PROGRAMMATION ORIENTÉE OBJET

PROGRAMME

- Classe et objet
- Attribut et accesseur
- Les constructeurs
- Static attribut/ méthode
- Héritage et polymorphisme
- Classe abstraite
- interface

POO: PROGRAMMATION ORIENTÉ OBJET

La méthode orientée objet permet de concevoir une application sous la forme d'un ensemble d'objets reliés entre eux par des relations

Lorsque que l'on programme avec cette méthode, la première question que l'on se pose plus souvent est :

«qu'est-ce que je manipule ? »,

Au lieu de **« qu'est-ce que je fait ? ».**

L'une des caractéristiques de cette méthode permet de concevoir de nouveaux projets à partir de structures existantes.

On peut donc réutiliser les objets dans plusieurs applications.

La réutilisation du code est un argument déterminant pour venter les avantages des langages à objets.

Pour faire de la programmation orientée objet il faut maitriser les fondamentaux de l'orienté objet à savoir:

Objet et classe

Héritage

Encapsulation (Accessibilité)

Polymorphisme

Interface

Des classes déjà vues

- Integer (contrairement à int, version classe des types classiques), Random, Scanner…
- On peut créer ses propres classes
- Définition :

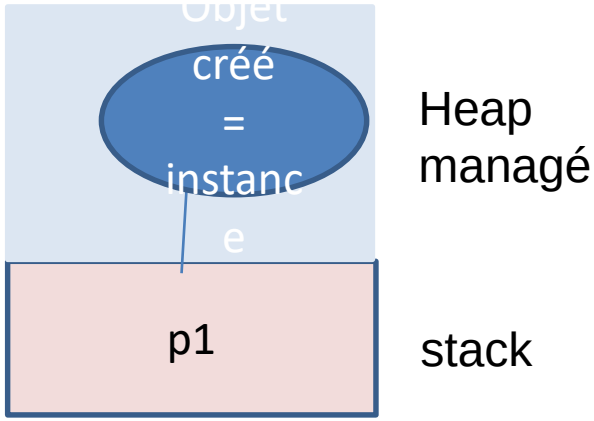
Une classe est une entité qui définie un nouveau type

Exemple : class Personne, qui permet de définir des personnes avec un nom, et prénom, un age.
Une classe est constituée d attributs , constructeurs , méthodes

On va pouvoir définir des personnes particulières, dites des instances

COMMENT SE CODE UNE CLASSE ?

```
String str1;   // déclaration
str1="toto"; //affectation

Personne p1; //déclaration :
Personne p1 = null ;
p1=new Personne(); //instanciation
p1=null; // objet non référencé
```



VOCABULAIRE

- p1 est une référence vers l'objet créé
- Ce qui est crée est une instance , un objet en mémoire
- Classe utilisateur / classe métier

CRÉER SES CLASSES : 1ER EXEMPLE GUIDÉ

- Créer une nouvelle classe (sous eclipse) sans cocher main, donner un nom commençant par une majuscule
- (classe métier ex:Personne) :

  On obtient une classe valide

Créer une classe TestPersonne contenant un main

(classe de test)

Contenu de TestPersonne.java :

```
public class TestPersonne {

public static void main(String[] args) {

Personne p= new Personne();
//declaration et instanciation
p.nom="toto";
p.prenom="titi";
p.age=10; //initialisation

System.out.println(p.nom+" "+p.prenom+""+p.age);

}
}
```

Contenu de Personne.java :

```
public class Personne
 {
String nom;
String prenom;
int age;
  }
```

EXPLICATION DU CONTENU DES CLASSES

Que trouve-t-on dans une classe ?

- Attributs (aussi appelés data member ou variables de classe/d'instance) à différencier d'une variable locale (d'une méthode par exemple)
  Les attributs ont une valeur par défaut(String:null,int:0 etc…)
- Méthodes
- Constructeurs

```
package debut;

public class Personne {
String nom;
String prenom;
int age;
}
```

attributs

```
Personne p1= new Personne();
p1.nom="toto";
p1.prenom="titi";
p1.age=10;

Personne p2= new Personne();
p2.nom="aa";
p2.prenom="bb";
p2.age=20;
```

DES ERREURS À CONNAÎTRE

```
p1=null;
System.out.println(p1.nom+" "+p1.prenom+" "+p1.age);
```

Déclenche un erreur d'exécution (compile bien) :

```
Exception in thread "main" java.lang.NullPointerException
at debut.TestPersonne.testinstanciation(TestPersonne.java:22)
at debut.TestPersonne.main(TestPersonne.java:6)
```

Signifie que l'on tente d'utiliser un objet non-instancié :
On ne peut pas y accéder !

DES VALEURS PAR DÉFAUT

```
Personne p1= new Personne();
System.out.println(p1.nom+" "+p1.prenom+" "+p1.age);
```

On obtient à l'exécution l'affichage :

null null 0

Valeur par défaut de int

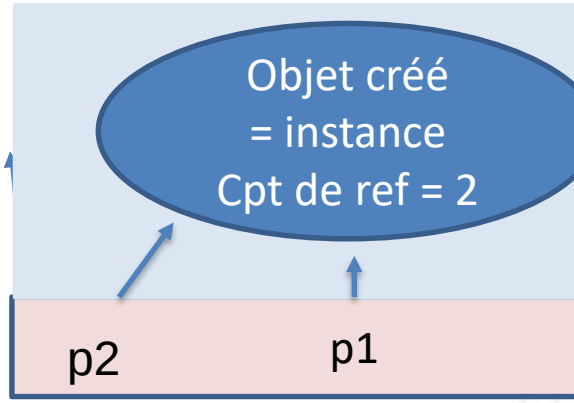
Valeur pas défaut pour String

Lors de l'instanciation, les attributs prennent leur valeurs par défaut

RÉFÉRENCES

```
Personne p1= new Personne();
p1.nom="toto";
p1.prenom="titi";
p1.age=10;
Personne p2= p1; // crée une deuxième ref vers le même objet
p1=null; // l'instance n'est pas supprimé : il reste encore une réf
p1=p2; // p1 re pointe vers l'instance
System.out.println(p1.nom+" "+p1.prenom+" "+p1.age);
System.out.println(p2.nom+" "+p2.prenom+" "+p2.age);
```

Fonctionne !
Attention : Ca ne crée pas de clone de l'instance !
Schéma mémoire après `Personne p2= p1;` :



UNE MÉTHODE DANS NOTRE CLASSE


```
package debut;

public class Personne {
String nom;
String prenom;
int age;

void affiche(){
System.out.println(nom+" "+prenom+" "+age);
}

}
```

CONSTRUCTEURS

Constructeur : deux rôles :
- Crée l'objet en mémoire
- Affecte les valeurs

```
Personne p;
p= new Personne();
```

On a pourtant pas écrit ce constructeur. C'est parce qu'il existe différents types de c

1. Constructeur par défaut de la JVM (= constructeur implicite) : si aucun constructeur n'est codé, qui permet de créé un objet et donne aux attributs la valeur par défaut de leur type, on a donc une instance utilisable. C'est ce qui se passe dans l'exemple ci-dessus.
   N'existe que s'il n'y a aucun constructeur dans la classe
2. Constr. par défaut : ex. par slide suivant
3. Constr. d'initialisation : lorsqu'il y a au moins un paramètre en entrée
4. Constr. par copie

UN CONSTRUCTEUR PAR DÉFAUT

- Un constructeur porte toujours le même nom que la classe
- Ressemble à une méthode, mais n'a aucun type retour
- Un constructeur **par défaut** ne prend pas de paramètre

Exemple :

Dans la classe Personne :

```
Personne(){
}
```

Autre version :

```
Personne(){
nom="Dupont";
}
```

Donnera la valeur « Dupont » au nom lorsque l'on appellera notre constructeur (en écrasant le null qui arrive automatiquement à la cration de l'objet

Dans classe de test :

```
Personne p;
p= new Personne();
```

C'est maintenant notre constructeur qui est appelé
(ici fait exactement la même chose)

ETAPE D'UNE INSTANCIATION

1. Création + mettre les valeurs par défaut
2. Initializer (remplace les valeurs par défauts) par celles définies au niveau des attributs (même dans le cas du constr. JVM), les valeurs ne peuvent être que des constantes.
3. Execution du contenu du constructeur (dans lequel on peut faire un traitement)
4. Affectation de l'extérieur depuis la référence

```
public class Personne {
String nom = "Defaut";
String prenom;
int age;

Personne(){
nom="Dupont";
}
}
```

Personne p1= **new Personne();**
p1.nom="toto";

Valeurs prises par p1.nom selon les étapes :

1. Null
2. « Defaut »
3. « Dupont »
4. « toto »

CONSTRUCTEUR VS MÉTHODE

- Le constructeur n'est appelé qu'une seule fois lors de la vie d'un objet: lors de sa création.
- Une méthode peut être appelé à différent moment de la vie d'une objet, autant de fois que l'on veut.

Exemple :

```
Personne p = new Personne();
p.traitement();
p = new Personne();
```

crée une nouvelle instance et change l'objet auquel p fait référence. L'instance précédent n'étant plus référencée elle sera supprimée par le G.C.

CONSTRUCTEUR D'INITIALISATION

```
Personne(String n){
nom=n;
}
```

Au moins un paramètre : constructeur d'initialisation

Lorsqu on code un constructeur quelconque, le constructeur implicite n est plus disponible,

On peut surcharger les constructeurs selon le même principe de la surcharge classique d une méthode

CONSULTER LES CONSTRUCTEURS DISPO

Apres un new Personne ctrl+space permet de voir la liste des constructeurs

- Paramètres p, a, n etc pas très parlant, on voudrait pouvoir donner des noms parlants comme prenom, age, nom.
- Risque de confusion lorsque les noms d'attributs et les noms de parametres sont les meme
- solution simple le « this »

```
Personne(String nom){
this.nom=nom;
}
```

Celui passé en paramètre

- But : réutilisation sous toutes ses formes
- Un constructeur peut faire appel à un autre (si un bout du traitement est commun, pour pas avoir à le refaire

```
Personne(String nom, String prenom){
this.nom=nom;
this.prenom=prenom;
}

Personne(String nom, String prenom, int age){
this(nom,prenom);
this.age=age;
}
```

Limite : on ne peut appeler qu'un seul constructeur (qu'un this) et forcément en première position

VARIANTE DE NOMMAGE

- Certains package utilise des attributs préfixé de _ (par exemple int _age) pour éviter les this (dans des cas plus ou moins restreints et normés)
- Important : garder des noms de variables clairs pour l'utilisateur

CONSTRUCTEUR PAR COPIE

- Permet à l'utilisateur de cloner un objet :
  on a une instance en mémoire, que l'on veut conserver, et en créé une autre identique mais que l'on pourra manipuler sans modifier la première (instance indépendante)

```
Personne(Personne x){
nom=x.nom;
prenom=x.prenom;
age=x.age;
}
```

Prend en paramètre une instance

Met les attributs à la même valeur que dans l'instance donnée

TABLEAU DE PERSONNES : DÉCLARATION

```
Personne[] tab = new Personne[3];
```

Attention : il ne s agit pas d un constructeur de Personne !
Lorsqu on crée un tableau de 3 personnes, par defaut il ya des null

```
static void test2(){
Personne[] tab = new Personne[3];
Personne p1=new Personne("A", "B", 41);
tab[0]=p1;
p1.affiche();
tab[0].affiche();
p1=null;
tab[1]= new Personne("C", "D",20);
tab[2]= new Personne("E","F",30);
}
```

```
for (Personne p : tab){
if (p!=null) p.affiche();
}
```

On aurait bien sur pu faire une boucle classique for plutôt que for each

Permet de boucler sur tout un tableau de personne pour effectuer un traitement voulu (ici, affiche) sans planter si le tableau est « trop grand », ie, si certaines cases ne sont pas des ref vers des instances effectivement créée.

