Bases de la programmation orientée objet
Polymorphisme

Denis Poitrenaud

IUT de l'Université Paris Descartes
Denis.Poitrenaud@ParisDescartes.fr

DUT Informatique – 2ème Semestre

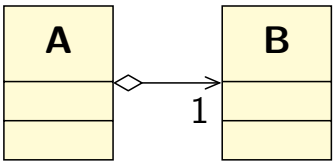
Grandes lignes de ce cours

Polymorphisme par héritage

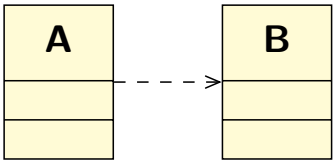
Principe

Rappel – principes de la POO

1. Encapsulation
   - Rapprochement des données (attributs) et traitements (méthodes)
   - Protection de l'information (`private` et `public`)
2. Agrégation (et composition)
   - Classe `A` "A UN" Classe `B`



3. Utilisation
   - Classe `A` "UTILISE" Classe `B`



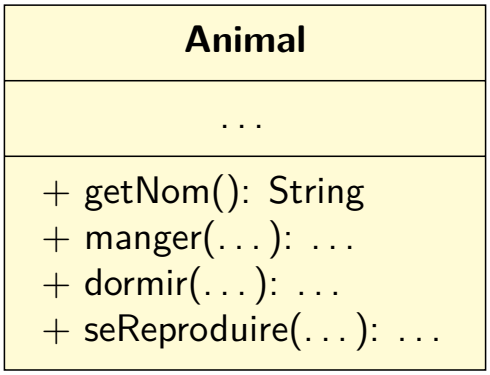
Un nouveau principe POO : Héritage

Soient deux classes A et B

- Relation d'héritage: Classe `B` "EST UN" Classe `A`
  - A est la super-classe ou classe mère de B
  - B est la sous-classe ou classe fille de A
- Exercice: Héritage, Agrégation ou Utilisation ?
  - Cercle et Ellipse ?
  - Salle de bains et Baignoire ?
  - Piano et Joueur de piano ?
  - Entier et Réel ?
  - Personne, Enseignant et Étudiant ?

Motivation pour un nouveau type de relation entre classes

Une classe décrit les services (comportements et données) d'un ensemble d'individus
Exemple : la classe `Animal` (animation graphique)



Problèmes pour implémenter cette spécification :

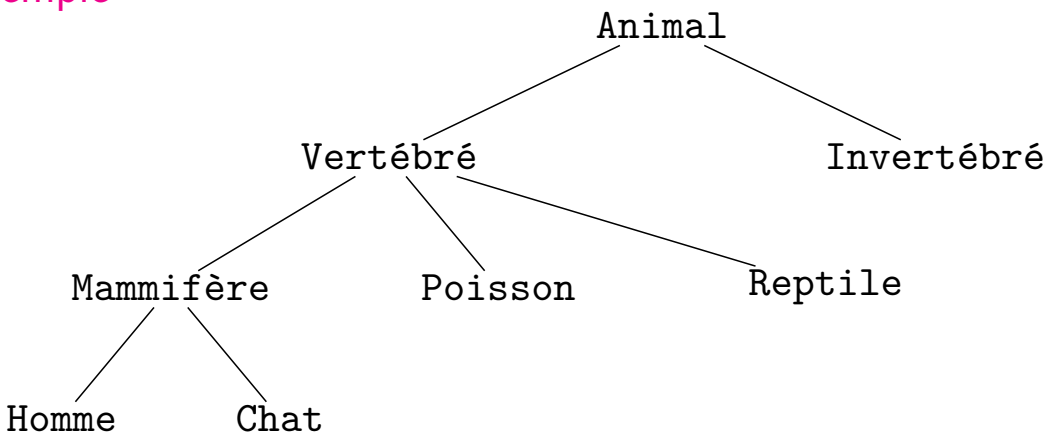
- Elle est trop générale (exemple : `manger()` est très différent selon les animaux)
- Il manque les services spécifiques à certaines catégories d'animaux (exemple : `voler()`, `nager()`, ...)

Les mauvaises solutions

1. Faire que la classe `Animal` puisse vraiment représenter tous les animaux
   - accroître l'interface publique (tous les services possibles de tous les animaux)
   - ajouter des attributs booléens pour les catégories (`isOiseau`, `isPoisson`, ...) ou un attribut qui indique la catégorie
   - tester les attributs pour savoir si on peut répondre à un message
   - Exemple : `Animal unChat = new Animal ("chat",...);` (avec un attribut `genreAnimal` de type `String` dans la classe Animal)
   - code très complexe et non maintenable
2. Faire autant de classes qu'il existe d'espèces animales
   - beaucoup de services vont être très similaires entre les classes, beaucoup de factorisation perdue
3. Faire de la délégation $\Rightarrow$ une bonne solution que nous verrons plus tard

Une solution - Classification hiérarchisée

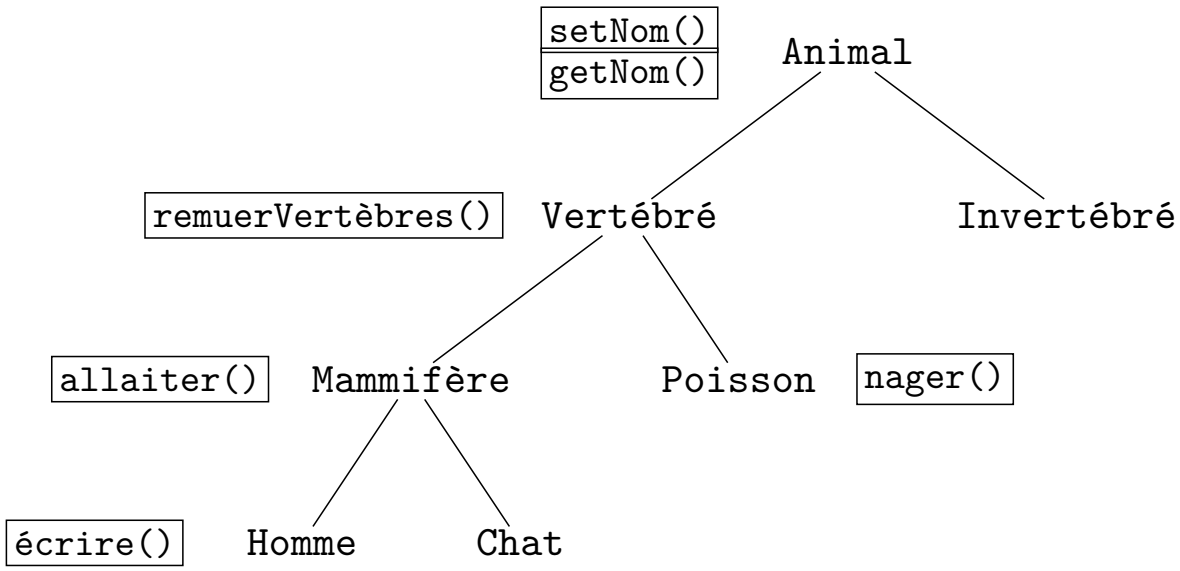
Exemple



- Les sous-classes (spécialisations) de la classe `Mammifère` sont les classes `Homme` et `Chat`
- Les ascendants (généralisations) de la classe `Mammifère` sont les classes `Vertébré` et `Animal`

