CPO

HÉRITAGE : PROBLÉMATIQUE

On doit coder une application où interviennent des étudiants (classe Student) et des délégués de classe (classe ClassRep).

```
class Student {

   String name;
   String forname;
   int year;

   void attendTo(Course c);
   void doTheir(Homework w);
}
```

```
class ClassRep {

   String name;
   String forname;
   int year;
   boolean isSubsitute;

   void attendTo(Course c);
   void doTheir(Homework w);
   void informTheir(ClassMate cm);
   void participateTo(Jury j);
}
```

CPO

HÉRITAGE

On observe visiblement une redondance de code lié au recouvrement des deux concepts (tous les ClassRep sont aussi des Student).

```
class Student {

   String name;
   String forname;
   int year;

   void attendTo(Course c);
   void doTheir(Homework w);
}
```

```
class ClassRep {

   String name;
   String forname;
   int year;
   boolean isSubsitute;

   void attendTo(Course c);
   void doTheir(Homework w);
   void informTheir(ClassMate cm);
   void participateTo(Jury j);
}
```

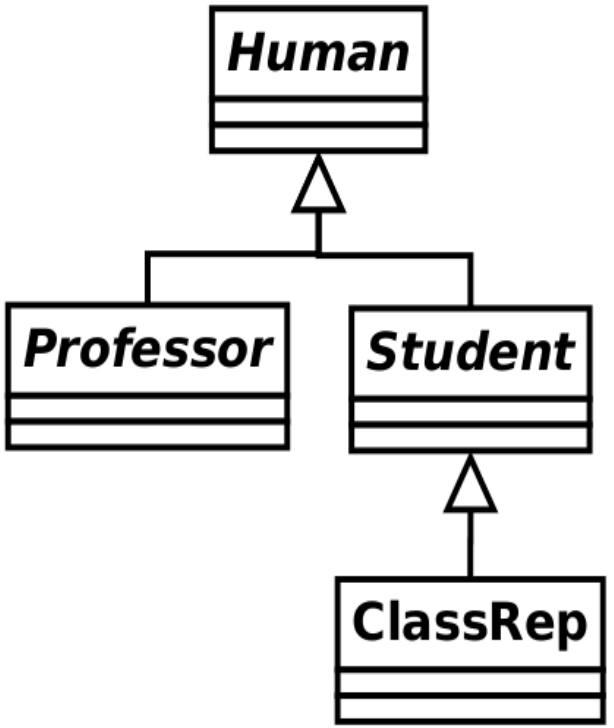
CPO

On peut dire que les Délégués de Classe sont des cas particuliers des Étudiants (**spécialisation**).
On pourrait dire à l'inverse que tous les Étudiants et tous les Professeurs sont tous des Humains (**généralisation**).

On traduit la notion d'héritage par « est un ». Un Délégué de Classe « est un » Étudiant.

CPO

On en déduit que les délégués de classe, les étudiants, les professeurs et les humains partagent des caractéristiques (attributs) et des capacités (méthodes) communes.
On en déduit aussi qu'ils possèdent des caractéristiques et des capacités différentes justifiant des concepts séparés.
Certaines classes possèdent visiblement une relation de filiation (tous les délégués sont des étudiants) et d'autres non (professeurs et étudiants sont disjoints).



CPO

L'héritage est un mécanisme permettant de créer une nouvelle classe à partir d'une classe existante en lui proférant ses propriétés et ses méthodes.

```
class Student {

   String name;
   String forname;
   int year;

   void attendTo(Course c);
   void doTheir(Homework w);
}
```

```
class ClassRep extends Student {

   boolean isSubsitute;

   void informTheir(ClassMate cm);
   void participateTo(Jury j);
}
```

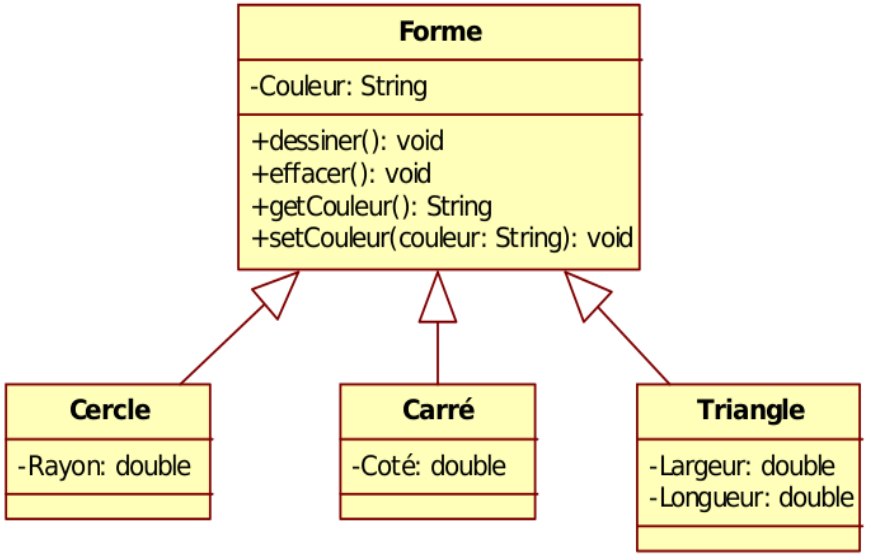
En Java, on utilise le mot-clé « extends ».

CPO

- La classe qui hérite est dite **classe fille ou sous-classe ou classe dérivée**.
- La classe dont on hérite est dite **classe mère**.
- La classe fille possède au moins tous les attributs et méthodes de la classe mère.
- La classe mère ne possède **PAS** tous les attributs et méthodes de la classe fille.

En Java, une classe ne peut hériter que d'une seule classe mère.

