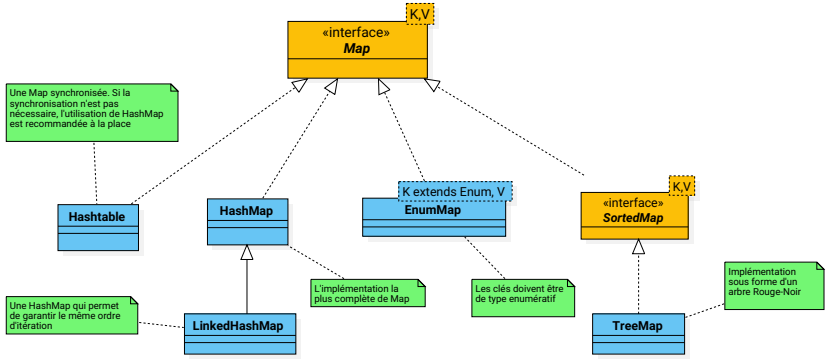


API de collections

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



En vrac

Beaucoup d'autres notions à réviser :

- la super-classe `Object` en `Java`
- L'emballage/déballage automatique (autoboxing)
- surcharge des fonctions
- la méthode `toString()`
- collaboration entre constructeurs
- les détails sur la généricité
- les interfaces `Comparable` et `Comparator`
- les collections en `Java`
- les tests unitaires (des beaux souvenirs du projet...)

Reprenez vos notes de cours de l'an dernier !