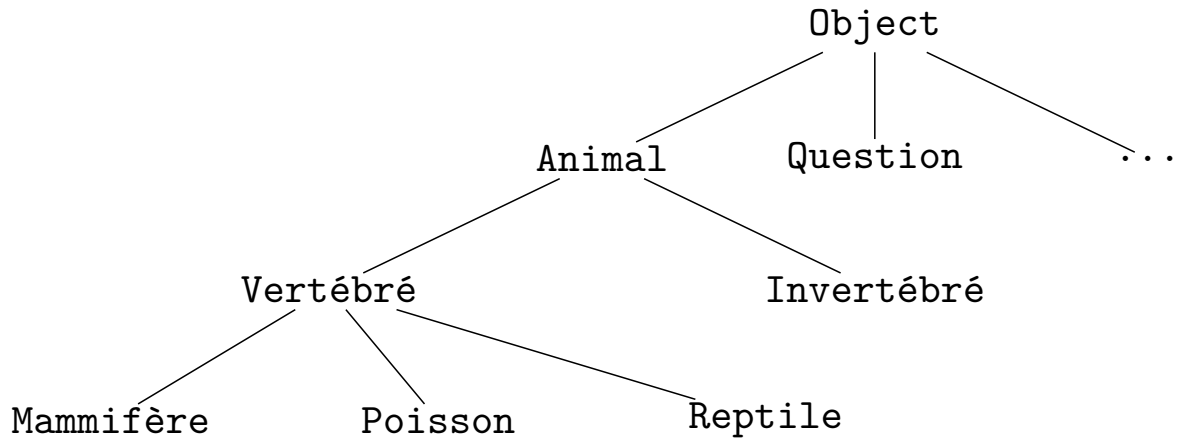
Héritage

- Une sous-classe hérite des services (comportements et données) de ses ascendants
- Pour factoriser au mieux la programmation, il suffit de placer les services à la bonne place dans la relation d'héritage



Hiérarchie en Java

- Une classe hérite toujours d'une seule et unique classe (héritage simple *versus* héritage multiple)
- Cette classe est dite être la super-classe de la sous-classe
- Par défaut, toute classe hérite de la classe `Object` qui est la racine unique de l'arbre d'héritage



Implémentation en Java

```
public class Homme extends Mammifère {
  // attributs propres aux hommes uniquement
  private boolean droitier;
  ...
  public void écrire() {
    if (droitier) {...} else {...}
  }
}
```

```
Homme unHomme = new Homme(...);
// appel d'un service propre aux hommes
unHomme.écrire();
// appel d'un service commun aux vertébrés (hérité)
unHomme.remuerVertèbres();
// appel d'un service commun aux animaux (hérité)
unHomme.setNom("Adam");
```

Héritage de comportements

Durant l'exécution, l'instruction :

```
unHomme.setNom("Adam");
```

provoque :

- La recherche de `setNom()` dans la classe `Homme`
- Puisse qu'elle n'existe pas, la recherche se poursuit en remontant l'arbre d'héritage jusqu'à trouver `Animal.setNom()`
- `Animal.setNom()` est exécutée (dans le contexte de `unHomme`)

Si la méthode `Animal.setNom()` n'était pas définie, une erreur serait détectée lorsque la classe `Object` serait atteinte

Héritage d'attributs

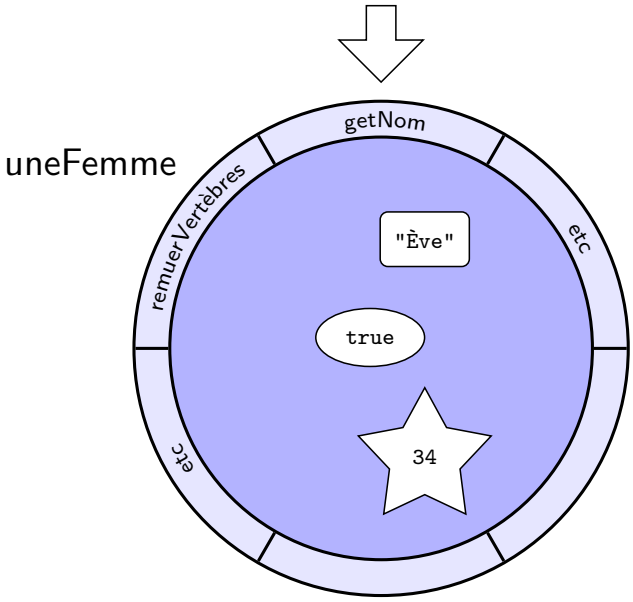
```
public class Animal {
  // attributs propres à tous les animaux
  private String nom; ...
  public String getNom(){ return nom; }
  public void setNom(String n) { nom = n; }
  ...
}
public class Vertébré extends Animal {
  // attributs propres aux Vertébrés
  private int nbVertèbres; ...
  public void setNbVertèbres(int n) {
    nbVertèbres = n;
  }
  public void remuerVertèbres( ){
    System.out.println (this.getNom() + "remue"
                        + nbVertèbres + "vertèbres");
  }
}
```

Héritage d'attributs – suite

```
public class Homme extends Mammifère {
  // attributs propres aux hommes uniquement
  private boolean droitier;
  ...
  public Homme (String unNom, boolean droitier) {
    this.setNbVertebres(34);
    this.setNom(unNom);
    this.droitier = droitier;
  }
  ...
}
```


Héritage d'attributs – fin

```
Homme uneFemme = new Homme("Ève", true);
```




```
uneFemme.remuerVertèbres();
System.out.println(uneFemme.getNom());
```

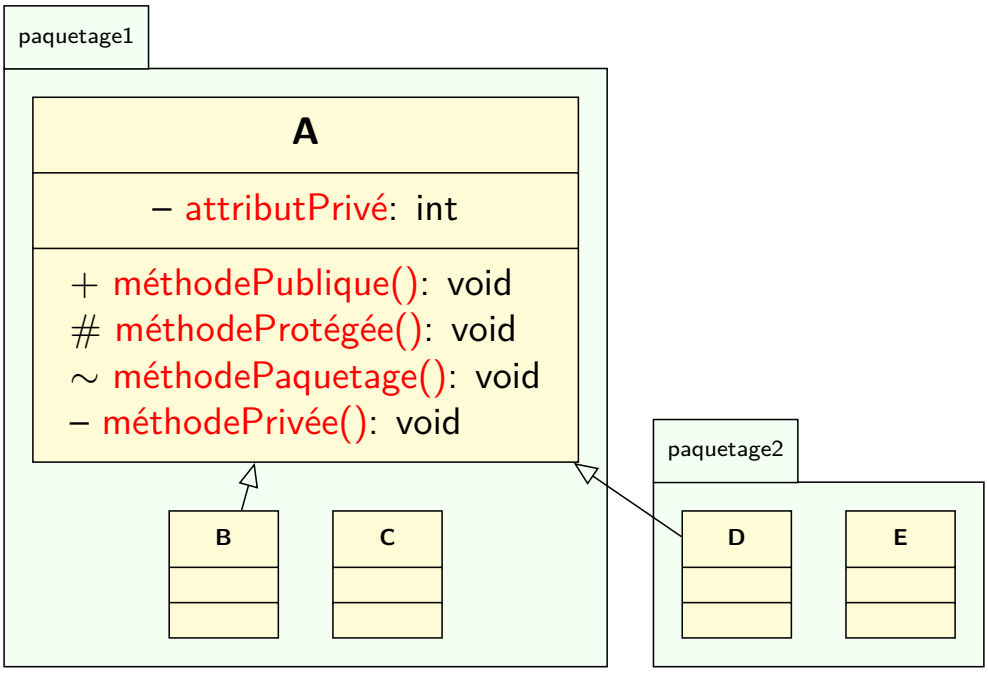
Héritage et encapsulation

- Les membres (attributs et méthodes) déclarés privés dans une classe ne sont pas visibles dans une sous-classe
- En d'autres termes, l'héritage n'implique pas la visibilité

```
public class Homme extends Mammifère {
  // attributs propres aux hommes uniquement
  private boolean droitier;
  ...
  public Homme (String unNom, boolean droitier) {
    this.setNbVertebres(50);
    nom = unNom; // erreur
    this.droitier = droitier;
  }
  ...
}
```