CPO



- Les classes Cercle, Carré et Triangle héritent de la classe Forme.

- Elles ont des attributs propres.

- La classe Cercle accède à toutes les méthodes (et les attributs) publiques, mais pas privées de la classe Forme.

- Toutefois, les attributs privés de Forme ont bien été créés pour Cercle, Carré et Triangle. Ils restent accessibles pas des getters ou des setters.

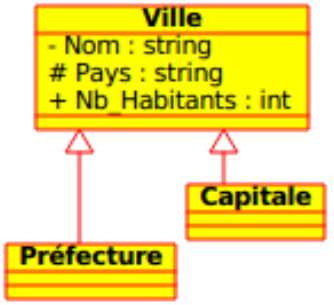
CPO

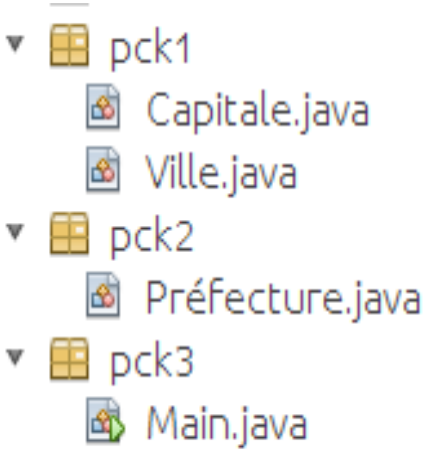
Une visibilité peu employée : « protected »

Un élément « protected » (protégé) est accessible uniquement aux classes d'un package et à ses classes filles.

Dans un même package, « protected » est équivalent à « public ».

```
package pck1;

public class Ville {
    private String Nom;
    protected String Pays;
    public int nbHabitants;
}
```

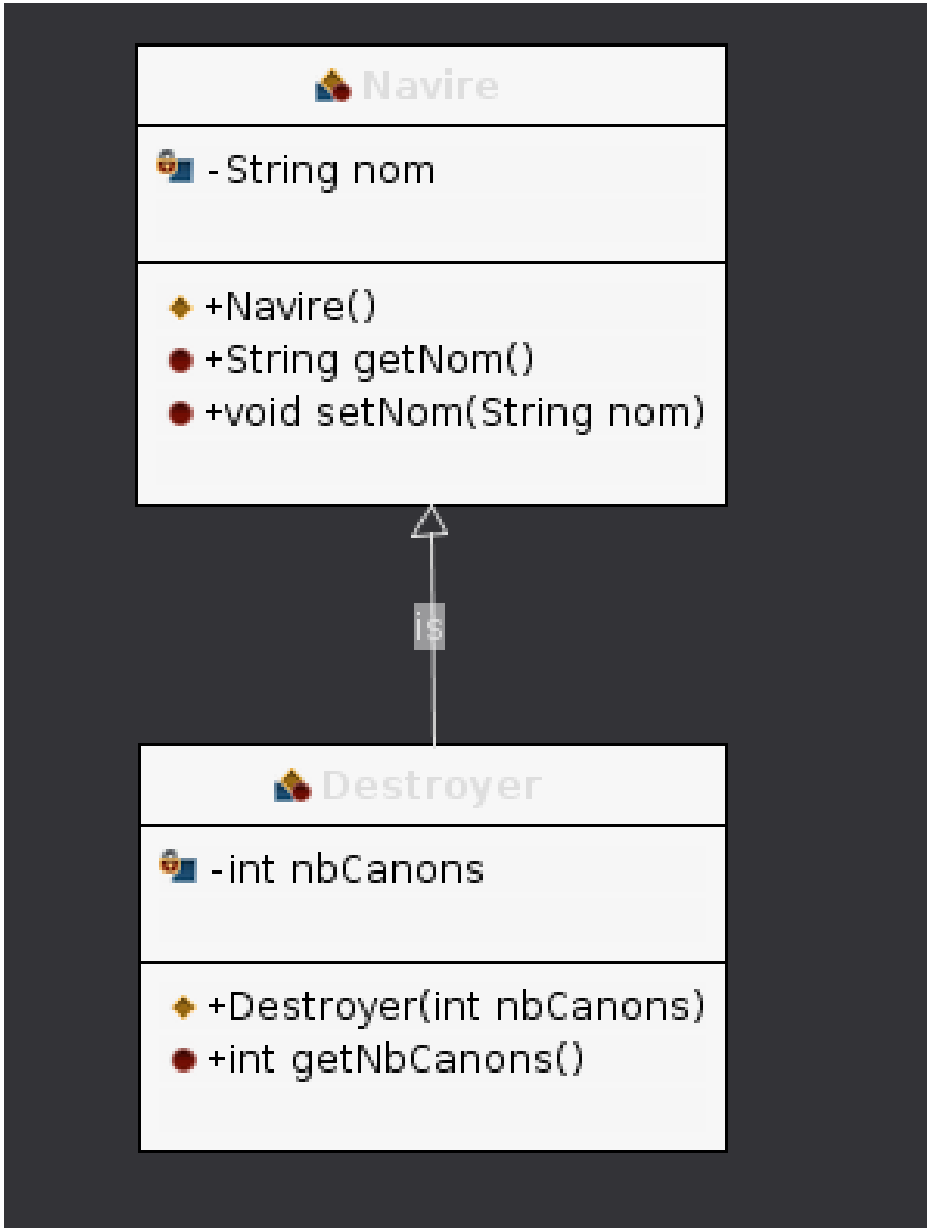




CPO

Exemple :