APARTÉ : MÉTHODE CLONE

- Une autre manière de construire un élément similaire que le constructeur par copie.
- Permet l'utilisation du constructeur par copie via instance.clone() plutôt que comme un constructeur
- Redéfinition d'une méthode clone existant dans Object

```
public Object clone(){
return new Personne(this);
}
```

Renvoie un Object, doit donc être casté

Appel au constructeur par copie, doit être défini dans la classe

Autour de la méthode affiche

- On a toujours (norme), une méthode nous permettant de nous renseigner sur les attributs de la classe (comme affiche ici)
- Deux problèmes dans notre classe affiche :
  1. On a décidé de l'appeler affiche mais le nom n'est pas normalisé, on voudrait une norme commune à toute les classes pour pouvoir facilement savoir comment appelé ce genre de méthode sur tout type de classe
  2. on voudrait pouvoir l'utiliser dans tout contexte (ici, limité à un projet de type console, car on a utilisé System.out.println). On va ensuite éviter les println dans les classes métiers pour pouvoir être utilisé dans différents cadre.

La méthode toString


```
public String toString(){ //il faut exactement cette signature
return nom+prenom+age; // on pourrait avoir un autre format
}
```

- N'utilise pas de println : renvoie un String
- Nom normalisé, toute les classes ont une méthode toString
- Pas de norme sur le formatque l'on renvoie
- Générable automatique avec eclipse (source, generate toString). On peut faire des constructeurs idem (et d'autres)

Exemples d'utilisation dans la classe utilisateur (qui elle est dépendante du support)

```
for (Personne p : tab)
if (p!=null) System.out.println(p.toString());
```

Appel implicite d'un toString dans `System.out.println :`

```
Personne p1=new Personne("A", "B", 41); System.out.println(p1);
```

Fonctionne ! (dans les autres cas il faut appeler explicitement la méthode)

MÉTHODE CREATE()

- Scenario : un utilisateur préfère utiliser une méthode create plutôt que des constructeurs

```
Personne create(String nom, String prenom, int age){
return new Personne(nom, prenom, age);
}
```

A surcharger avec tous les types de constructeurs

Utilisation :

```
static void test2(){
        Personne pivot=new Personne(); //il faut une instance
quelque part pour     //pouvoir utiliser des methodes
        Personne p1=pivot.create();
        Personne p2=pivot.create("blab", "hj");
        Personne p3=pivot.create("haha", "moi", 4);
}
```

Instance anonyme

- Exemple :
  new Personne();

  instance sans référence qui pointe dessus, donc sont compteur à 0, sera supprimée par le G.C.

INSTANCE ANONYME

```
static void test3(){
System.out.println(new Personne().getHello());
}
```

```
Dans la classe Personne:
String getHello(){
return "hello";}
```

déclenche une méthode une fois.
On peut aussi mettre une ref dessus si on pense faire plusieurs getHello (par ex.) : mieux dans ce cas, ça évite de faire plein de création d'instance
Si par contre c'est très rare, mieux vaut une instance anonyme (et donc supprimée vite, traine pas en mémoire)

private:

L'autorisation **private** est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes et classes internes).

Les éléments déclarés **private** ne sont accessibles que depuis la classe qui les contient.

Ce type d'autorisation est souvent employé pour les variables qui ne doivent être modifiées ou lues qu'à l'aide d'un *getter* ou d'un *setter*.

public:

Un membre public d’une classe peut être utilisé par n’importe quelle autres classe.

En UML le membres public sont indiqués par le signe +

protected:

Les membres d'une classe peuvent être déclarés **protected**.

Dans ce cas, l'accès en est réservé aux méthodes des classes appartenant

au même package

aux classes dérivées de ces classes,

ainsi qu'aux classes appartenant aux mêmes packages que les classes dérivées.

Autorisation par défaut : package

L'autorisation par défaut, que nous appelons ***package***, s'applique aux classes,

interfaces, variables et méthodes.

Les éléments qui disposent de cette autorisation sont accessibles à toutes les méthodes des classes du même package.

Visibilité (private, public, protected, default/package)

- Devant toute méthode, attribut, constructeur il faut mettre private ou public (on verra les autres cas plus tard) mettons tout en public :

```
public class Personne {
public String nom = "Defaut";
public String prenom;
public int age;

public Personne(){
nom="Dupont";}

public String toString(){
return nom+prenom+age;
}
public String getHello(){
return "hello";
}
}
```

QUE VEUT DIRE PUBLIC/PRIVATE ?

- public : Accessible de l'extérieur de la Classe depuis une instance

  Exemple : p.methode(); dans une classe test : j'ai pu accédé à methode

- private : non accessible depuis l'extérieur de la classe, la méthode n'est pas visible depuis d'autres classe, comme si elle n'existait pas.

  Une tentative d'utilisation d'une telle méthode depuis l'exterieur déclenche une erreur (de compilation):

- N'a de sens qu'à l'extérieur de la classe, dans une même classe toutes les méthodes ses voient les unes les autres.

Quels cas d'utilisation ?

- Constructeurs public sauf exceptions (une bonne raison de bloquer un ou plusieurs constructeurs)
- Méthodes : cela dépend
  - Méthodes « centrales » publiques, celle comme point d'entrée nécessaire au traitement demandée par l'utilisateur
  - Méthode auxilliaires existantes pour factoriser du code, utilisée juste par d'autres méthodes de la classe : private
- Attributs : private

  Pourquoi ? Sensible, mieux vaut y accéder via des méthodes pour pouvoir conditionner les modifications

Les autres ?

- protected : lié à l'héritage
- Default(package) : En java, cela correspond à la visibilité **package** : agit comme public dans un package et private dès qu'on en sort

  A ne pas utiliser en général : Problème lors de la livraison/mise en production

ACCESSEURS (GETTER ET SETTER)

- Pour répondre au fait qu'avec des attributs private on veut quand même y accéder.

```
public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}
```

Utilisation :

```
Personne p0=new Personne();
p0.setAge(25);
System.out.println(p0.getAge());
```

On peut controler les données avant d'accepter la modification :

```
public void setAge(int age) {
if (age>=0&&age<120) this.age = age;
}
```

On peut utiliser les setters dans les constructeurs pour garantir la cohérence des données :

```
public Personne(String nom, String prenom, int age){
this(nom,prenom);
setAge(age);
}
```

LA METHODE TOSTRING

- Méthode public , présente dans la classe pour retourner les informations sur les attributs de la classe
- Permet d éviter de conditionner la classe , en évitant l utilisation des « println»
- Public String toString(){return nom+prenom+age;}

EXERCICE : CODER ET TESTER LA CLASSE VILLE

Ville :

- nom
- pays
- capital
- nbHabitant (en Million)
- Categorie (« petit », « grand »)
- Is Capital (true- false)
- Constructeurs
- toString()

NA : non-affecté

public String comparer (Ville v2) : renvoie par ex. « Lille est plus petite que paris »

UTILISATION DE TYPES COMPLEXES/ASSOCIATION

Une classes est appelée aussi un type complexe

Une association est un lien entre deux classes.

On peut donc avoir dans une classe un attribut de type complexe

TYPE INTRINSÈQUE VS TYPE COMPLEXE

class Adresse

Composé de numéro , rue , ville

Un constructeur d initialisation

Une méthode toString

class Personne

Composé de age, nom , prénom , status et une adresse

Un constructeur d initialisation

Une méthode toString

Un setAge() et computeStatus()

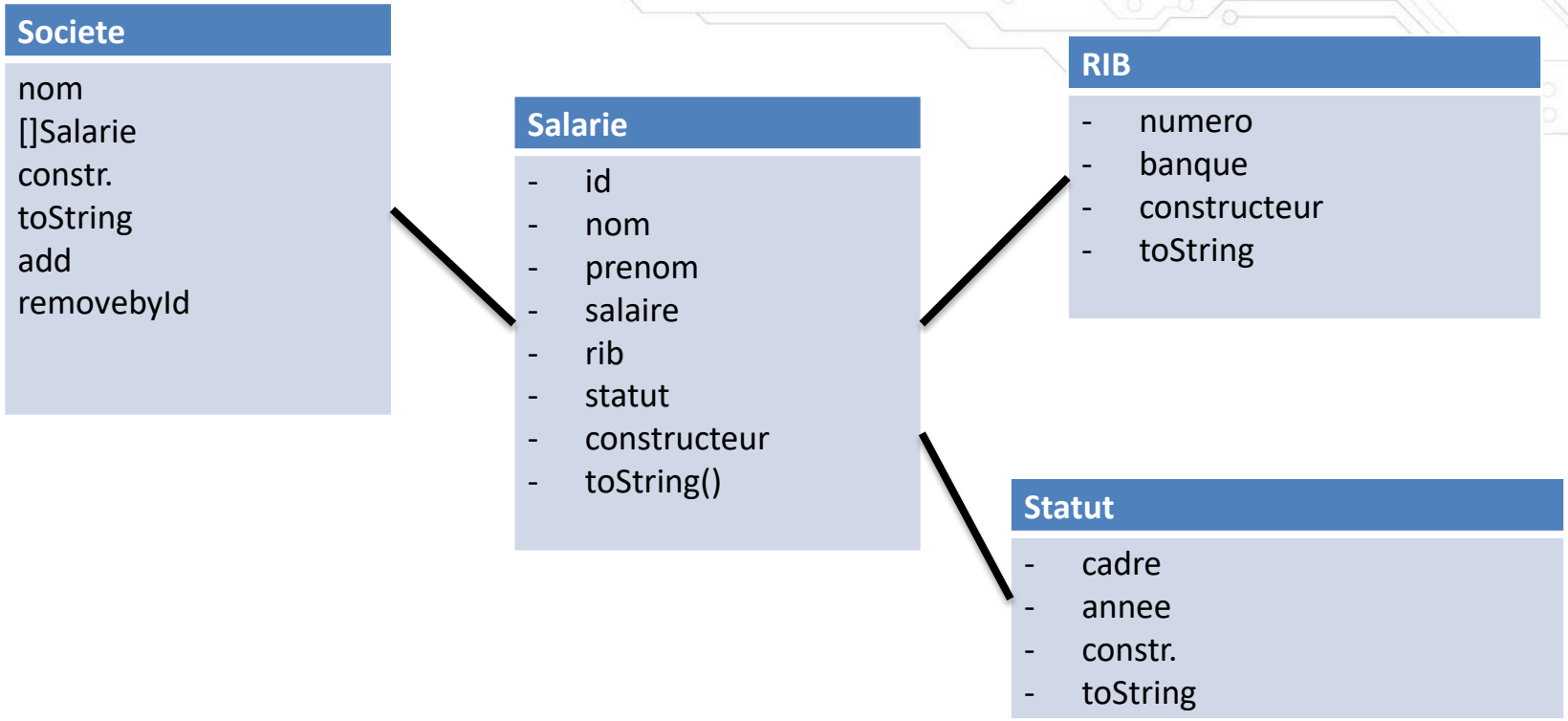
class Test-→main

Adresse a1=new Adresse(10, » rue untel», » paris»);