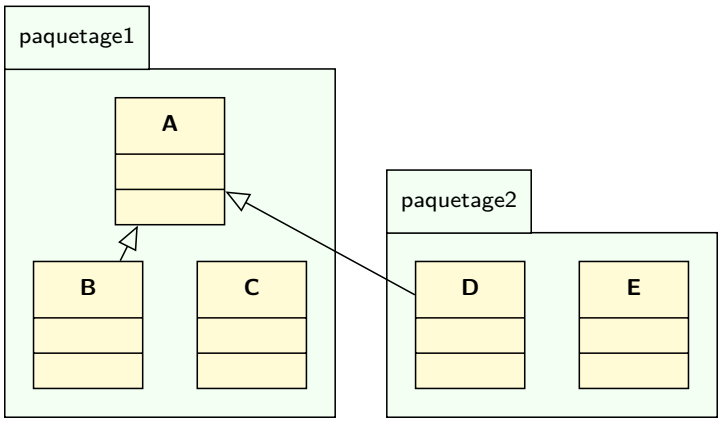

Visibilité

Le modificateur qui précède chaque méthode détermine sa visibilité



Les attributs sont toujours déclarés privés (encapsulation)

Visibilité – suite



	a : A (*déclaré dans . . .*)	A	B	C	D	E
`private`	a.attributPrivé	✓				
`public`	a.méthodePublique()	✓	✓	✓	✓	✓
`protected`	a.méthodeProtégée()	✓	✓	✓	✓	
rien	a.méthodePaquetage()	✓	✓	✓		
`private`	a.méthodePrivée()	✓				

Héritage et construction

```
public class Homme extends Mammifère {
  private boolean droitier;
  ...
  public Homme (String unNom, boolean droitier) {
    this.setNbVertèbres(34);
    this.setNom(unNom);
    this.droitier = droitier;
  }
  ...
}
```

Problèmes

- Obligation d'introduire des modificateurs tel que `setNbVertèbres()`
- Redondance (chaque constructeur des sous-classes de `Animal` doit faire un appel à `setNom()`) → Risque d'erreurs/oublis

La super-classe `Mammifère` doit avoir un constructeur

Héritage et construction – Correction

```
public class Homme extends Mammifère {
  private static final int NB_VERTEBRES = 34;
  private boolean droitier;

  public Homme (String unNom, boolean droitier,
                                  int nbVertèbres) {
    // super doit être la première instruction
    super(unNom, nbVertèbres);
    this.droitier = droitier;
  }

  public Homme (String unNom, boolean droitier) {
    this(unNom, droitier, NB_VERTEBRES);
  }
}
```


Constructeur implicite

Si aucun constructeur de la classe ou de la super-classe n'est invoqué, le compilateur ajoute un appel au constructeur sans arguments de la super-classe

```
public class A extends B {
  public A(int x) {
    // appel super() implicite
    this.x = x;
    ...
  }
}
```

Attention : Dans ce cas, le constructeur sans arguments doit être défini dans la super-classe (ou elle doit comporter aucun constructeur)

Héritage et spécialisation

Une sous-classe peut *spécialiser* une méthode héritée

- Il suffit de déclarer au sein de la sous-classe une nouvelle méthode ayant la même signature que la méthode héritée

```
public class Homme extends Mammifère {
  ...
  public void manger() {
    System.out.println("je␣mange");
  }
}

public class HommeCivilisé extends Homme {
  ...
  public void manger() {
    System.out.println("je␣prends␣mes␣couverts");
    super.manger();
  }
}
```

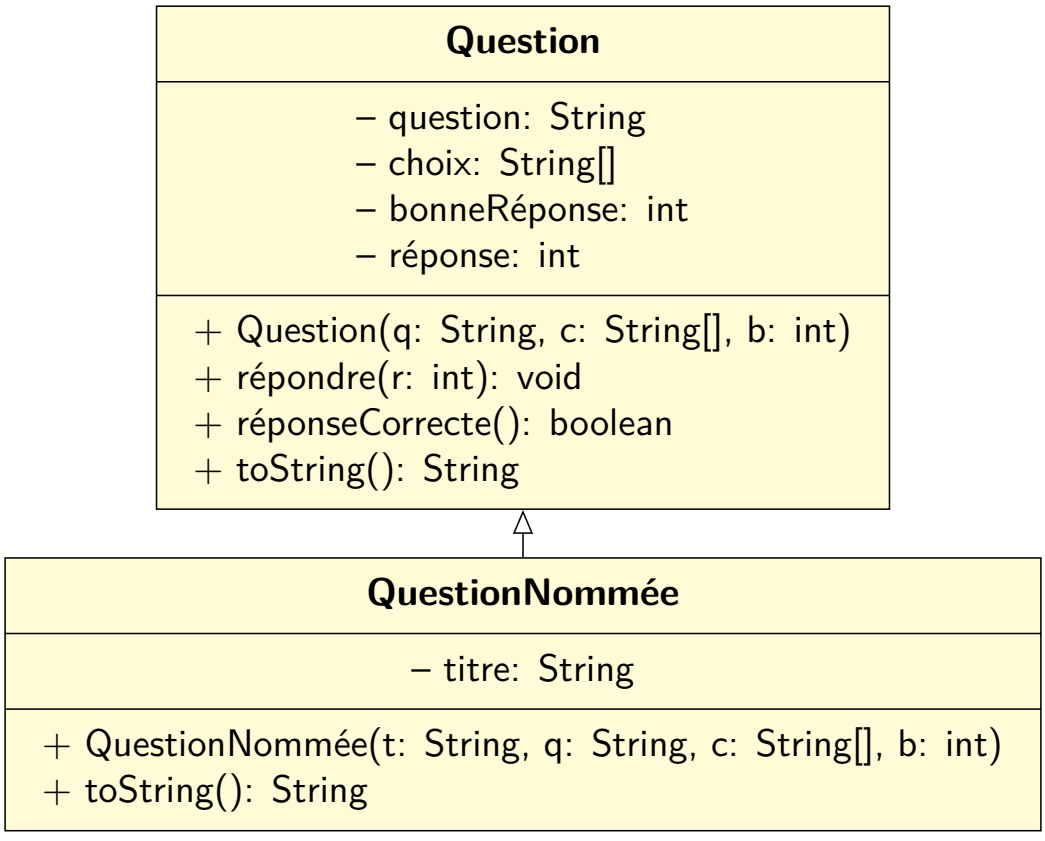

Un exemple complet

```
public class Question {
  private String question;
  private String[] choix;
  private int bonneRéponse, réponse;

  public Question(String question,
                  String[] choix, int bonneRéponse) {...}

  public void répondre(int réponse) {...}

  public boolean réponseCorrecte() {...}

  public String toString() {...}
}
```