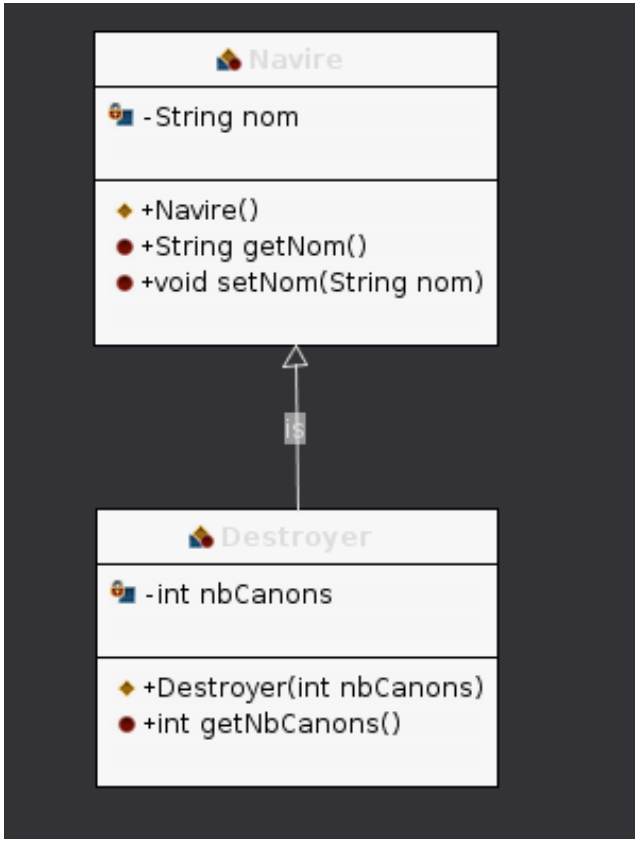
Codage des 2 classes suivantes :



CPO

```
public class Main
{
public static void main(String[] args) {
    Destroyer destr1 = new Destroyer(25);
    destr1.setNom("Le Chacal");
    System.out.println("nom : "+destr1.getNom());
    System.out.println("canons : "+destr1.getNbCanons());
  }
}
```

Qu'obtient-on à l'exécution ?



CPO

La création d'un objet (de type Destroyer) se déroule en 6 étapes.

1. **Allocation mémoire** pour un objet de type Destroyer ; il s'agit bien de l'intégralité de la mémoire nécessaire pour un tel objet (champs propres à la classe Destroyer et ceux hérités de Navire).

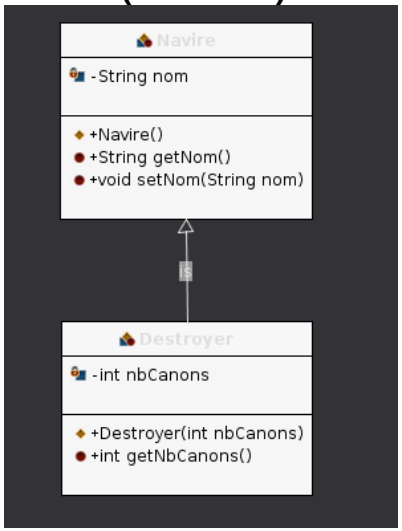
2. **Initialisation par défau**t de tous les champs de la classe fille (Destroyer) ,aussi bien ceux hérités de la classe mère (Navire), que ceux propres à la classe fille (Destroyer) aux valeurs "nulles" habituelles.

3. **Initialisation explicite**, s'il y a lieu, des champs hérités **de la classe mère** (Navire).

4. **Exécution** du corps du **constructeur de la classe mère** (Navire).

5. **Initialisation explicite**, s'il y a lieu, des champs propres **de la classe fille** (Destroyer).

6. **Exécution** du corps **du constructeur de la classe fille** (Destroyer).



CPO

L’instruction super

Par défaut, Java appelle toujours **le constructeur de la classe mère sans argument** (qu'il existe ou pas).

```
public class Navire {    //classe mère
    private String nom;

    public Navire() {
       System.out.println("constructeur");
    }

    //autres méthodes
}
```

CPO

L’instruction super

Que se passe-t-il si la classe mère possède un constructeur avec un argument ?

```
public class Navire { //classe mère
    private String nom;

    public Navire(String nom) {
        this.nom = nom;
    }

    //autres méthodes
}
```

CPO

L’instruction super

Si un autre constructeur avec argument a été codé, il faut explicitement l’appeler avec l’instruction super.

```
public class Destroyer extends Navire{ //classe fille
    private int nbCanons;

    public Destroyer(String nom, int nbCanons) {
        super(nom); //appel au constructeur de Navire
        this.nbCanons = nbCanons;
    }
     //autres méthodes
}
```

CPO

```
public class Navire { //classe mère
    private String nom;

    public Navire(String nom) {
        this.nom = nom;
    }
    //autres méthodes
}
public class Destroyer extends Navire{ //classe fille
    private int nbCanons;

    public Destroyer(String nom, int nbCanons) {
        super(nom); //appel au constructeur de Navire
        this.nbCanons = nbCanons;
    }
     //autres méthodes
}
```

CPO

Quelques règles :

1. Si la classe de base ne possède pas de constructeur, c'est le constructeur par défaut (sans paramètre) qui est appelé. Coder alors super() est superflu.

```
public class Navire { //classe mère
    private String nom;

    //autres méthodes
}
public class Destroyer extends Navire{ //classe fille
    private int nbCanons;

    public Destroyer(int nbCanons) {
        super(); //inutile
        this.nbCanons = nbCanons;
    }
     //autres méthodes
}
```

CPO

Quelques règles :

2. Si la classe dérivée n'a pas de constructeur, il y a donc appel à un constructeur par défaut (sans paramètre) de cette classe. Dès lors, il ne peut y avoir appel qu'à un constructeur sans paramètre (par défaut ou non) de la classe de base.

```
public class Navire { //classe mère
    private String nom;

    //autres méthodes
}
public class Destroyer extends Navire{ //classe fille
    private int nbCanons;

     //autres méthodes
}
```