Personne p=new Personne(« x », »y »,a1);p,setAge(20);

SOP(p.toString()); -----→ x y 20 ans majeur 10 rue untel paris

Diagramme des classes à coder



METHODE STATIC

- Méthode non static= méthode d'instance, il faut créer une instance pour l'utiliser dessus
- Méthode static = méthode de classe, ne dépend pas de l'instance, pas besoin de créer d'instance pour l'utiliser
- Ne s'appellent pas de la même manière :

```
public class A {

public void m1(){
System.out.println("m1");
}

public static void m2(){
System.out.println("m2");
}
}
```

```
public class Test {

public static void main(String[] args)
A.m2();
A monA=new A();
monA.m1();
}
}
```

EXEMPLES DANS LES BIBLIOTHÈQUES JAVA

- Des méthodes d'instances (méthode non static):
  - nextInt de Scanner :

```
Scanner sc = new Scanner(System.in);
sc.nextInt(); // methode d'instance
```

- Des méthodes de classe (méthode static):
  - pow de Math :

```
Math.pow(10.0, 2.5);//methode de classe
```

- Class loader : le process qui charge les classe en mémoire, au début du programme. Charge les méthodes static.
- Les méthodes static et non static peuvent appeler les méthodes static
- Les méthodes static ne peuvent pas appeler les méthodes non static (sans créer d'instance)

ATTRIBUTS STATIC

- Partagé par toutes les instances, créer lors du chargement de la classe en mémoire. Visible par toutes les méthodes. Partagée par toutes les instances, toutes peuvent la lire et l'écrire.
- Première utilisation : un attribut commun que l'on a pas à dupliquer pour gain de place
- Deuxième utilisation : compter des instance.

  A incrémenter à chaque appel d'un constructeur.

BLOC STATIC

static{
instructions;
}

Dans une classe, en dehors de toute méthode.

- Automatiquement appelé par le class loader au démarrage de l'application
- Ne peut pas être appelé
- Conséquence : est appelé une et une seule fois dans la vie de l'appli.
- Permet d'effectuer des traitements (pour initialiser des attributs statiques)

Remarque : s appelle constructeur static en .NET mais a le même rôle

CONSTANTES

- Attributs fixés à une constante, non modifiable
- Mot clef : **final**
  - Empêche la modification, donc peut être public
  - Dans beaucoup de cas sera aussi static
- Exemple : PI dans la classe Math

```
public static final int age_min=0;
```

ASSOCIATION : 2 TYPES

- Deux types d'association entre Classes :
  - Association faible : la clase associée peut servir ailleurs

  ex. entre RIB et salarié, on a une association faible car on pourrait utiliser le RIB ailleurs (par ex. pour une classe pole emploi), n'est pas à supprimer lorsque l'on supprime l'instance correspondante.
  - Association forte : l'instance ne sert que pour l'instance à laquelle elle est associée

  Ex : entre statut et salarié, statut n'a pas de sens sans le salarie associé
- Incidence sur la mémoire : les instances associées de manière faibles sont conservées lors de la suppression de l'instance qui les utilise, alors qu'elles sont supprimées pour les liens forts.
  Ex : si je supprime une instance de salarie, je voudrais que son rib soit conservé mais son statut supprimé
- Symboles : losange vide pour association faible, losange plein pour association forte

Quel type de lien?

Email

destinataire
Content
File
Constr.
toString

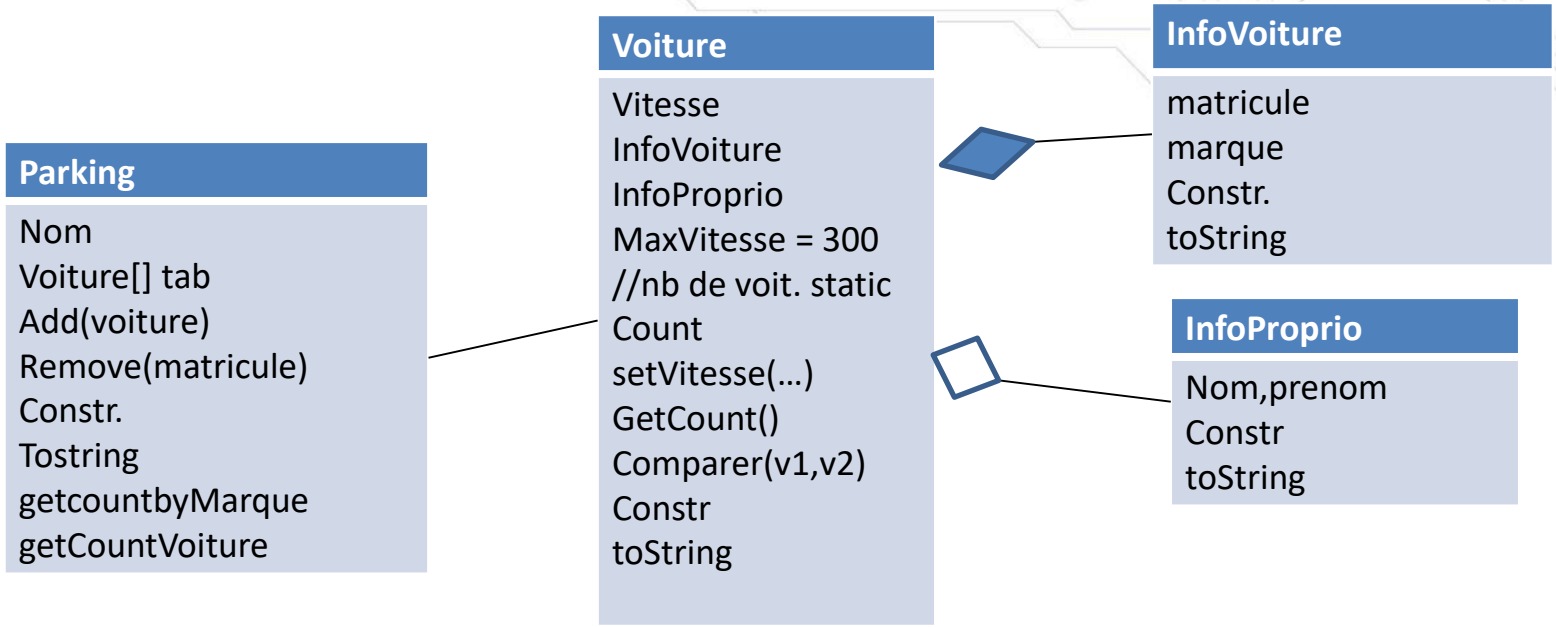
Content

Titre
message
Constr.
toString

File

Nom
taille
Constr
toString

TP


Parking
Nom
Voiture[] tab
Add(voiture)
Remove(matricule)
Constr.
Tostring
getcountbyMarque
getCountVoiture
Voiture
Vitesse
InfoVoiture
InfoProprio
MaxVitesse = 300
//nb de voit. static
Count
setVitesse(…)
GetCount()
Comparer(v1,v2)
Constr
toString
InfoVoiture
matricule
marque
Constr.
toString
InfoProprio
Nom,prenom
Constr
toString

Design Pattern

DÉFINITION

- Une solution à une problématique récurrente dans la programmation objet
- Les design pattern sont des solutions pratiques à des problématiques identifiées
- Solutions toutes prêtes à appliquer/permet de normaliser le code

Les design pattern du GOF

Gof = gang of four (4 dev. qui ont listé et baptisé un ensemble de design pattern régulièrement utilisé)

Les design pattern les plus connus, ils sont au nombre de 23

Ils ont un nom et sont classés par catégorie

Un (d'autres existent) site recommandé : dofactory.com

- Dans ref. Guides, .NET design Pattern
- Codé en .NET

Certains sont très simple, d'autres très complexes (aussi bien côté des problématiques que des solutions)

Intéressant d'en connaitre, de pouvoir les trouver facilement = gain de temps + maintenance évolutive

Exemple 1 :Le pattern Singleton

Catégorie : pattern de création (créer des instances, etc)

Classe qui ne permet qu'une instance et pas plus

Constructeur en privé : on empêche donc l'utilisateur de créer des instance

une méthode getInstance() qui ne crée l'instance que si l'on en a pas encore créé

(variable static donc commune à toute instance) :

on peut donc en créé une, et pas plus.

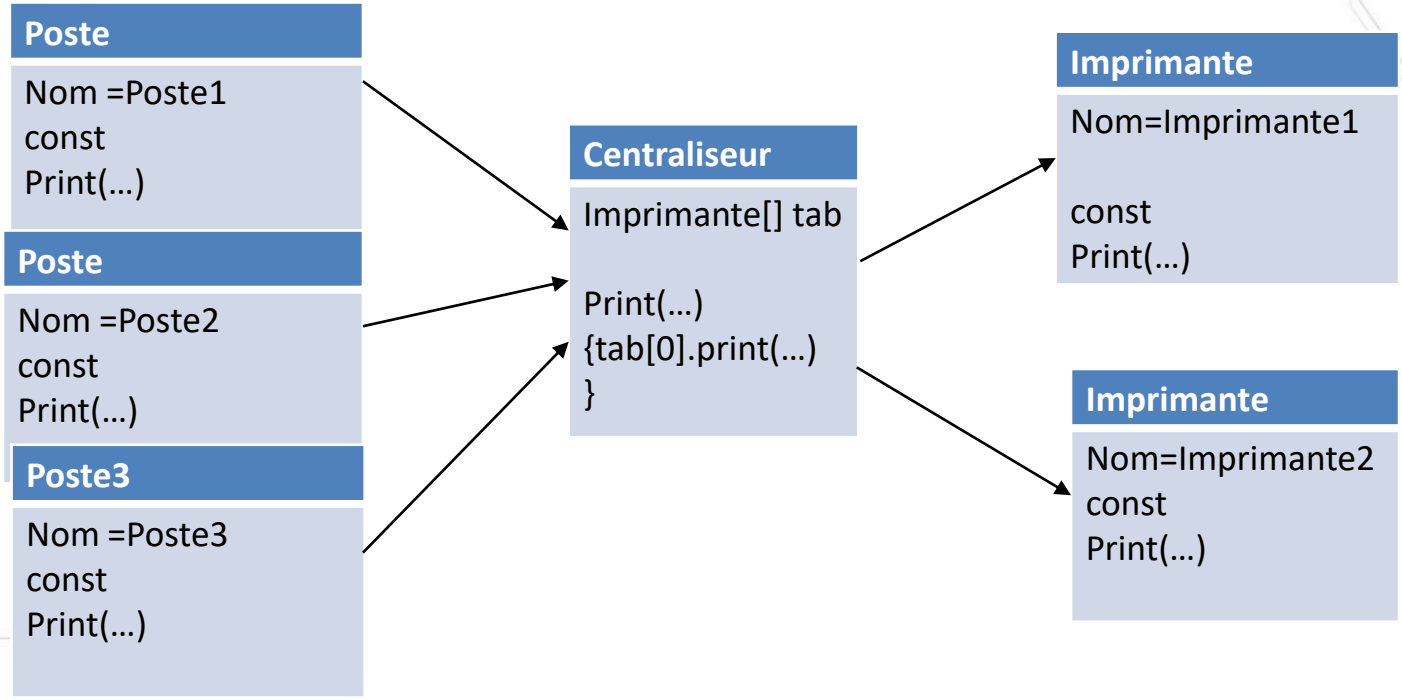
```
public class Singleton {
private static Singleton instance = null;
private Singleton(){}
public static Singleton getInstance(){
if (instance==null) instance = new Singleton();
return instance;
}
public String m(){return "m";}
}
```