Un exemple complet – suite

```
public class QuestionNommée extends Question {
  private String titre;

  public QuestionNommée(String titre, String question,
                        String[] choix, int bonneRéponse) {
    super(question, choix, bonneRéponse);
    this.titre = titre;
  }

  public String toString() {
    return "[" + titre + "] " + super.toString();
  }
}
```


Diagramme UML - Héritage



Sous-Typage et polymorphisme

Principe POO : Sous-typage $\Longrightarrow$ Polymorphisme

Une sous-classe est un sous-type de sa super-classe et ses ascendants

- Classe `B` "EST-UN" Classe `A`
  - Toute méthode `m` de `A` peut-être invoquée sur une instance de la classe `B` (+ transitivité sur l'héritage de `A`)
  - *Subsomption* : Dans toute expression "qui attend" un `A`, un `B` peut être fourni à la place

Le polymorphisme consiste à exploiter la subsomption

```
// un homme est un animal
Animal anim1 = new Homme("Caïn", false);
// un chat aussi
Animal anim2 = new Chat("Européen");
System.out.println(anim1.getNom());
System.out.println(anim2.getNom());
// un animal n'est pas nécessairement un homme
Homme h = anim1; // Erreur
```

Exemple plus intéressant

```
public class AppliQuestion {
  private static void bilan(Question[] tab) {
    for (Question q : tab)
      if (q.réponseCorrecte())
        System.out.println("ok");
      else
        System.out.println("ko");
  }
  ...
}
```


Exemple plus intéressant – suite

```
public class AppliQuestion {
  ...
  public static void main(String[] args) {
    Question[] t = new Question[3];
    String[] rep = { "Blanc", "Noir" };

    t[0] = new Question("Noir␣c'est", rep, 2);

    t[1] = new QuestionNommée("Débile",
                  "Le␣cheval␣blanc␣est", rep, 1);

    t[2] = new QuestionNommée("Difficile",
                   "La␣mariée␣est␣en", rep, 1);
    bilan(t);
  }
}
```

Encore plus intéressant

```
public class AppliQuestion {
  private static void affiche(Question[] tab) {
    for (Question q : tab)
      System.out.println(q.toString());
  }
  ...
}
```

Si q référence une `QuestionNommée`, quelle méthode `toString()` va être invoquée ?

Spécialisation et liaison dynamique

Si q référence une `QuestionNommée`, quelle méthode `toString()` va être invoquée ?

- La méthode `toString()` est définie dans `Question`
- Mais elle est *spécialisée* dans `QuestionNommée`

C'est la version la plus spécialisée (`QuestionNommée.toString()`) qui est invoquée car la recherche de la méthode débute dans la classe effective de l'objet référencé par q

La recherche est menée à l'exécution et ce mécanisme est appelé la *liaison dynamique*

Conséquence

La classe QCM (vue au cours précédent) peut être employée indistinctement avec des objets `Question` et/ou `QuestionNommée`

```
public class QCM {
  private ArrayList<Question> questions
              = new ArrayList<Question>();

  public void ajouter(Question q) { ... }
  public int nbQuestions() { ... }
  public void répondre(int q, int r) { ... }
  public void afficher(int q) { ... }
  public int score() {
    int s = 0;
    for (Question q : questions)
      if ( q.réponseCorrecte() ) ++s;
    return s;
  }
}
```


Utilisation

```
public class AppliQCM {
  public static void main(String[] args) {
    QCM qcm = new QCM();
    String[] rep = { "titi", "toto", "tata" };
    Question q =
        new Question("Gros␣minet␣et", rep, 1);
    qcm.ajouter(q);
    q = new QuestionNommée("Math.",
                    "0␣+␣0␣=␣la␣tête␣à", rep, 2);
    qcm.ajouter(q);
    Scanner s = new Scanner(System.in);
    for (int q = 0; q < qcm.nbQuestions(); ++q) {
      qcm.afficher(q);
      System.out.print("Votre␣réponse␣:␣");
      qcm.répondre(q, s.nextInt());
    }
    System.out.println("Score␣:␣" + qcm.score());
  }
}
```

Exercice – Qu'affiche ce programme ?

```
public class Fruit {
  public void af(Fruit f) {
    System.out.print(1);
  }
}

public class Kiwi
       extends Fruit {
  // spécialisation
  public void af(Fruit f) {
    System.out.print(2);
  }

  // surcharge
  public void af(Kiwi k) {
    System.out.print(3);
  }
}
```

```
public class Appli {
  public static
    void main(String[] a) {
    Fruit f  = new Fruit();
    Fruit fk = new Kiwi();
    Kiwi  k  = new Kiwi();

    f.af(f); f.af(fk);
    f.af(k);

    fk.af(f); fk.af(fk);
    fk.af(k);

    k.af(f); k.af(fk);
    k.af(k);
  }
}
```


Écrire du code polymorphe

Spécification d'une classe Cage pour la gestion d'un zoo

```
Cage uneCage1 = new Cage(...);
Cage uneCage2 = new Cage(...);
Lion unLion = new Lion(...);
uneCage1.accueillir(unLion);
Singe unSinge = new Singe (...);
uneCage2.accueillir(unSinge);

public class Cage {
  public void accueillir(Lion l) {...}
  public void accueillir(Singe s) {...}
  // etc.
}
```

- Si une nouvelle espèce animale doit être prise en compte, il faudra modifier le code de la classe Cage

⇒ Très mauvaise solution

Écrire du code polymorphe – correction

```
public class Cage {
  ...
  public void accueillir( Animal a) {
    System.out.println(a.getNom( ) + "est␣en␣cage");
    ...
  }
  ...
}
```


Protocole et polymorphisme

Motivation

Nouvelle spécification à prendre en compte : *tous les animaux ne peuvent pas aller en cage*

La méthode `Cage.accueillir()` doit être à même de détecter les animaux ne pouvant l'être

```
Cage uneCage1 = new Cage(...);
Homme unHomme = new Homme(...);
uneCage.accueillir(unHomme); ⟹ il refuse !!!
```


Première solution

```
public class Cage {
  ...
  public void accueillir(Animal a) {
    if ( a instanceof Homme )
      System.out.println(a.getNom( )
                  + "refuse d'aller en cage");
      return;
    ...
  }
}
```

- Des connaissances propres à la classe `Homme` sont dans la classe `Cage`
- si une nouvelle espèce animale refuse d'aller en cage, il faudra modifier le code de la classe `Cage`