CPO

Quelques règles :

3. Le constructeur de la classe de base étant exécuté avant celui de la classe dérivée, l'instruction super() doit donc être codée en premier.

```
public class Navire { //classe mère
    private String nom;
    public Navire(String nom) {
        this.nom = nom;
    }
    //autres méthodes
}
public class Destroyer extends Navire{ //classe fille
    private int nbCanons;
    public Destroyer(String nom, int nbCanons) {
        this.nbCanons = nbCanons;
         super(nom);//impossible
    }
     //autres méthodes
}
```

APPLICATION

Que manque-t-il nécessairement à la classe B ? Qu'affiche le code ci-dessous ?

```
class A {
    private int p = 10;
    private int n;
    public A(int nn) {
        System.out.println("Entrée Constructeur A  n=" + n + " p=" + p);
        n = nn;
        System.out.println("Sortie Constructeur A  n=" + n + " p=" + p);
    }
    public int getP() {
        return p;
    }
    public void setP(int p) {
        this.p = p;
    }
}
class B extends A {
    private int q = 25;
    public B(int n, int pp) {
        System.out.println("Entrée Constructeur B  n=" + n + " p=" + getP() + " q=" + q);
        setP(pp);
        q = 2 * n;
        System.out.println("Sortie Constructeur B  n=" + n + " p=" + getP() + " q=" + q);
    }
}
public class Main {
    public static void main(String[] args) {
        A a = new A(5) ;
        B b = new B(5, 3) ;
    }
}
```

CPO

Redéfinition de méthodes

Une classe dérivée peut fournir une nouvelle définition d'une méthode d'une classe ascendante. Il s'agira non seulement de **méthodes de même nom**, mais aussi **de même signature** et **de même type de valeur de retour**.

```
public class Navire {
    private String nom;
    public void afficher(){
        System.out.println("son nom est : "+nom);
    }
     //autres méthodes
}
public class Destroyer extends Navire{
    public void afficher(){
        System.out.println("c'est un destroyer");
    }
     //autres méthodes
}
```

Redéfinition de méthodes

Toutefois, un destroyer étant un navire, comment faire pour afficher son nom, c'est-à-dire, comment faire appel à la méthode afficher() de Navire ? Il suffit d'appeler le mot-clé super.

```
public void afficher(){    //méthode de Destroyer
      super.afficher();
      System.out.println("c'est un destroyer");
}
```

Redéfinition de méthodes

1) La redéfinition d'une méthode **ne doit pas diminuer les droits d'accès** à cette méthode, **mais peut les augmenter**. Cela signifie que la méthode afficher () de Navire pourrait être private et celle de Destroyer public, mais pas l'inverse.

2) Redéfinir un attribut est possible mais rarement (voir jamais) utilisé.

3) La redéfinition de méthodes ne concerne que les méthodes ayant **même signature** (même nom, même paramètre, même valeur de retour).

APPLICATION

Qu'affiche le code ci-dessous ?

```
class A
{
	public void affiche() {
		System.out.println ("Je suis un A") ;
	}
}
class B extends A {
}
class C extends A {
	public void affiche() {
		System.out.println ("Je suis un C") ;
	}
}
class D extends C {
	public void affiche() {
		System.out.println ("Je suis un D") ;
	}
}
class E extends B {
}
class F extends C {
}
public class Main {
	public static void main (String arg[]) {
	A a = new A() ; a.affiche() ;
	B b = new B() ; b.affiche() ;
	C c = new C() ; c.affiche() ;
	D d = new D() ; d.affiche() ;
	E e = new E() ; e.affiche() ;
	F f = new F() ; f.affiche() ;
}}
```

CPO

Surcharge de méthodes

La surcharge concerne les méthodes dont la signature est différente entre la classe de base et la classe dérivée.

```
class A
{
   public void f(int n) { ..... }
}

class B extends A
{
   public void f(float x) { ..... }
}

A a = new A();
B b = new B();
int n ;
float x ;
a.f(n) ; // appelle f(int) de A
a.f(x) ; // erreur de compilation
         // et on ne peut pas convertir x de float en int
b.f(n) ; // appelle f(int) de A
b.f(x) ; // appelle f(float) de B
```

CPO

On peut également changer la valeur de retour lors de la surcharge.

```
class A
{
    public void f (int n) { ..... }
}

class B extends A
{
    public int f (int x) { ..... }
}
```

La surcharge et la redéfinition peuvent cohabiter, mais il faut être bien sûr de son intérêt !

```
class Main {
    public static void main(String[] args) {
        byte bb = 1;
        short p = 2;
        int n = 3;
        long q = 4;
        float x = 5.f;
        double y = 6.;
        A a = new A();
        a.f(bb);
        a.f(x);
        System.out.println();
        B b = new B();
        b.f(bb);
        b.f(x);
        System.out.println();
        C c = new C();
        c.f(bb);
        c.f(q);
        c.f(x);
        System.out.println();
        D d = new D();
        d.f(bb);
        d.f(q);
        d.f(y);
        System.out.println();
        E e = new E();
        e.f(bb);
        e.f(q);
        e.f(y);
        System.out.println();
        F f = new F();
        f.f(bb);
        f.f(n);
        f.f(x);
        f.f(y);
        f.f(p);
    }
}
```

```
class A {
    public void f(double x) {
        System.out.print("A.f(double=" + x + ") ");
    }
}

class B extends A {}

class C extends A {
    public void f(long q) {
        System.out.print("C.f(long=" + q + ") ");
    }
}

class D extends C {
    public void f(int n) {
        System.out.print("D.f(int=" + n + ") ");
    }
}

class E extends B {}

class F extends C {
    public void f(float x) {
        System.out.print("F.f(float=" + x + ") ");
    }
    public void f(int n) {
        System.out.print("F.f(int=" + n + ") ");
    }
}
```

CPO

Le mot clef « final »

Rappel :
Pour un attribut : ne peut plus être modifié après une 1ère affectation.