Utilisation :

```
public class Test {
public static void main(String[] args) {
Singleton s1=Singleton.getInstance();
Singleton s2=Singleton.getInstance();
System.out.println(s1.m());
System.out.println(s1==s2);
//s1 et s2 sont deux références vers la même instance
} }
```

EXERCICE: A FAIRE

- **3 postes à relier à deux imprimantes, lors d'une demande d'impression un centraliseur choisit à quelle imprimante envoyer l impression (load balancing)**



Exemple 2: Pattern Observer

- Une problématique : deux entités un professeur et plusieurs élèves, tous ont un nom. Les élèves passent un examen, un projet collectif qui donnera une même note à tous, qui correspond à une note devant être accessible personnellement. Au début, la note n'est pas affectée, puis à un moment le professeur donne la note. On veut que les élèves soient notifiés au moment où la note est attribué. On veut éviter que l'élève ait à demander régulièrement.
- add/remove = attach/detach
- updateAll= notifAll, update=notif

CRÉATION D UN JAR, PARTAGER SES PROJETS

- Création du JAR : (concepteur/dev)
  - Création du projet, avec package et classe (+ tester)
  - Génération du jar et export :
    - Sur eclipse : clic droit sur le projet/export/java/jar
- Création d'un projet qui utilise le JAR : (utilisateur/chef de projet)
  - création d'un projet avec le main
  - Rajouter le JAR
    - Clic droit sur projet / properties/ java build path/onglet libraries/ add external jar
    - Le jar est rajouté dans « références libraries » dans le projet
  - Import du package : **import monpackage.*;**
  - Utiliser la classe/ la méthode

ENUMÉRÉ

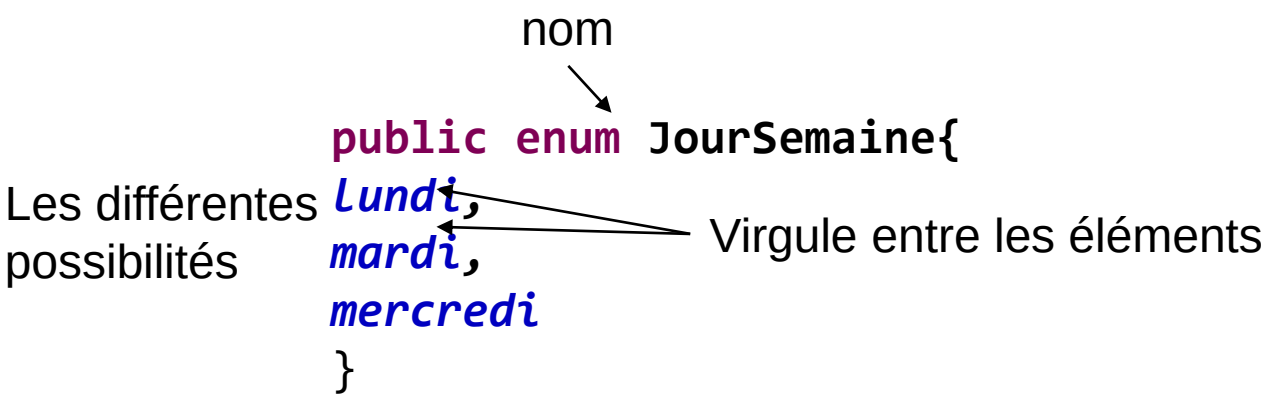
TYPE ENUM : L ENUMÉRÉ

C’est une liste d’options.

Ce n’est pas une classe (même si se compile en.class)

Type qui permet de manipuler des catégories de manière pratique (plus facile à gérer qu’un tableau par ex.)

Constitué d’une liste de possibilité



- Chaque option peut être associée à une valeur

  (ressemble a une classe constructeurs, méthodes etc.)
- Instances crées au chargement de l'enum, on ne peut pas en créé d'autres, appelle le constructeur défini avec les valeurs définis dans l'enum

```
enum Langage {
java(1000), csharp(2000), cpluspus;
private int prix;
private Langage(){}
private Langage (int prix){
this.prix=prix;}

public String toString(){
return this.name()" coute "+ prix;}
}
```

DES MÉTHODES

- name() : renvoie la chaine de caractère du nom de l'option
- La méthode toString() peut être redéfinit , afin de donner un comportement diffèrent de celui par défaut
- ***values() : renvoie le tableau des options possibles***

  Langage[] liste= **Langage**.*values();*

  for (Langue l:liste) System.*out.println(l.name());*
- ***On peut coder ses méthodes comme pour une classe, (getPrix(), toString,…)***
- ***valueOf : converti une chaine de caractère en l'option correspondante***

  String s = "java";

  Langue l = Langue.*valueOf(s);*

  System.***out.println(l.getPrix());***

- Tp:
- Une classe Commande définit

Un nom de client

Une boisson parmi une liste(Eboisson étant un enum)

Un plat parmi une liste

Un dessert parmi une liste

A chaque sélection est associé un prix

Un méthode getPrixTotal() qui retourne le prix total a payer

La méthode toString () qui retourne de détail de la commande

- Une classe de Test déroule le scenario de choix de ses articles et affiche les infos a l utilisateur en fonction de ses choix

HÉRITAGE

Héritage : idée

Permet d enrichir une classe a partir d une autre :

soit une classe A contient de infos

Problème:

la classe B souhaite récupérer les fonctionnalités de A

On peut réaliser ceci de 2 manières:

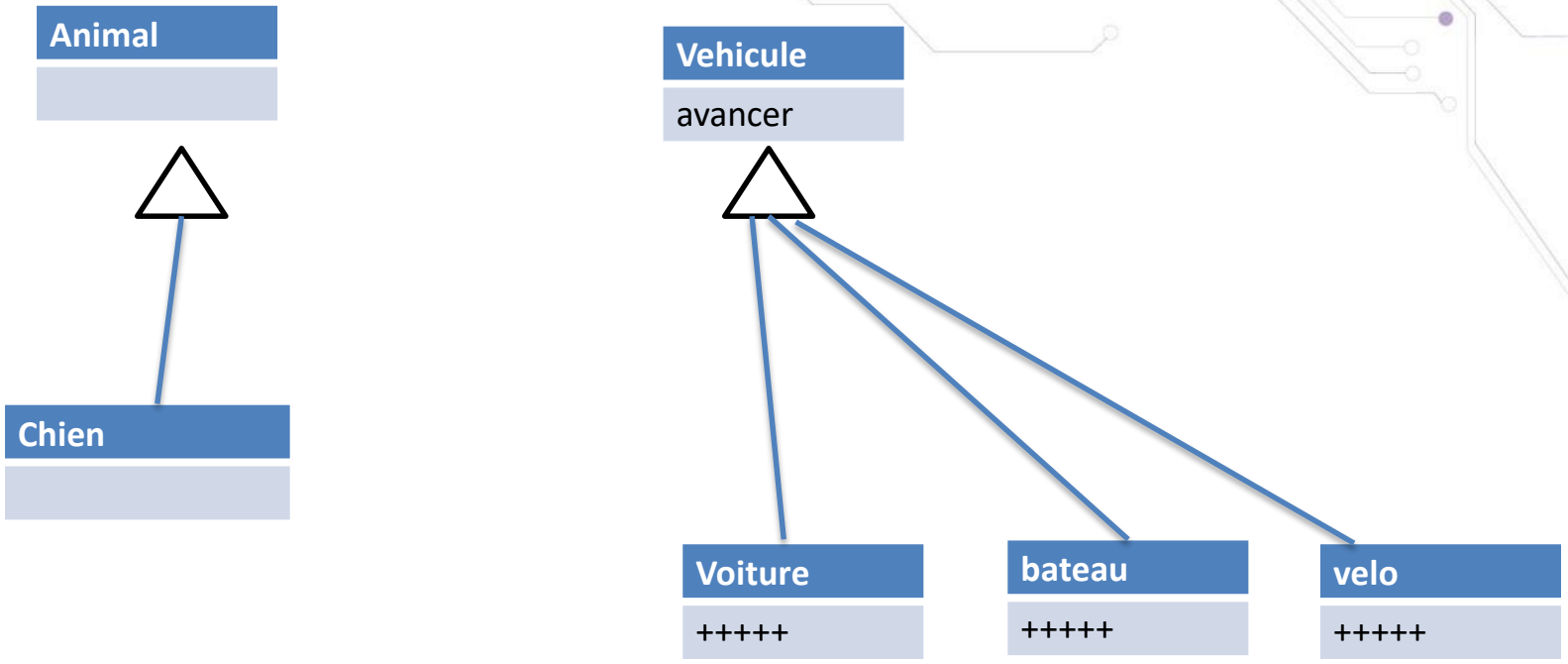
1. avec une association : la classe A possède un B, et a donc accès aux infos contenus dans B
2. On peut utiliser de l'héritage : B est un cas particulier de A et possède les deux ensemble d'info

Choix du type de lien : en fonction du bon sens

- Pour l'héritage : si B est une sorte de B (ex : chien/animal)
- Pour l'association : est composé de (ex : email/piece jointe)
- Un mauvais choix ne déclenche pas d'erreurs de compilation mais donne des non-sens

Pas d'héritage multiple (existe dans d'autres langages, permet d'hériter de deux classes à la fois. pb : ambiguïté possible sur les méthodes)

SYMBOLE : LIEN D'HÉRITAGE



VOCABULAIRE

B hérite de A, on dit que:

B est une spécialisation (un enrichissement) de A

A est une généralisation de B

B est la classe dérivée / la classe fille

A est la classe de base/la classe mère

COMMENT LE CODER ?

```
public class A {

}
```

```
public class B extends A {

}
```

Lien d'héritage :
B hérite de A

- Existe dans tous les languages objets, en .NET :
  Class B : A {}

PREMIER EXEMPLE

- Faire un nouveau package avec les classes suivantes :

Classe A :

```
public class A {
public void m1(){
System.out.println("m1");
}
}
```

Classe B :

```
public class B extends A {
public void m2(){
System.out.println("m2");
}
}
```

Classe Test :

```
public class Test {

public static void main(String[] args) {
A monA=new A();
monA.m1();
B monB=new B();
monB.m1();
monB.m2();
}
}
```

C'EST QUOI L'HÉRITAGE ?