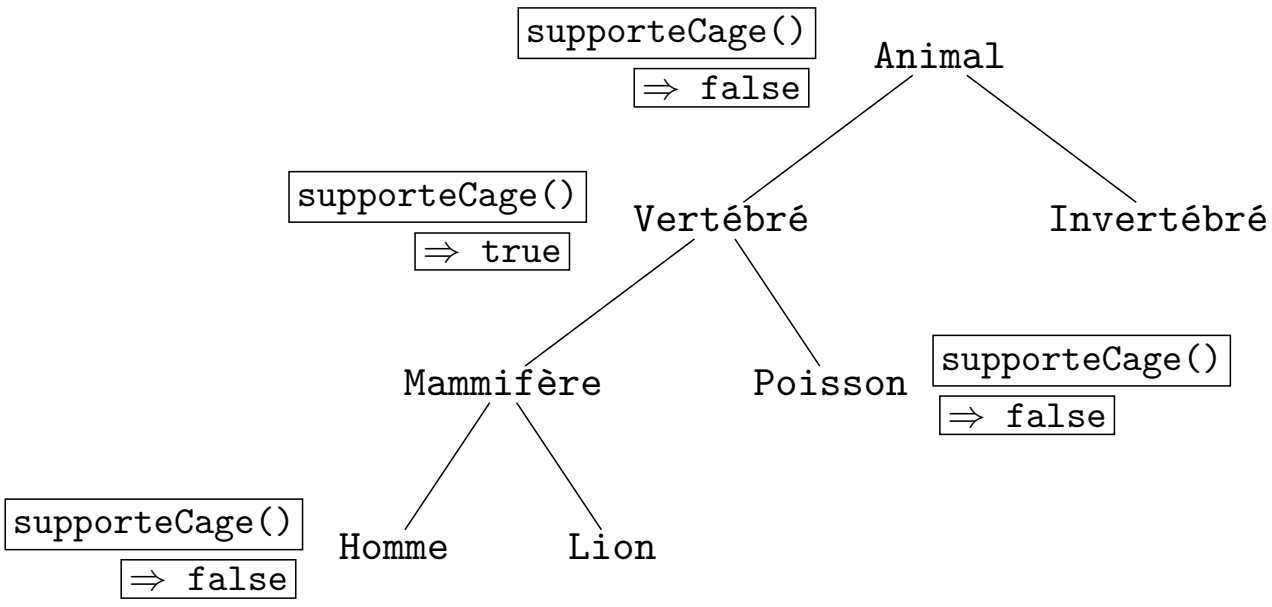
⇒ Très mauvaise solution

Mise en place d'un protocole

```
public class Cage {
  ...
  public void accueillir (Animal a) {
    if (!a.supporteCage()) {
      System.out.println(a.getNom( )
                  + "refuse d'aller en cage");
      return;
    }
    ...
  }
}
```

- Tout animal doit pouvoir répondre à ce protocole
- Nous allons implémenter le protocole dans la hiérarchie de racine `Animal`, en utilisant l'héritage et en spécialisant lorsque c'est nécessaire

Vue de l'arbre d'héritage



Mise en place d'un protocole

```
public class Animal {
  ...
  public boolean supporteCage() { return false; }
}
```

```
public class Vertébré extends Animal {
  ...
  public boolean supporteCage() { return true; }
}
```

```
public class Homme extends Mammifère {
  ...
  public boolean supporteCage() { return false; }
}
```

```
  ...
  uneCage.accueillir(unLion);// Vertébré.supporteCage()
```


Retour sur les protocoles standards

Le programme suivant est valide

```
public class AppliQuestion {
  public static void main(String[] args) {
    String[] rep = { "Blanc", "Noir" };
    Question q1 = new Question("Noir␣c'est", rep, 2);
    System.out.print(q1);

    Question q2=new Question("La␣mariée␣est␣en",rep,1);
    if (q1.equals(q2))
      System.out.println("bizarre");
  }
}
```

- À l'exécution, pourquoi `System.out.print(q1)` provoque un appel `q1.toString()` ?
- La classe `Question` ne dispose pas de méthode `equals()`, pourquoi n'y a-t-il pas d'erreur de compilation ?

Le protocole toString()

```
public class System {
  public static PrintStream out;
  ...
}
```

```
public class PrintStream {
  public void print (Object arg) {
    print(arg.toString()); }
  ...
}
```

```
public class Object {
  public String toString () {
    return getClass().getName() +
             "@" +Integer.toHexString(hashCode()); }
  ...
}
```


Le protocole toString() – suite

- La classe `Object` est la racine de l'unique arbre d'héritage
- La classe `Question` hérite (implicitement) de la classe `Object`
- Il suffit de spécialiser la méthode `toString()` dans vos classes pour qu'un objet de type `PrintStream` (tel que `System.out`) puisse afficher vos instances

```
public class Question  {
  ...
  public String toString() {
    String s = question + " : ";
    ...
    return s;
  }
}
```

Le protocole equals()

La classe `Object` dispose d'une méthode `equals()`

```
public class Object {
  public boolean equals (Object arg) {
    return this == arg;
  }
  ...
}
```

Elle ne retourne `true` que si les deux références désignent le même objet

Le protocole equals() – suite

Le protocole `equals()` est employé dans les paquetages standards pour comparer les objets entre eux

Exemple

`java.util.ArrayList` (extrait de la doc.)

```
public boolean contains(Object o)
```

Returns `true` if this list contains the specified element. More formally, returns `true` if and only if this list contains at least one element e such that

```
(o==null ? e==null : o.equals(e))
```

Spécialisation de equals() dans vos classes

```
public class Coordonnée {
  private int x, y;
  ...
  public boolean equals(Object obj) {
    // test sur les références
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    // test sur les classes
    if (this.getClass() != obj.getClass())
      return false;
    // test sur les données
    Coordonnée other = (Coordonnée) obj;
    return (this.x == other.x && this.y == other.y);
  }
}
```


Classes et méthodes abstraites

Motivation

```
public class Animation {
   ...
   public void faireManger(Animal a , Nourriture n) {
      ...
      a.manger(n);
      ...
   }
   ...
}
```

- Il faut introduire une méthode `manger()` dans la classe `Animal` pour que cela compile
- problème $\Longrightarrow$ quel comportement y décrire puisque la façon de manger dépend de l'espèce animale ?

Spécification d'un protocole sans implémentation par défaut

Spécification du protocole – *le plus haut possible dans l'arbre d'héritage*

```
public abstract class Animal {
   ...
   public abstract void manger(Nourriture n);
   // pas de code associé
   ...
}
```

Spécialisation du protocole – *là où c'est nécessaire dans l'arbre d'héritage*

```
public class Lion extends Mammifère {
   ...
   public void manger(Nourriture n) {
      ... // code spécifique aux lions
   }
   ...
}
```

Méthodes abstraites $\Longrightarrow$ Classe abstraite

- Une classe qui contient une méthode abstraite est une classe abstraite (et cela doit être explicitement précisé par `abstract class`)
- Une classe abstraite peut contenir des méthodes concrètes
- Une classe abstraite ne peut pas être instantiée (pour pouvoir disposer d'un objet, son comportement doit être complètement défini)

```
Animal unAnimal; // OK
unAnimal = new Animal (...); // ERREUR
```

- Une sous-classe d'une classe abstraite peut
  - implémenter toutes les méthodes abstraites de sa super-classe et alors elle peut être concrète
  - ne pas implémenter toutes ces méthodes et alors elle est nécessairement abstraite (`abstract class`)

Un exemple

```
public abstract class Question {
  private String question;

  public Question(String question) {
    this.question = question;
  }

  public abstract void répondre(String réponse);
  public abstract void corriger();
  public abstract boolean réponseCorrecte();

  public int score() {
    return réponseCorrecte() ? 1 : 0;
  }

  public String toString() {
    return question;
  }
}
```

Une première concrétisation

```
public class QuestionOuverte extends Question {
  private String réponse; private boolean ok;

  public QuestionOuverte(String question) {
    super(question);
    this.ok = false; this.réponse = "";
  }

  public void répondre(String réponse) {
    this.réponse = réponse;
  }

  public void corriger() {
    System.out.println(this + réponse);
    System.out.print("la␣réponse␣est␣correcte␣?␣:␣");
    ok = (new Scanner(System.in)).nextBoolean();
  }

  public boolean réponseCorrecte() { return ok; }
}
```


Une deuxième concrétisation

```
public class QuestionFermée extends Question {
  private String[] choix;
  private int bonneRéponse, réponse;

  public QuestionFermée(String question,
              String[] choix, int bonneRéponse) {
    super(question);
    assert (choix.length >= 2);
    assert (1 <= bonneRéponse);
    assert (bonneRéponse <= choix.length);
    this.choix = choix;
    this.bonneRéponse = bonneRéponse;
    this.réponse = 0;
  }
  ...
}
```