Pour une méthode : ne peut pas être redéfinie dans une sous-classe.

```
public class Ville {
    public final void calculer_PNB()
    {
        //code
    }
}
```

Pour une classe : ne peut pas être dérivée.

```
public final class Ville{
}
```

Transtypage et polymorphisme

Le transtypage (conversion de type ou cast en anglais) consiste à modifier le type d'une variable ou d'une expression.

Rappels

- Transtypage implicite

Traité automatiquement par le compilateur.
Le type cible a un plus grand domaine que le type d'origine.
Exemple :

```
long destination;
int origine=9;
destination=origine;
```

- Transtypage explicite

Doit être spécifié par le programmeur.
Le type d'origine a un plus grand domaine que le type cible.
Exemple :

```
long destination=9;
int origine;
origine=(int) destination;
```

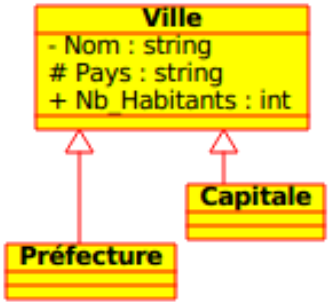
Transtypage et polymorphisme

Il est possible de convertir un objet d'une classe en un objet d'une autre classe si les classes ont un lien d'héritage.

Le transtypage d'un objet dans le sens fille/mère est implicite.
En revanche, le transtypage dans le sens mère/fille doit être explicite et n'est pas toujours possible.

Exemple :

```
Capitale c = new Capitale();
Ville v = new Ville();
v=c;
c=v;    //impossible
c=(Capitale) v;
```

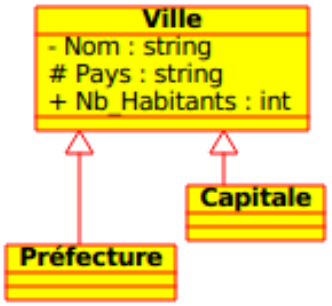


CPO

Transtypage et polymorphisme

De la même façon, on aura :

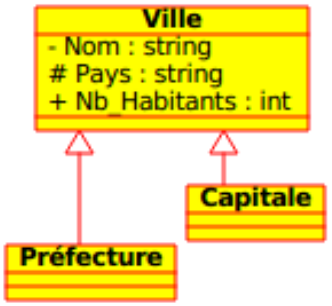
```
Capitale c2 = new Ville();  //impossible
Ville v2 = new Capitale();
```



Transtypage et polymorphisme

De la même façon, on aura :

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

 Capitale c = new Capitale();
 Ville v = new Ville();
 Préfecture p = new Préfecture();

 tab[0]=c;
 tab[1]=(Capitale)v;
tab[2]=p;//impossible
 tab[2]=(Capitale)p;//impossible
```



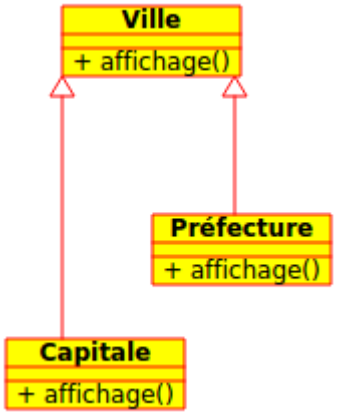
Par contre :

```
Ville []tab2 = new Ville [3];
 tab2[0]=c;
 tab2[1]=v;
 tab2[2]=p;
```

CPO

Transtypage et polymorphisme

```
Capitale c = new Capitale();
 Ville v = new Ville();
 Préfecture p = new Préfecture();


Ville []tab2 = new Ville [3];
 tab2[0]=c;
 tab2[1]=v;
 tab2[2]=p;

 for (int i=0;i<tab2.length;i++)
     tab2[i].affichage();
```



Déclaration d'un tableau de « Ville ».
Pourtant, on appellera la méthode « affichage » successivement de Capitale, puis de Ville, puis de Préfecture.

Transtypage et polymorphisme

En Java, dans la plupart des situations où il y a des relations d'héritage, la détermination de la méthode à invoquer n'est pas effectuée lors de la compilation.

C'est seulement **à l'exécution** que la machine virtuelle déterminera la méthode à invoquer selon le type effectif de l'objet référencé à ce moment là.

Ce mécanisme s'appelle "Recherche dynamique de méthode" (Late Binding ou **Dynamic Binding**).

Ce mécanisme de recherche dynamique (durant l'exécution de l'application) sert de base à la mise en œuvre de la propriété appelée **polymorphisme**.

CPO

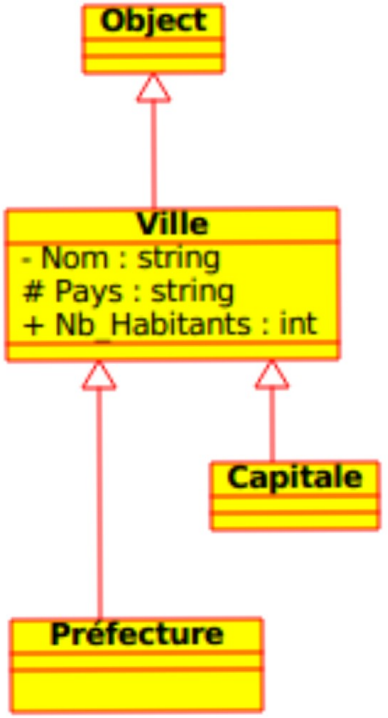
La classe Object

- Toutes les classes n'ayant pas de classe mère héritent de la classe Object.

- Seule la classe Object n'hérite pas d'une autre classe.

- Toutes les classes possèdent donc les méthodes héritées de la classe Object

(toString, equals, hashCode, etc)



CPO

La classe Object

Les méthodes de la classe Object peuvent être redéfinies

→ equals et hashCode seront abordées plus tard

→ toString