B hérite de A s écrit : class B extends A {…}

Toutes les méthodes de A sont disponible dans B, comme si tous le code de A été copié-coller dans B : permet la réutilisation

Enrichissement : peut ajouter des choses (attributs, méthodes,…) dans la classe qui hérite

Toutes les classes héritent implicitement de la classe Object

On peut avoir des héritages en chaine: D hérite de C qui hérite de B qui hérite de A (qui hérite de Object).

On se retrouve dans D a accéder a toutes les méthodes de toutes les classes précédentes

Possible de sceller des classes pour empêcher l'héritage (ex : String)

PROTECTED

Visibilité protected (a un sens public et private) :

public pour les classe qui héritent

private pour l accès de l extérieur de la classe où il est défini

De quoi hérite-t-on ?

De tous ce qui est public : y compris pour appel local comme si le code été présent localement

Ce qui est privé dans la classe mère ne peut pas être appeler depuis la classe fille

Attribut : privé, ne sont donc pas accessibles depuis les classes filles

On peut redéfinir (override) des méthodes héritées (ex : toString() que l'on redéfini par rapport à la version existante dans Object)

Une classe peut hériter d'une autre classe en utilisant le mot **extends**.

Une classe Hérite d'une autre tout ses membres sauf le constructeur.

Quand une classe hérite d'une autre classe, elle a le droit de redéfinir les méthodes héritées.

Dans une méthode redéfinie, on peut appeler la méthode de la classe parente en écrivant le mot **super** suivi d'un point et du nom de la méthode parente. (**super.méthode()**).

Classe A :

```
public String toString() {
return "passage par A";
}
```

Classe B :

```
public String toString() {
return super.toString()+" et par B";
}
```

Dans main :

```
System.out.println(new B());
```

Affiche : passage par A et par B

CONSTRUCTEURS

Classe **A** :

```
public class A {
private int x;

public A(int x) {
this.x = x;
}
public void m1(){???}
public String toString() {
return "x="+x;
}
}
```

Classe B

```
public class B extends A {
private int y;

public B(int x, int y) {
super(x);
this.y = y;
}
public void m2(){???}
public String toString() {
return super.toString()+" y="+y;
} }
```

LES CONSTRUCTEURS

Il faut toujours définir le constructeur de la nouvelle classe dérivée.

Le constructeur de la classe dérivée peut appeler le constructeur de la classe parente en utilisant le mot **super(),** avec la liste des paramètres.

Il est obligatoire de faire appel au constructeur de A dans B !
(seul cas différent : le constructeur implicite appelle le constructeur implicite ou par défaut de la classe mère, ne marche pas si la classe mère a des constructeurs d'initialisation et pas de constructeur par défaut)

Un constructeur peut appeler un autre constructeur de la même classe en utilisant le mot **super()** avec des paramètres du constructeur.

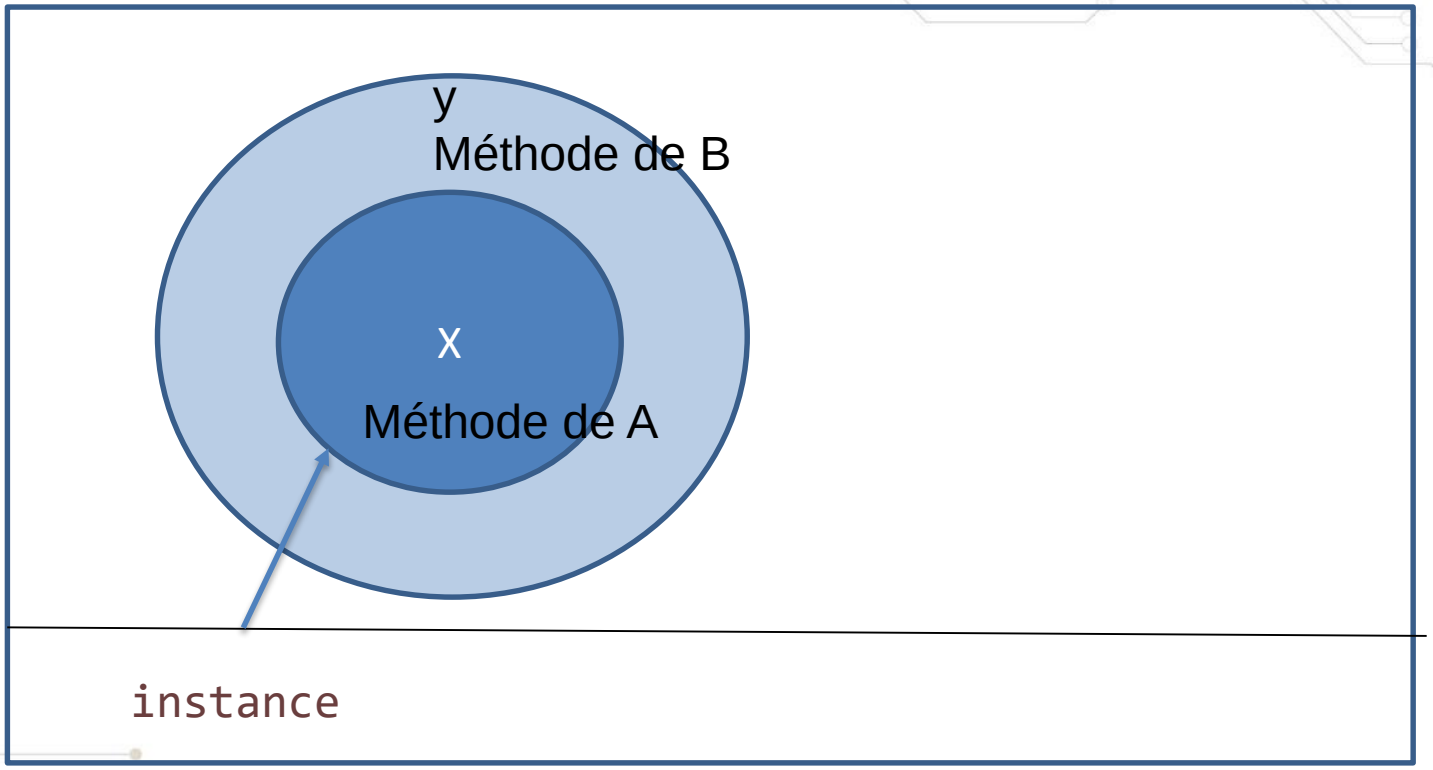
INSTANCE « HYBRIDE »

B hérite de A

```
A instance= new B(14,25);
```

cette déclaration est possible : on ne voit alors que les méthodes de A.

SCHÉMA MÉMOIRE



POLYMORPHISME :

```
Animal[] tab = new Animal[5];
tab[0]=new Animal();
tab[1]=new Chat();
tab[2]=new Chien();
for (Animal A:tab) {
if(A!=null)A.affiche();
}
```

Une méthode existant sous différentes versions dans chacune des classes liées par héritage permet un traitement spécifique à chaque classe fille.

CAST IMPLICITE/EXPLICITE

Soit les classe A, B hérite de A

B monB=new B();

A monA=monB; cast implicite

(restreint la visibilité aux méthodes de A)

B newB=(B)monA; cast explicite

(on retrouve les capacités de l objet initial)

GÉNÉRICITÉ DU CODE

Soit une classe mère A, et des classes B,C,D qui héritent de A

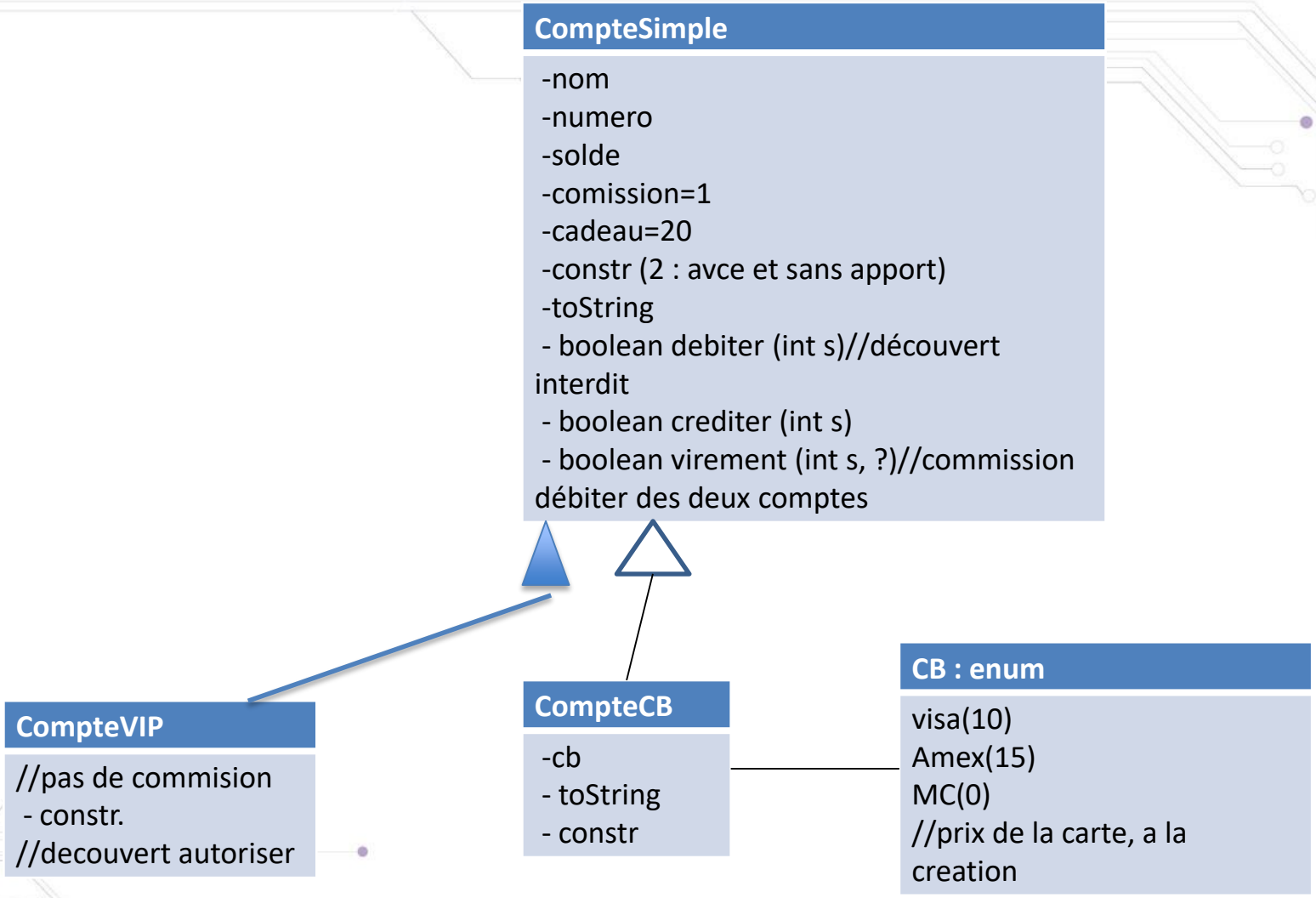
A propose la méthode m et son implémentation, B,C et D redéfinissent leur m