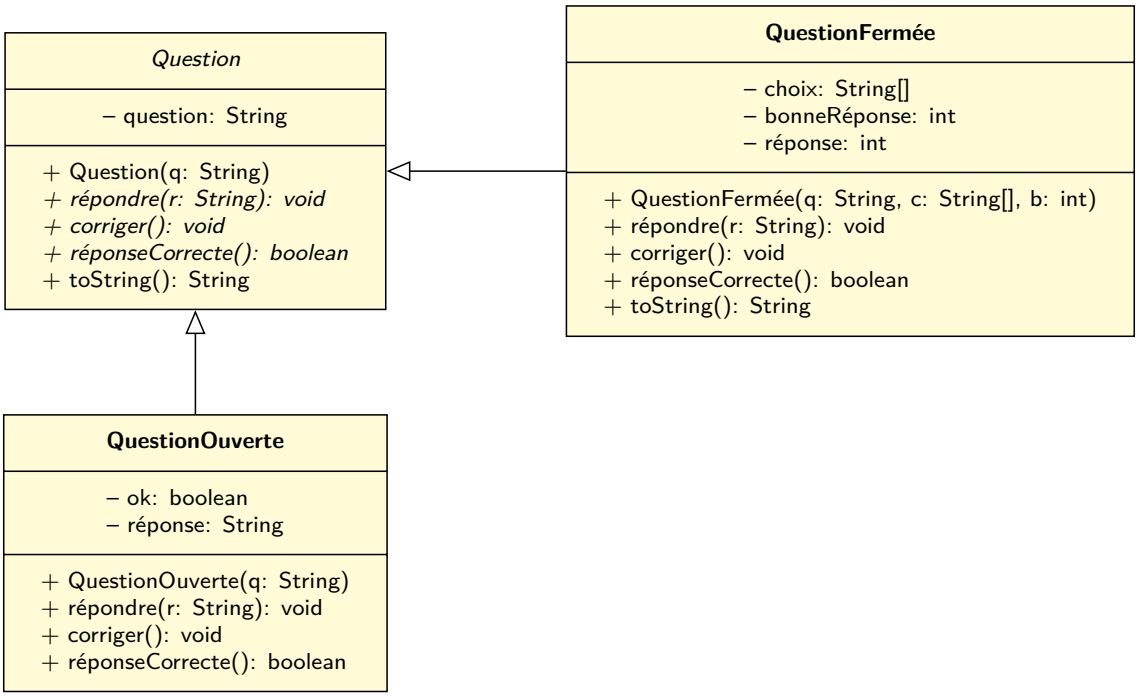
Une deuxième concrétisation – suite

```
public class QuestionFermée extends Question {
  ...
  public void répondre(String réponse) {
    // conversion de la chaîne en entier
    this.réponse = Integer.parseInt(réponse);
  }

  public void corriger() {
    System.out.println("correction␣automatique");
  }

  public boolean réponseCorrecte() {
    return réponse == bonneRéponse;
  }
  ...
}
```


Une deuxième concrétisation – fin

```
public class QuestionFermée extends Question {
  ...
  public String toString() {
    String s = super.toString() + "␣:␣";
    int i = 1;
    for (String r : choix) {
      if (i != 1)
        s += ",␣";
      s += i + "␣-␣" + r;
      ++i;
    }
    return s;
  }
}
```

Diagramme UML - Héritage et abstraction



Utilisation

```
public class AppliQuestion {
  public static void main(String[] args) {
    ArrayList<Question> quest
            = new ArrayList<Question>();

    quest.add(new QuestionOuverte(
              "Quel est l'âge du capitaine ?"));

    String[] rep = { "Blanc", "Noir" };
    quest.add(new QuestionFermée(
              "Le cheval blanc est", rep, 1));
    ...
```

Utilisation – suite

```
    ...
    for (Question q : quest) {
      System.out.println(q);
      System.out.print("Votre réponse : ");
      Scanner sc = new Scanner(System.in);
      q.répondre(sc.next());
    }

    int sc = 0;
    for (Question q : quest) {
      q.corriger();
      if (q.réponseCorrecte())
        ++sc;
    }
    System.out.println("Score : " + sc);
  }
}
```


Exemples du standard

Extrait de la documentation de **`java.lang.Number`** :

- The abstract class `Number` is the superclass of classes `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short`.
- Subclasses of `Number` must provide methods to convert the represented numeric value to `byte`, `double`, `float`, `int`, `long`, and `short`.

```
public abstract int intValue();
public abstract long longValue();
public abstract float floatValue();
...
```

Héritage et conception

Du cahier des charges aux classes

Cahier des charges

- on souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :
  - Figures simples : Point, Segment, Droite
  - Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
  - Courbes : Béziers, Lignes brisées, Cercles, Ellipses
  - et le logiciel pourra être étendu en ajoutant de nouveaux types de figure
- Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sur afficher les figures

Concepts abstraits

Concepts abstraits en rouge

- on souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :
  - Figures simples : Point, Segment, Droite
  - Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
  - Courbes : Béziers, Lignes brisées, Cercles, Ellipses
  - et le logiciel pourra être étendu en ajoutant de nouveaux types de figure
- Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sur afficher les figures

Concepts concrets

Concepts concrets en bleu

- on souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :
  - Figures simples : Point, Segment, Droite
  - Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
  - Courbes : Béziers, Lignes brisées, Cercles, Ellipses
  - et le logiciel pourra être étendu en ajoutant de nouveaux types de figure
- Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sur afficher les figures