```
public class Ville {
}
```

```
public class Ville {
    public String toString()
    {
        return "Ville";
    }
}
```

```
public class Main {
    public static void main(String[]
        Ville v = new Ville();
        System.out.println(v);
    }
}
```

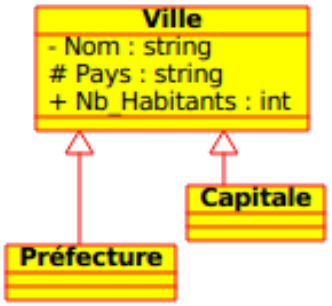
pck1.Ville@428c6e04

Ville

CPO

L'opérateur instanceof

L'opérateur instanceof permet de savoir à quelle classe appartient une instance (un objet).



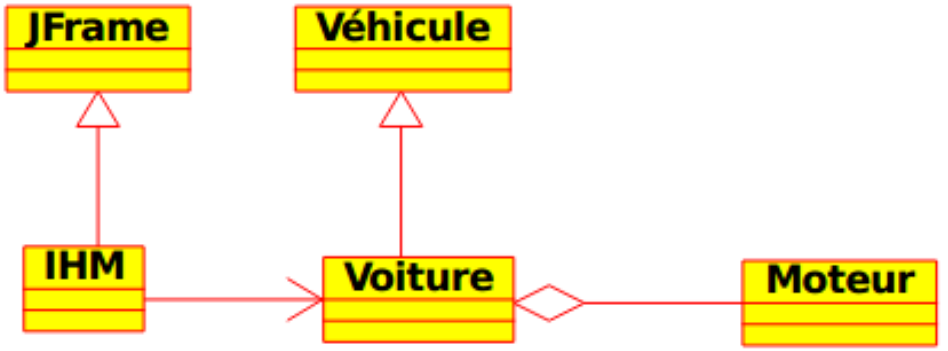

```
Ville []tab2 = new Ville [3];
tab2[0]=new Capitale();
tab2[1]=new Ville();
tab2[2]=new Préfecture();

for (int i=0;i<tab2.length;i++)
{
    if (tab2[i] instanceof Préfecture)
        System.out.println("c'est une préfecture");
    if (tab2[i] instanceof Capitale)
        System.out.println("c'est une capitale");

}
```

run:
c'est une capitale
c'est une préfecture

CPO

Codage des relations entre classes

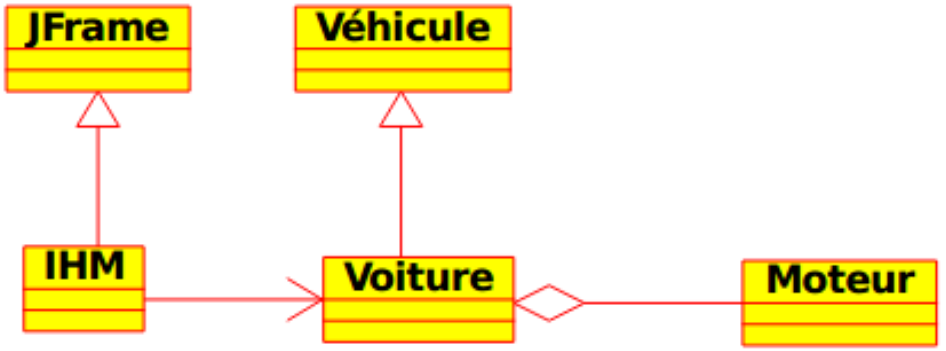
Une interface graphique permet de saisir (et de sauvegarder) les caractéristiques d'un véhicule (une voiture).



a) Quels sont les objets créés par la méthode main() ?
Quel est le lien à établir ?
Coder la méthode main().

CPO

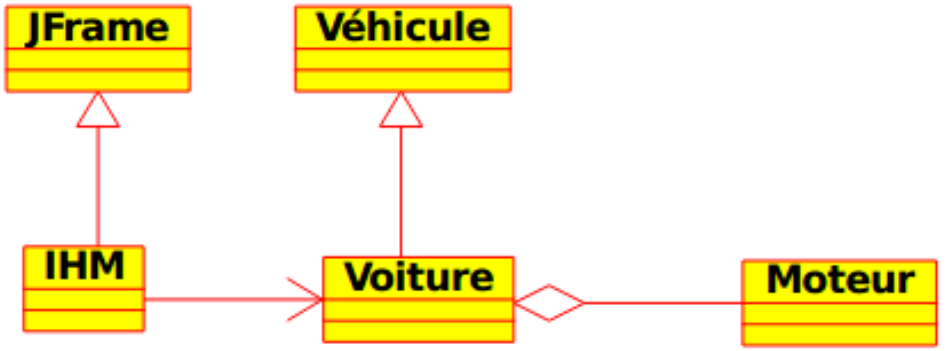
Codage des relations entre classes



b) Écrire la classe IHM
- Sa signature.
- Son attribut.
- Son constructeur.

CPO

Codage des relations entre classes



c) Écrire la classe Voiture :
- Sa signature.
- Son attribut.
- Son constructeur si le constructeur de Voiture crée le moteur.

CPO

Les classes abstraites

- Une classe abstraite est une classe qui ne permet pas d'instancier des objets.

- Elle ne peut servir que de classe de base pour une dérivation.

- Elle peut contenir des méthodes et des attributs dont hériteront les classes filles.

- Elle peut contenir des méthodes abstraites.

CPO
Les classes abstraites