On peut proposer le code générique suivant

Dans la classe de

```
static void test5() {
A monA = new A(10);  B monB = new B(10, 20);
C monC = new C(10,"titi");  D monD = new D(10, 20, "toto");
affiche2(monA); affiche2(monB); affiche2(monC); affiche2(monD);
}
static void affiche(A x) {
x.m();
}
```

TP HÉRITAGE


CompteSimple
-nom
-numero
-solde
-comission=1
-cadeau=20
-constr (2 : avce et sans apport)
-toString
- boolean debiter (int s)//découvert interdit
- boolean crediter (int s)
- boolean virement (int s, ?)//commission débiter des deux comptes
CompteVIP
//pas de commision
- constr.
//decouvert autoriser
CompteCB
-cb
- toString
- constr
CB : enum
visa(10)
Amex(15)
MC(0)
//prix de la carte, a la creation

Classe et méthodes scellées

- Classe préfixée par **final** : empêche l'héritage
- Méthode **final** : empêche la surcharge dans les classes dérivées
- Rappel : attribut final = constante
- Exemple : Integer est une classe finale qui hérite de la classe abstraite Number (qui possède des méthodes abstraites et non abstraites, on voit que les abstraites sont bien implémentée dans Integer)

CAST IMPLICITE ET EXPLICITE

- Vers une classe plus général : implicite
- Vers une sous classe : explicite
- Attention : le cast est couteux, éviter si on peut (typiquement si on sait que les choses sont d'un type précis ne pas les stocker dans un type plus général)

```
public class Personne {
private String nom,prenom;
public Personne(String nom, String
pre){
this.nom=nom; this.prenom=pre;
}
@Override
public String toString() {
return nom + prenom;
}
public void affiche(){
System.out.println(this);
}
}
```

```
public class Test {
public static void main(String[] args) {
Personne p1 = new Personne("A","B");
p1.affiche();
Object o1=p1;
System.out.println(o1);
m(o1);
Personne p2 = (Personne)o1;
p2.affiche();
}
static void m(Object o){
System.out.println(o);}
}
```

Cast implicite

o1 ne voit pas la méthode affiche()

Cast explicite

CLASSE ABSTRAITE

CLASSE ABSTRAITE

- Une classe abstraite est préfixée par le mot clef « abstract »
- Empêche la création d'instance : même avec des constructeurs, on ne pourra pas faire de « new A » a savoir création d instances,
- Permet quand même l'utilisation de méthode static

```
public abstract class A {

public void m1(){
System.out.println("m1");}
public static void m2(){
System.out.println("m2");}

}
```

```
public static void main(String[] args){
A monA;
monA = new A();
A.m2();

}
```

possible

Pas possible :
On ne peut pas créer d'instance
D'une classe abstraite

POURQUOI ?

- Une classe possédant une (ou plus) méthode abstraite, la classe sera forcément abstraite
- Méthode abstraite : méthode non-implémentée, préfixé du mot-clef abstract et sans implémentation
- Sert pour l'héritage : les classe qui hérite d'une classe abstraite doivent implémenter les méthodes abstraites, et rend utilisable les méthodes implémentées dans la classe abstraite

Classe A :

```
public abstract class A
{

public void m1(){
System.out.println("m1")
;}
public static void m2(){
System.out.println("m2")
;}
abstract public void
m3();
}
```

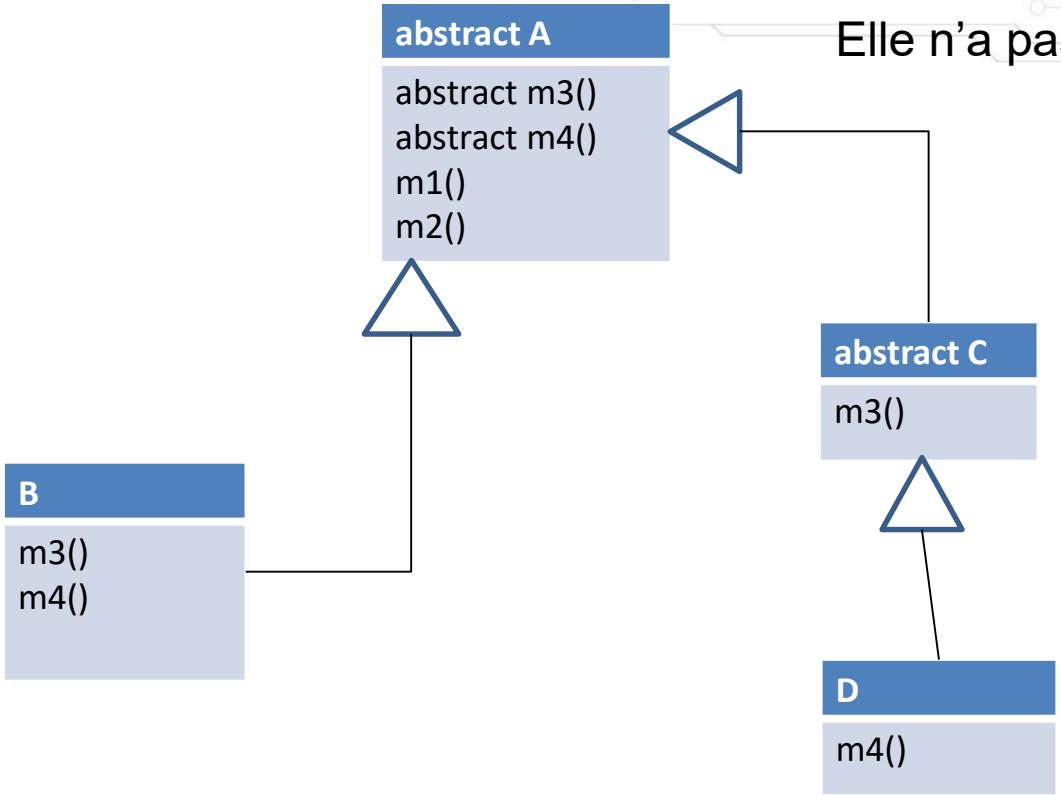
Classe B :

```
public class B extends A {

public void m3() {
System.out.println("m3");}
public void m4() {
System.out.println("m4");}
}
```

Classe de Test :

```
public class Main {
public static void main(String[] a){
A monA;
A.m2();
B monB = new B();
monB.m1();
monB.m3();
monB.m4();
}
}
```

EXEMPLE DE DIAGRAMME



C est forcément abstraite :
Elle n'a pas implémenté toute les méthodes abstraites

D n'est pas forcément abstraite :
Elle a implémenté toutes les méthodes abstraites (directement ou par héritage)

MOTIVATIONS

- Une classe abstraite permet l'utilisation de méthodes déjà codées à condition d'implémenter certaines méthodes, donne une contrainte sur ses classes dérivées
- Une classe fille bénéficie des méthodes de la classe abstraite, avec pour contre partie de devoir implémenter les méthodes abstraites.
- Avec l'exemple du diagramme précédent, on peut faire :

```
A[] tab = new A[3];
tab[0] = new B();
tab[1] = new D();
tab[2] = new B();
for (A e : tab) e.m3();
```

Comme la classe A impose à ses dérivées d'implémenter la méthode m3(), on peut utiliser le polymorphisme pour exécuter la méthode sur des différents types réels(les instances).

POURQUOI UTILISER ABSTRACT ?

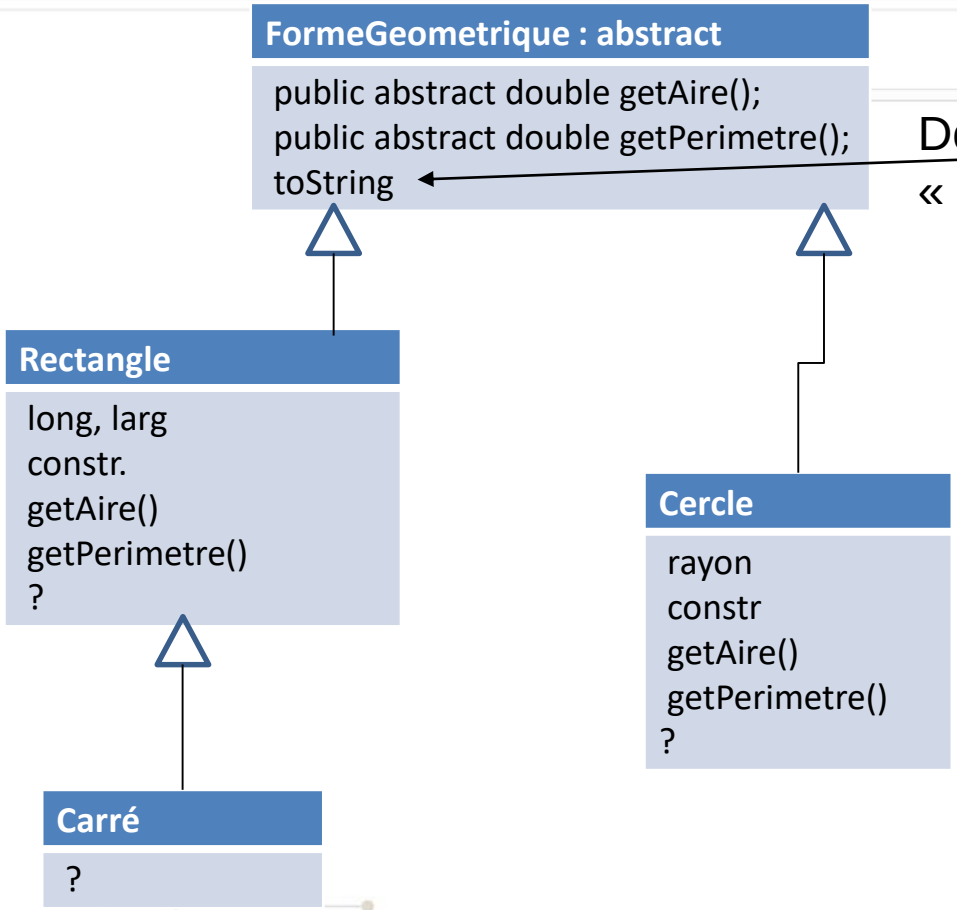
Une méthode est abstraite quand elle a pas vraiment de sens dans la méthode où elle est codée, que sont fonctionnement dépend intrinsèquement de la classe fille pour laquelle elle sera appelée

Les classes abstraites : non instanciable, pas de réalité concrète, une généralité pour laquelle il nous faut une spécialisation. N'a pas lieu d'être créée car n'a pas de sens propre logique.