Traitements

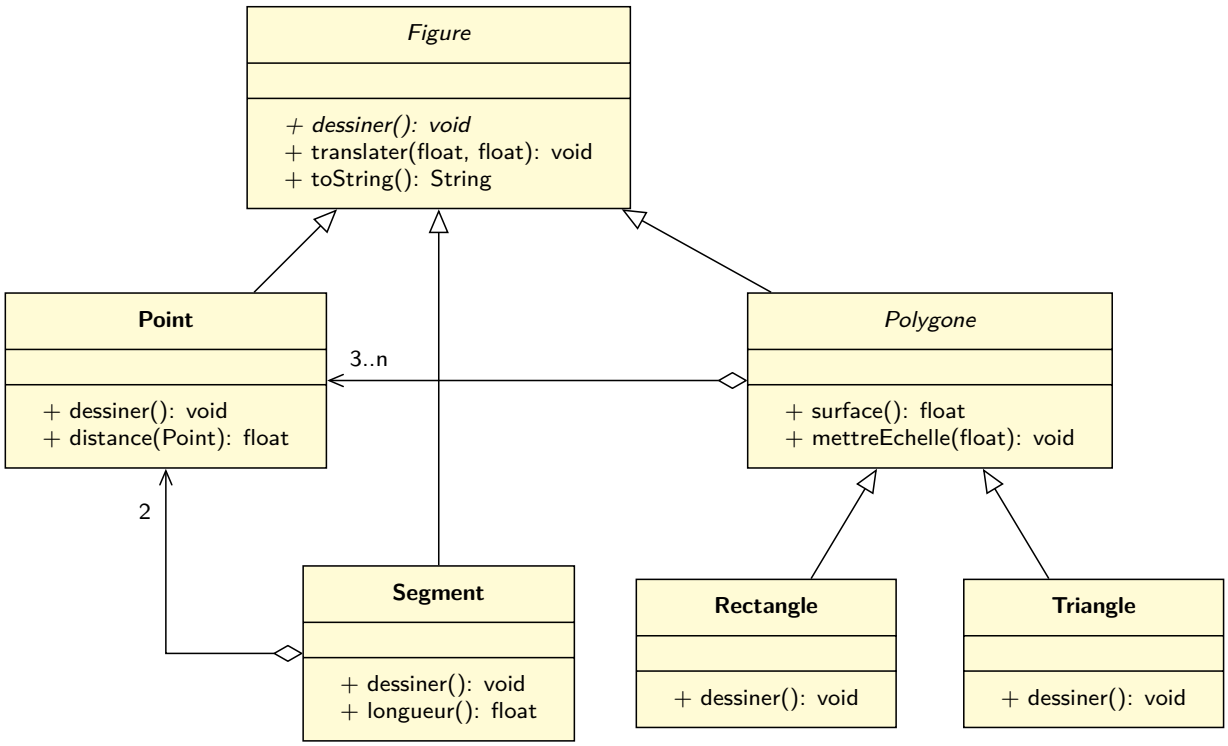
Traitements en vert

- on souhaite réaliser un logiciel de géométrie permettant de construire et manipuler des figures géométriques. Il faudra pouvoir intégrer différents types de figures :
  - Figures simples : Point, Segment, Droite
  - Polygones : Triangle, Carré, Losange, Rectangle, Parallélépipède
  - Courbes : Béziers, Lignes brisées, Cercles, Ellipses
  - et le logiciel pourra être étendu en ajoutant de nouveaux types de figure
- Nous souhaitons pouvoir transformer les figures : translater, mettre à l'échelle, calculer des distances et des surfaces (figures fermées), et bien sur afficher les figures

Résultat: modèle objet

- Concepts abstraits $\Longrightarrow$ Classes abstraites + héritage éventuel
  - Figure, Polygone hérite de Figure, Courbe hérite de Figure
- Concepts concrets $\Longrightarrow$ Classes concrètes + héritage
  - Point hérite de Figure (idem Segment, Droite, LigneBrisée)
  - Triangle hérite de Polygone (idem Carré, Rectangle, etc.)
  - etc.
- Traitements $\Longrightarrow$ Méthodes (privilégier les classes mères)
  - afficher et translater dans Figure
  - calculer la distance dans Point
  - calculer la longueur dans Segment
  - mettre à l'échelle, calculer la surface dans Polygone

Diagramme de classes



Polymorphisme par interface

Motivation

Problème à résoudre

1. Spécifier un ensemble minimal de services communs que devront offrir des classes (très utile dans le cadre du développement en équipe)
2. Utiliser des objets sans connaître leurs types réels
3. Faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage

Les classes abstraites permettent de couvrir les points 1 et 2 mais pas le point 3

Solution

Définition d'un type complètement abstrait : une **interface**

Exemple

Déclaration

```
public interface Prédateur {
  boolean chasseProie(Proie p);
  void mangeProie(Proie p);
}
```

Implémentation

```
public class Lion extends Mammifère
                          implements Prédateur {
  ...
  public boolean chasseProie(Proie p) {
    ...
  }
  public void mangeProie (Proie p) {
    ...
  }
}
```

Règles

- Les méthodes d'une interface sont implicitement `public` et `abstract`
- Une interface peut être vue comme une classe totalement abstraite (i.e. ne comportant que des méthodes abstraites)
- Si une classe implémente une interface mais pas toutes ses méthodes, elle doit être déclarée comme étant `abstract` (ses sous-classes auront la responsabilité d'implémenter les méthodes manquantes)
- Si une classe implémente une interface alors ses sous-classes héritent de cette capacité

Sous-typage et polymorphisme

Une classe qui implémente une interface est un sous-type de celle-ci

- Partout où est attendue une référence d'un type interface, une instance d'une classe implémentant cette interface (ou d'une de ses sous-classes) peut être utilisée $\Longrightarrow$ polymorphisme

```
public class AppliProie {
  ...
  // le 1er qui attrape la proie, la mange
  public static void concours(
            Prédateur[] prédateurs, Proie proie) {
    for (Prédateur prédateur : prédateurs)
      if (prédateur.chasseProie(proie)) {
        prédateur.mangeProie(proie);
        break;
      }
}
```

Héritage multiple

- Une classe peut implémenter plusieurs interfaces

```
public class Grenouille
             implements Prédateur, Proie {
   ...
}
```

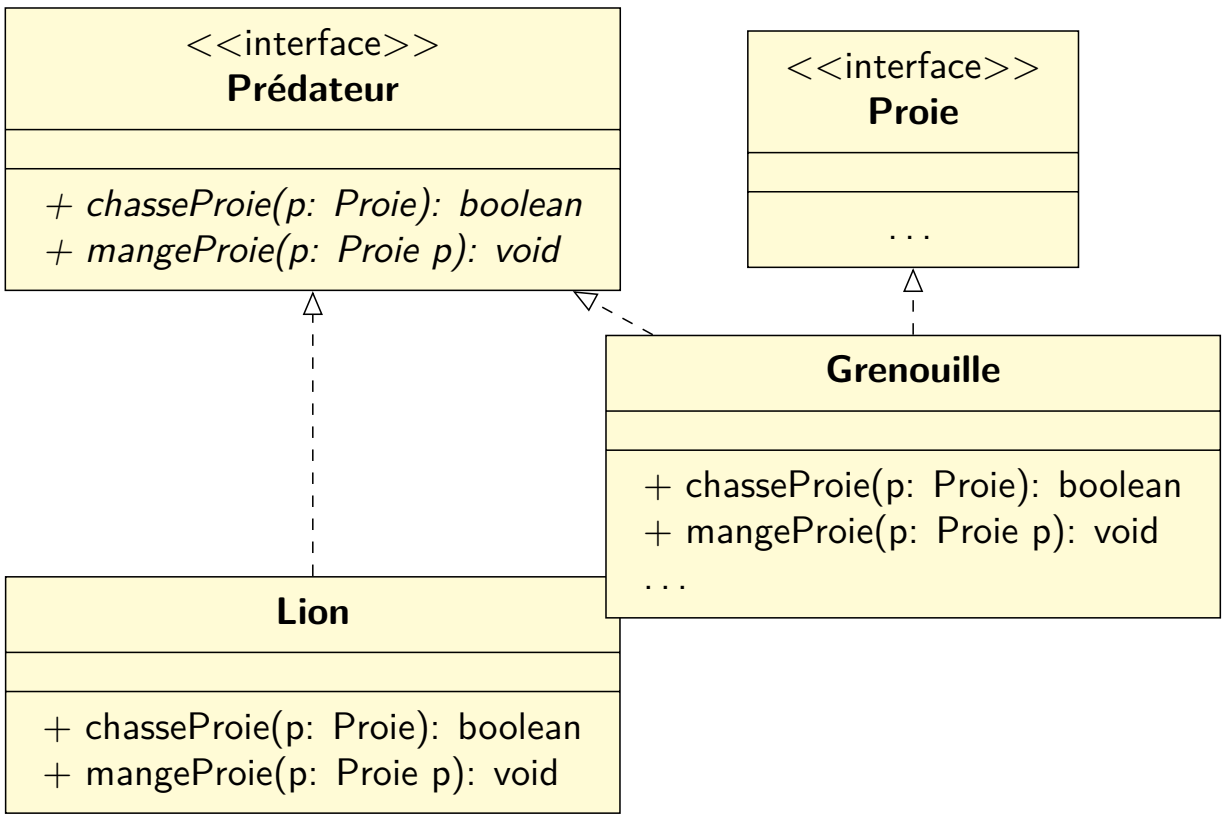
- Une interface peut étendre plusieurs interfaces

```
public interface PrédateurVenimeux
             extends Prédateur, Venimeux {
   ...
}
```