```
abstract class A
{
    //code de la classe
}
```

On peut déclarer une variable de type A :
A a ;
Mais il sera impossible de l'instancier (pas de new). Par contre, si on crée une classe B qui dérive de A, on pourra écrire :
a = new B() ;

```
//Si la classe Navire est abstraite
Navire destr1 = new Navire("test");  //impossible
Navire destr2 = new Destroyer("Le Chacal",25);
```

CPO

Les méthodes abstraites

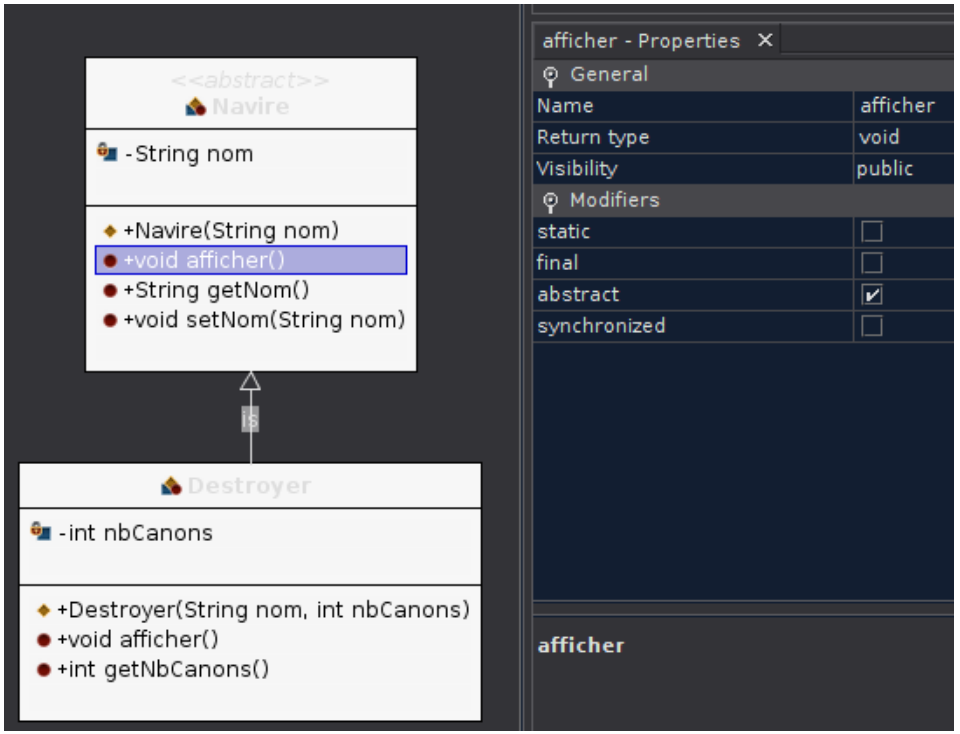
Une méthode abstraite est une méthode **sans corps** (sans code) que la classe fille doit obligatoirement implémenter.

public abstract void f() ;

- Au moins une méthode abstraite → classe abstraite

- Les méthodes abstraites sont toujours publiques

CPO
Les méthodes abstraites




```
public abstract class Navire {
    public abstract void afficher();
    //autres méthodes
}

public class Destroyer extends Navire{
    public void afficher(){
        System.out.println("c'est un destroyer");
    }
   //autres méthodes
}
```

CPO
Les méthodes abstraites

Une classe dérivée d'une classe abstraite n'a pas besoin de redéfinir toutes les méthodes abstraites de sa classe :

```
abstract class A
{
    public abstract void f1();
    public abstract void f2(char c);
    …
}

abstract class B extends A
{
    public void f1() {…}; //définition de f1
    //pas de définition de f2
}
```

Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites.

CPO
Les méthodes abstraites

```
public abstract class Navire {
    private int puissanceDeFeu = 5;
    public abstract void tirer();
    //autres méthodes (dont getter et setter)
}

public class Destroyer extends Navire{
    @Override
    public void tirer() {
        setPuissanceDeFeu(getPuissanceDeFeu()+10);
    }
    //autres méthodes
}

public class SousMarin extends Navire{
    @Override
    public void tirer() {
        setPuissanceDeFeu(9);
    }
    //autres méthodes
```

CPO
Les interfaces

Une interface est une classe dans laquelle on ne retrouve que :
- des méthodes abstraites
- des attributs finaux (constantes)

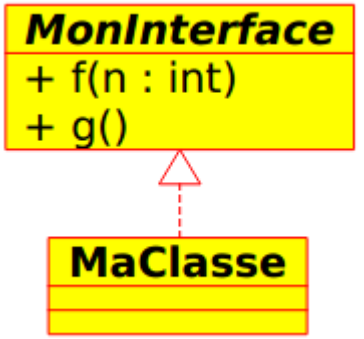
```
public interface MonInterface {
    void f(int n) ;
    void g() ;
    final int MAXI = 100 ;
}
```

Les méthodes d'une interface sont nécessairement publiques et abstraites.

CPO
Les interfaces

- Utilisation avec le mot-clef « implements »
- Les méthodes de l'interface sont nécessairement « abstract » et « public » (ces mots-clefs sont optionnels).

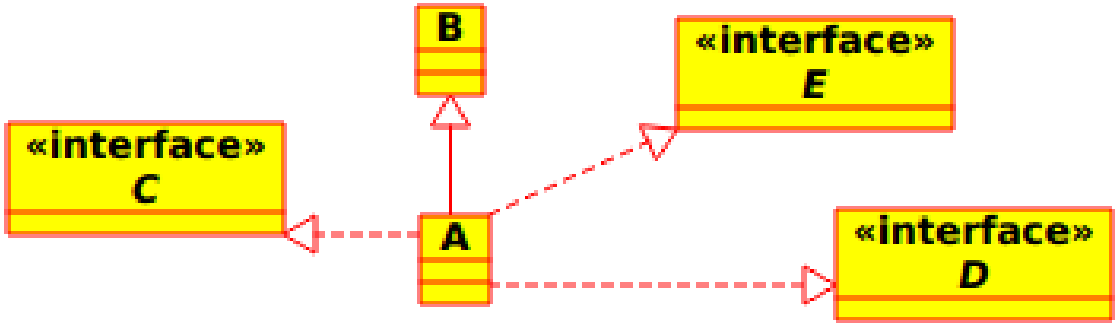
```
public interface MonInterface {
    void f(int n) ;
    void g() ;
    final int MAXI = 100 ;
}