Ex: classe Article, FormeGeometrique,Vehicule

Garantit une utilisation normale/un modèle cohérent

Permet de grouper des classes indépendantes pour pouvoir utiliser du polymorphisme afin par exemple de faire un tableau pour un traitement sur différents types de données, crée de la généricité



Donne par ex :
« Rectangle d'aire 20 et de périmètre 24 »

A tester avec :

```
public static void main(String[] args) {
FormeGeometrique[] tab = new FormeGeometrique[3];
tab[0]=new Rectangle(10,5);
tab[1]=new Carre(4.5);
tab[2]=new Cercle(3);
for (FormeGeometrique f :tab)
System.out.println(f);
}
```

```
Qui renvoie :
Rectangle d'aire 50.0 et de périmètre 30.0
Carre d'aire 20.25 et de périmètre 18.0
Cercle d'aire 28.2 et de périmètre 18.8
```

DES POINTS INTÉRESSANTS

abstract :

Une classe abstraite est une classe qui ne peut pas être instanciée. Une méthode abstraite est une méthode qui peut être définie à l'intérieur d'une classe abstraite. C'est une méthode qui n'a pas de définition. Par conséquent, elle doit être redéfinie dans les classes dérivées.

Une interface ressemble a une classe abstraite avec méthodes abstraites.

Dans java une classe hérite toujours d'une seule classe et peut implémenter plusieurs interfaces.

final:

Une classe final est une classe qui ne peut pas être dérivée.

Une méthode final est une méthode qui ne peut pas être redéfinie dans les classes dérivées.

Une variable final est une variable don la valeur ne peut pas changer

On utilise final pour deux raisons: une raison de sécurité et une raison d'optimisation

DES POINTS INTÉRESSANTS

static:

Les membres statiques d'une classe appartiennent à la classe et partagés par toutes ses objets

Les membres statiques sont accessible en utilisant directement le nom de la classe

Les membres statiques sont accessible sans avoir besoin de créer une instance de la classe qui les contient

Les membres statiques sont également accessible via les instances de la classe qui les contient

Les Interfaces

INTERFACES

Une interface est une sorte de classe abstraite qui ne contient que des méthodes abstraites.

En java une classe hérite d'une seule classe et peut implémenter plusieurs interfaces.

On dit qu'une classe implémente une ou plusieurs interfaces.

Une classe adopte le comportement de l interface

On dit que l interface constitue un contrat que doit être rempli par la classe

Une entité qui définit un nouveau type

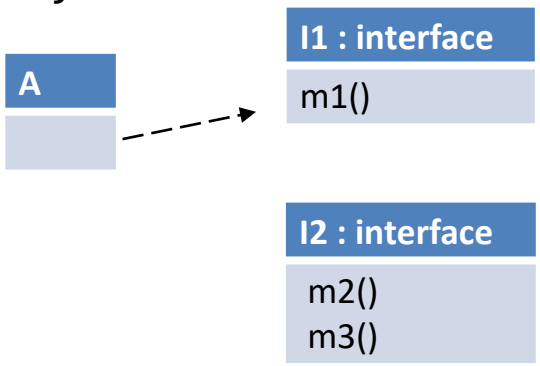
Soit A une classe et I une interface, on écrit

```
class A implements I {}
```

INTERFACE

- Une entité qui définit un nouveau type
- Contient **uniquement des méthodes non implémentées** (pas préfixée par abstract) : pas d'attributs, pas de méthodes implémentées

```
public interface I2 {
public void m2();
public void m3();
}
```




```
public interface I1 {
public void m1();
}
public class A implements I1 {

public void m1() {
System.out.println("M1 de A");
}
}
```

Doit implémenter toutes les méthodes de l'interface

utilisation

```
I1 a =new A();ou A a =new A();
a.m1();
```

UNE INTERFACE = UN CONTRAT

L'interface donne un contrat aux classes qui l'implémente

Une classe qui implémente une interface doit « adopter son comportement », ie, doit implémenter toutes ses méthodes

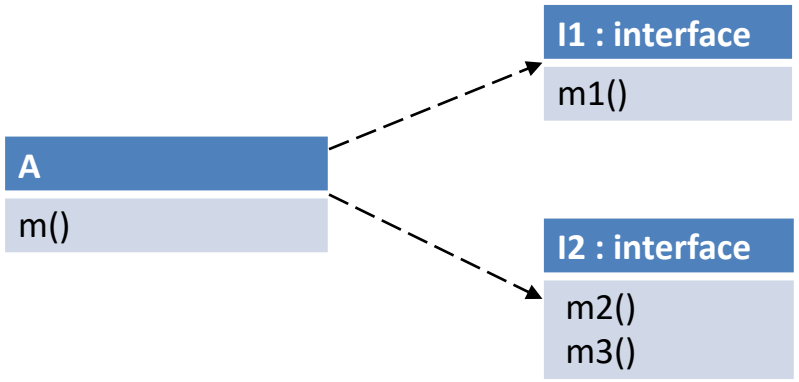
Exemple :
une interface cryptable possède des méthodes crypter et décrypter

la classe Document implémente l interface cryptable se doit d implémenter toutes les méthodes de cryptable

On peut typer avec l'interface,
visiblité = visibilité des méthodes de l'interface

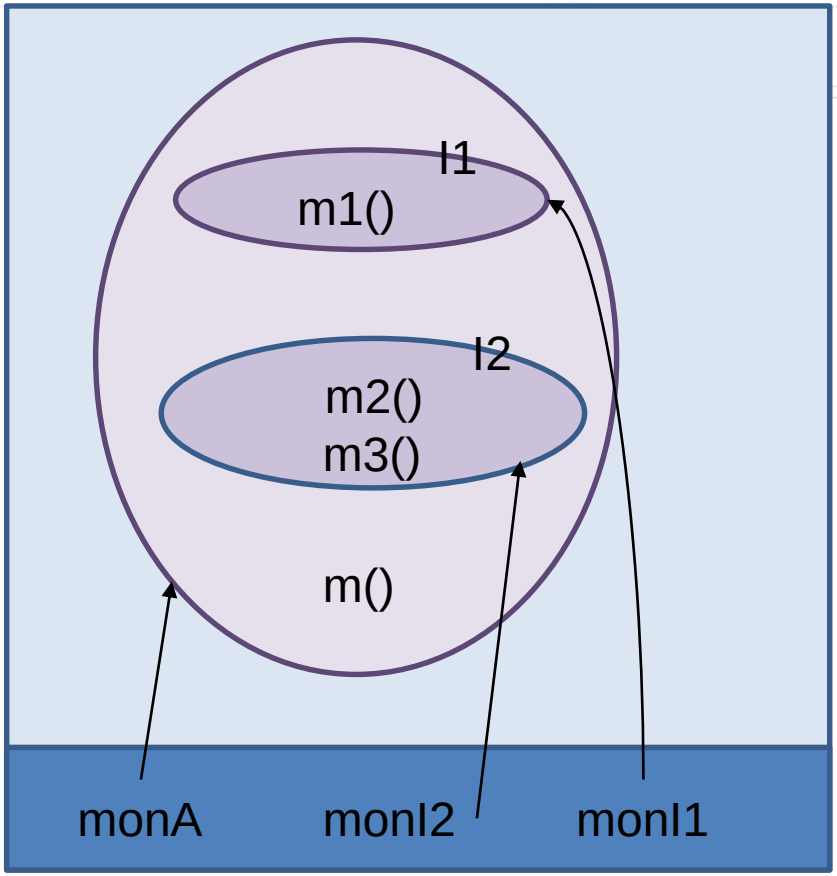
Multiple implémentation d'interface

- Une classe peut implémenter plusieurs interfaces :




```
public class A implements I1, I2 {

public void m1() {…}
public void m2() {…}
public void m3() {…}

public  void m() {…}
}
```

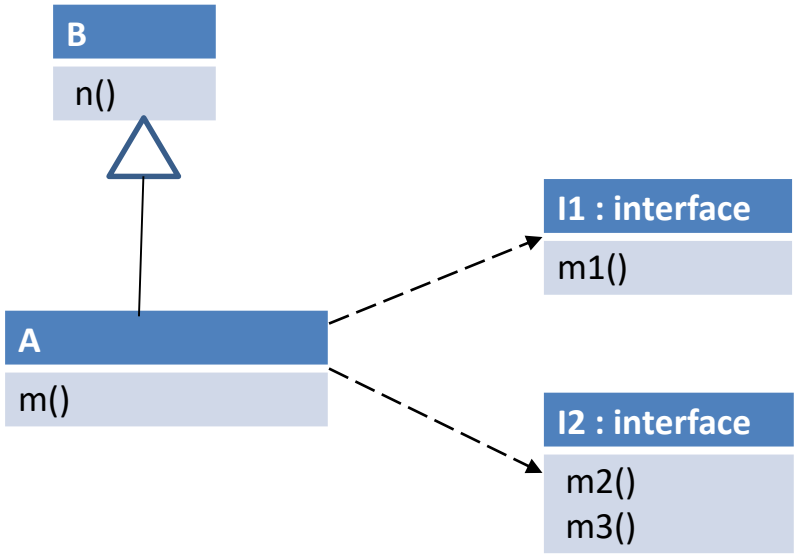
Il faut alors implémenter tous les méthodes de chacune des interfaces




```
A monA =new A();
I1 monI1 =  monA;
I2 monI2 = (I2) monA;
```

INTERFACE ET HÉRITAGE

- L'héritage et l'interface sont compatibles :



interfaçage
héritage

```
public class A extends B implements I1, I2 {

public void m1() {…}
public void m2() {…}
public void m3() {…}
public void m() {…}
}
```

Une classe abstraite peut 'implémenter' une interface.
Elle peut alors implémenter les méthodes ou déclarer des méthodes abstraite à implémenter (l'implémentation est alors déléguée aux classes dérivées)

Exemple : Integer hérite de Number et implémente Comparable
Number (classe abstraite) implémente Serializable

Pas de multiple héritage

Plusieurs solutions pour y palier, différents types de pattern pour s'y substituer dans différents cas.
Typiquement le pattern Façade

Multiple implémentation permet de récupérer l'aspect appartient à plusieurs famille, mais ne permet pas de récupérer des méthodes implémentées

CAST

Pas très propre :
Peut etre source d erreurs, plante à l'éxécution si A n'implémente pas I1