public class MaClasse implements MonInterface {
    void f(int n)
    {
        //à coder
    }
    void g()
    {
        //à coder
    }
}
```



CPO
Les interfaces

Une classe ne peut hériter (extends) qu'une seule fois (d'une classe abstraite ou non).
Une classe peut implémenter (implements) d'autant d'interfaces que souhaité.

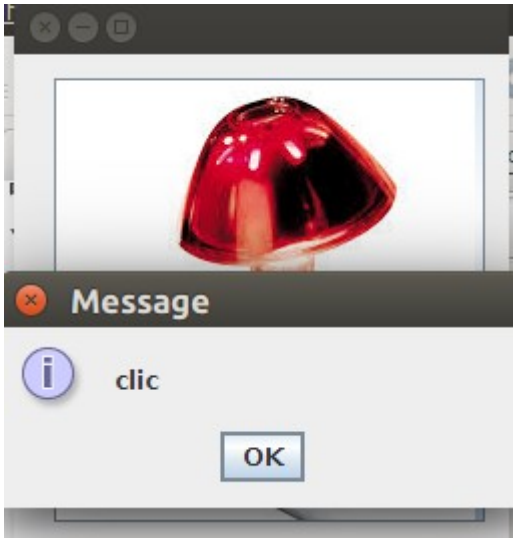
```
public class A extends B implements C,D,E {

}
```



CPO
Les interfaces

Exemple de l'interface MouseListener

```
public class IHM extends javax.swing.JFrame implements MouseListener{
    public IHM() {
        initComponents();
        this.addMouseListener(this);
    }

    private void initComponents() {/*code*/ }

    @Override
    public void mouseClicked(MouseEvent e) {
        JOptionPane.showMessageDialog(this, "clic");
    }

    @Override
    public void mousePressed(MouseEvent e) {   }

    @Override
    public void mouseReleased(MouseEvent e) {    }

    @Override
    public void mouseEntered(MouseEvent e) {   }

    @Override
    public void mouseExited(MouseEvent e) {    }
}
```



CPO

Soit la classe abstraite suivante :

```
abstract class Figure {
    private String nom;
    public Figure(String nom) {
        this.nom = nom;
    }
    public String getNom() {
        return nom;
    }
    public abstract float surface();
    public abstract float perimètre();
}
```

Soit l'interface suivante :

```
interface Affichable{
    void afficheinfos();
}
```

1) Coder la classe Carré qui est une figure affichable possédant un coté passé lors de la création. La méthode afficheinfos permet d'afficher le nom et le coté du carré.

2) Coder la méthode main() qui un carré « c1 » de 5 mètres de coté, affiche ses informations,
sa surface et son périmètre.

CPO
Méthode et attribut statiques

Méthode statique

Une méthode statique n'est associée à aucun objet.
On l'appelle avec le nom de la classe.
L'appel par l'objet est « toléré », donc possible.
Une méthode statique peut accéder à des attributs statiques (voir la suite), mais pas aux autres attributs de la classe.
Les méthodes sqrt(), sin(), cos(), etc, sont des méthodes statiques de la classe Math.
Exemple :

```
public class A
{
    public static void f (int n) { ..... }
    .....
}

//dans le main
A.f(5);    //pas d'objet à instancier
```

CPO
Méthode et attribut statiques

Attribut statique

L'attribut statique est unique, quel que soit le nombre d'objets créés.

```
public class MaClasse {
    private static int valeur;
    public MaClasse()
    {
        valeur++;
    }
    public int getValeur()
    {
        return valeur;
    }
}

//dans le main
      MaClasse m1 = new MaClasse();
       MaClasse m2 = new MaClasse();
       System.out.println(m1.getValeur()+ " "+ m2.getValeur());
```

CPO

```
public class Test {
    private static String nomProvisoire;
    private String nomDéfinitif;

    public String getNomDéfinitif() {
        return nomDéfinitif;
    }

    public void setNomDéfinitif(String nomDéfinitif) {
        this.nomDéfinitif = nomDéfinitif;
    }
    public String getNomProvisoire() {
        return nomProvisoire;
    }
    public void setNomProvisoire(String nomProvisoire) {
        Test.nomProvisoire = nomProvisoire;
    }
}

public class Main {
    public static void main(String[] args) {
    Test t1 = new Test();
    Test t2 = new Test();
    t1.setNomProvisoire("pro1");
    t2.setNomProvisoire("pro2");
    t1.setNomDéfinitif("def1");
    t2.setNomDéfinitif("def2");
    System.out.println(t1.getNomProvisoire()+ " "+t1.getNomDéfinitif()+"
                        "+t2.getNomProvisoire()+" "+t2.getNomDéfinitif());
    }
}
```