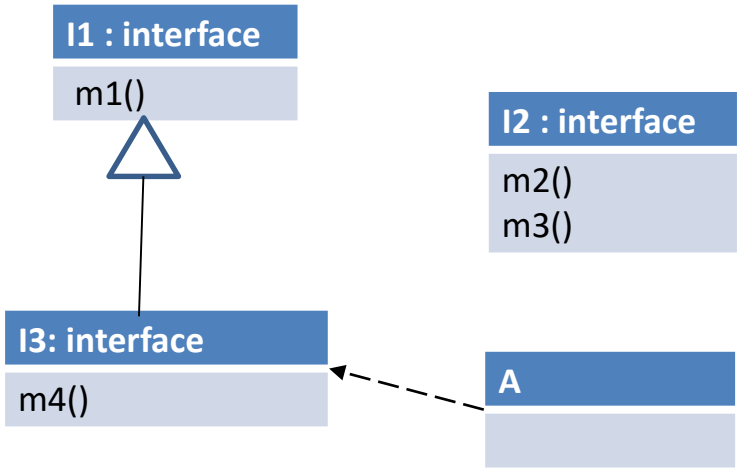
```
A monA =new A();
I1 monI1 =  monA;
```

Solution : on peut vérifier le type de l instance : via instanceof (pas seulement pour les interfaces)

```
if(monA instanceof I1 ){
        I1 monI1 =  monA;
}
```

HÉRITAGE ENTRE INTERFACES

- Une interface peut hériter d'une autre : on dit plutôt **étendre** (mais c'est le même principe et mot clef)

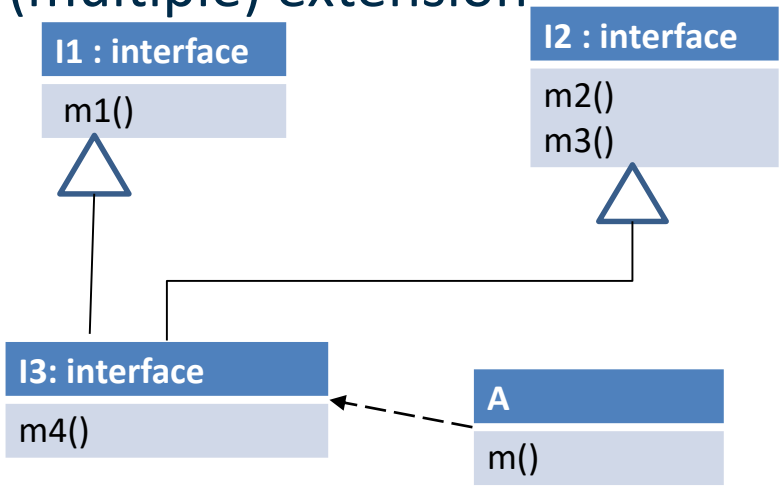



```
public interface I3 extends I1 {
public void m4();
}
```

A doit implémenter les méthodes de I3 et I1, et ses instances peuvent être 'castées' en I3 ou en I1

HÉRITAGE (MULTIPLE) ENTRE INTERFACE

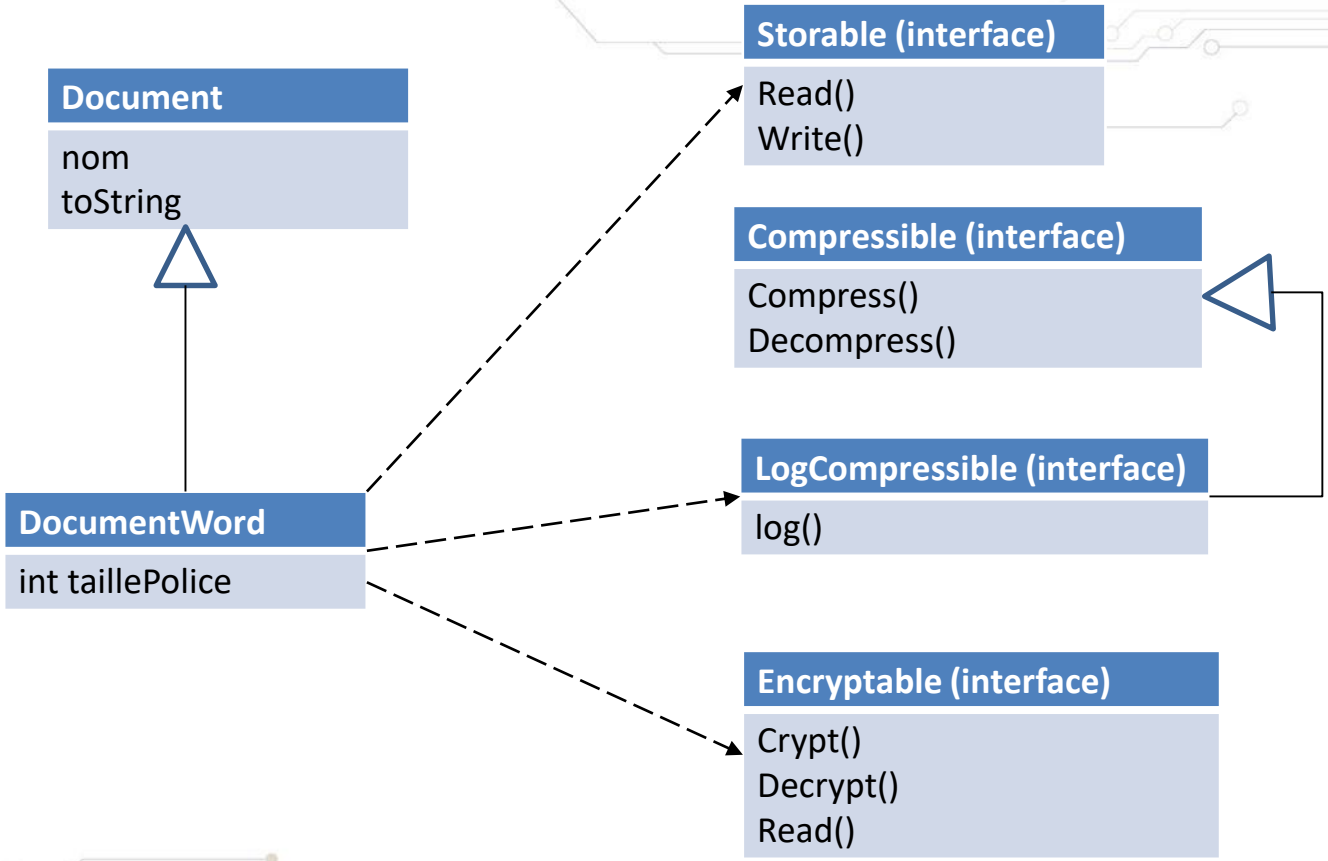
- On peut faire de l'héritage (multiple) **entre interfaces** (multiple) extension



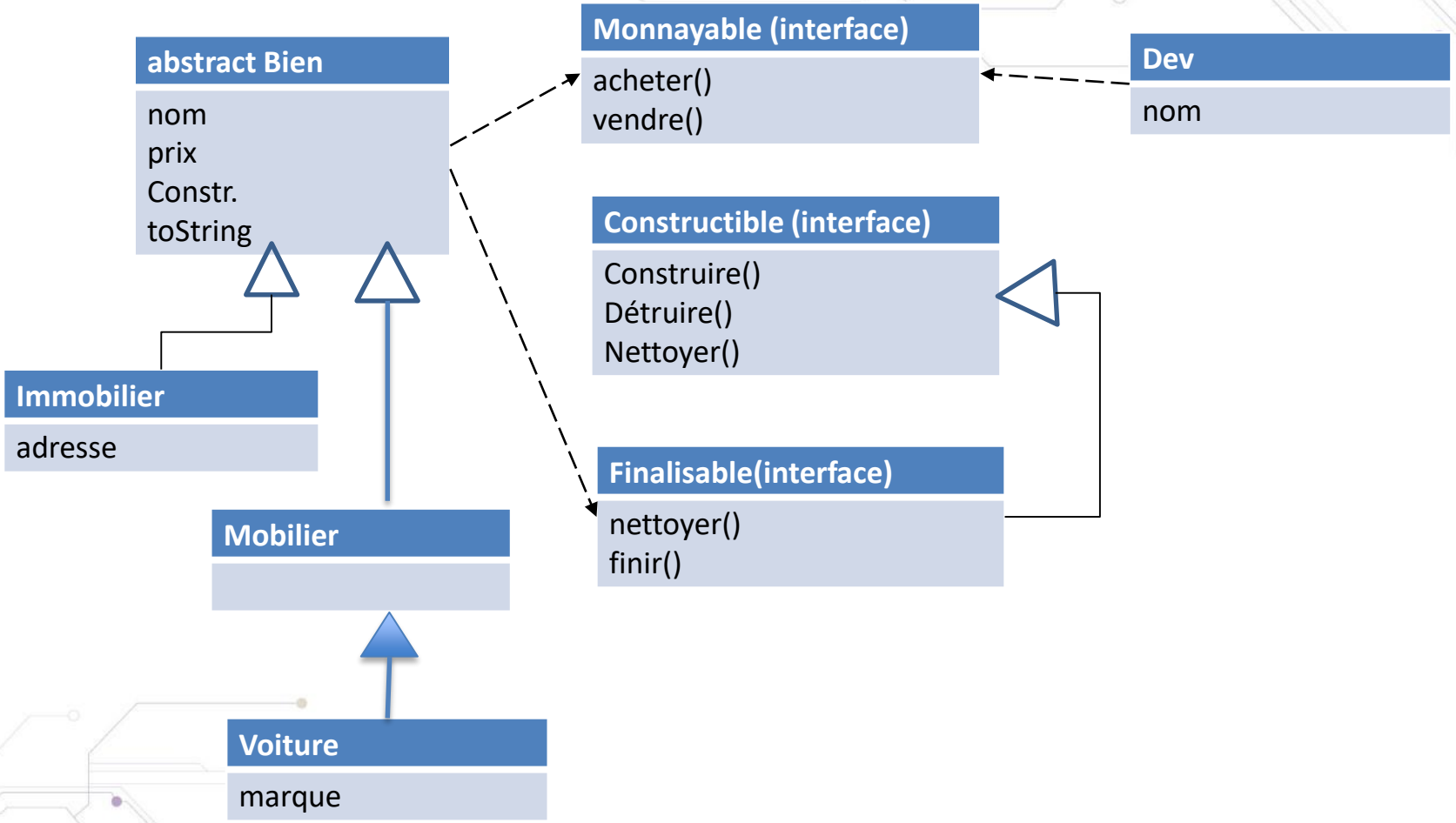

```
public class A extends B implements I3{

public void m1() {…}
public void m2() {…}
public void m3() {…}
public void m4() {…}
public void m() {…}
}
```

```
public interface I3 extends I1, I2 {
public void m4();
}
```

EX À CODER EN COURS : GESTION DE DOCUMENTS



TP : A FAIRE


abstract Bien
nom
prix
Constr.
toString
Monnayable (interface)
acheter()
vendre()
Dev
nom
Constructible (interface)
Construire()
Détruire()
Nettoyer()
Immobilier
adresse
Finalisable(interface)
nettoyer()
finir()
Mobilier
Voiture
marque

TP : A FAIRE

- Créations d une classe de traitement avec des méthodes génériques ayant en paramètre un Bien, un Monnayable etc,,