$\Longrightarrow$ L'héritage multiple est possible (contrairement aux classes)

Les différentes interfaces ne doivent pas présenter des méthodes dont les signatures différent uniquement par le type de retour

Diagramme UML et interface



Exemples du standard – Comparable<T>

```
public class Arrays { // extrait de java.util
  ...
  public static void sort(Object[] a) { ... }
}
```

Extrait de la documentation de `sort`

- *Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the `Comparable` interface.*

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

Extrait de la documentation de `compareTo`

- *Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.*

Comparable<T> – utilisation

```
public class Coordonnée
               implements Comparable<Coordonnée> {
  private int x, y;

  public Coordonnée(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public String toString() {
    return "(" + x + ",␣" + y +")";
  }
  public int compareTo(Coordonnée coord) {
    if (x != coord.x)
      return x - coord.x;
    return y - coord.y;
  }
}
```

Comparable<T> – utilisation

```
public class AppliTri {
  public static void main(String[] arg) {
    Coordonnée[] tab = new Coordonnée[4];
    tab[0] = new Coordonnée(1, 5);
    tab[1] = new Coordonnée(5, 5);
    tab[2] = new Coordonnée(2, 5);
    tab[3] = new Coordonnée(1, 3);
    System.out.println(Arrays.toString(tab));
    Arrays.sort(tab);
    System.out.println(Arrays.toString(tab));
  }
}
```

```
[(1, 5), (5, 5), (2, 5), (1, 3)]
[(1, 3), (1, 5), (2, 5), (5, 5)]
```


Exemples du standard – Interface en tant que foncteur

```
public class Arrays { // extrait de java.util
  ...
  public static <T> void sort(T[] a,
                Comparator<? super T> c) { ... }
}
```

Extrait de la documentation de `sort`

- *Sorts the specified array of objects according to the order induced by the specified comparator.*

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

Extrait de la documentation de `compare`

- *Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.*

Comparator<T> – utilisation

```
public class Comp implements Comparator<Coordonnée> {
  public int compare(Coordonnée c1, Coordonnée c2) {
    return -c1.compareTo(c2);
  }
}
```

```
public class AppliTri {
  public static void main(String[] arg) {
    Coordonnée[] tab = new Coordonnée[4];
    tab[0] = new Coordonnée(1, 5);
    ...
    System.out.println(Arrays.toString(tab));
    Arrays.sort(tab, new Comp());
    System.out.println(Arrays.toString(tab));
  }
}
```

```
[(1, 5), (5, 5), (2, 5), (1, 3)]
[(5, 5), (2, 5), (1, 5), (1, 3)]
```


Exemples du standard – les collections

Dans `java.util`

Interface	Implémentations (classes)
`Collection<E>`	*Interface racine de la hiérarchie des collections*
`List<E>`	`ArrayList`, `LinkedList`, `Stack`, `Vector`, ...
`Queue<E>`	`ArrayDeque`, `LinkedList`, `PriorityQueue`, ...
`Deque<E>`	`ArrayDeque`, `LinkedList`, ...
`Set<E>`	`EnumSet`, `HashSet`, `TreeSet`, ...
`Map<K,V>`	`EnumMap`, `HashMap`, `Hashtable`, `TreeMap`, ...

Exemples du standard – Collection<E>

```
public interface Collection<E> {
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  void clear();
  boolean contains(Object o);
  boolean containsAll(Collection<?> c);
  boolean equals(Object o);
  int hashCode();
  boolean isEmpty();
  Iterator<E> iterator();
  boolean remove(Object o);
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
  int size();
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```


Interface et conception

- Développement en équipe
- Définir en commun et au plus tôt des interfaces
- Certains vont les concrétiser, d'autres vont s'en servir (code polymorphe)
- Ceux qui s'en servent ne pourront faire aucune hypothèse d'implémentation
- Problème de la construction d'objet → les *fabriques*

Comparatif Java et C++

D'un point de vue général

- Le C++ autorise la surcharge de méthode
- Le C++ autorise la surcharge d'opérateur

- Le C++ autorise l'héritage et le sous-typage (et donc le polymorphisme)
- Le C++ autorise la spécialisation de méthode
- Le C++ autorise les méthodes abstraites
- Le C++ ne propose pas de notion d'interface mais elle peut être simulée par des classes totalement abstraites

- Le C++ autorise l'héritage multiple

D'un point de vue syntaxique – Spécification d'une classe

```
// personne.h
#include <ostream>

class Personne {
private:
  std::string prenom, nom;
public:
  Personne(const std::string& prenom = "",
           const std::string& nom = "");
  void afficher(std::ostream& os) const;
  const std::string& getPrenom() const;
  const std::string& getNom() const;
};

std::ostream& operator<< (std::ostream& os,
                          const Personne& p);
```


Implémentation

```
// personne.cpp
#include "personne.h"
Personne::Personne(const std::string& p,
    const std::string& n) : prenom(p), nom(n) {
}
void Personne::afficher(std::ostream& os) const {
  os << prenom << "␣" << nom;
}
const std::string& Personne::getPrenom() const {
  return prenom;
}
const std::string& Personne::getNom() const {
  return nom;
}
std::ostream& operator<< (std::ostream& os,
                          const Personne& p) {
  p.afficher(os);
  return os;
}
```

Utilisation

```
// main.cpp
#include <iostream>
#include "personne.h"

int main() {
  Personne bluesMan("Paul", "Personne");
  std::cout << "Quel␣guitariste␣que␣"
            << bluesMan << std::endl;
}
```

```
Quel guitariste que Paul Personne
```


Héritage en C++

```
// etudiant.h
#include "personne.h"

class Etudiant : public Personne {
public:
  Etudiant(const std::string& prenom = "",
           const std::string& nom = "",
           int annee = 1);
  int getAnnee() const;
  void afficher(std::ostream&) const;
private:
  int annee;
};
```

Implémentation

```cpp
// etudiant.cpp
#include "etudiant.h"

Etudiant::Etudiant(const std::string& p,
    const std::string& n, int a) : Personne(p, n) {
  annee = a;
}

int Etudiant::getAnnee() const {
  return annee;
}

void Etudiant::afficher(std::ostream& os) const {
  Personne::afficher(os);
  os << "␣" << getAnnee();
}
```


Utilisation (polymorphe)

```cpp
// main.cpp
#include <iostream>
#include "etudiant.h"

void affichePrenom(const Personne& p) {
  std::cout << p.getPrenom() << std::endl;
}
int main() {
  Personne bluesMan("Paul", "Personne");
  Etudiant genie("Albert", "Einstein", 1);

  affichePrenom(bluesMan);
  affichePrenom(genie);
}
```

```
Paul
Albert
```

Autre cas d'utilisation

```
// main.cpp
#include <iostream>
#include "etudiant.h"

int main() {
  Etudiant genie("Albert", "Einstein", 1);

  std::cout << "Quel étudiant que ";
  genie.afficher(std::cout);
  std::cout << std::endl;

  std::cout << "Quel étudiant que "
            << genie << std::endl;
}
```

```
Quel étudiant que Albert Einstein 1
Quel étudiant que Albert Einstein
```


Spécialisation de méthode

- En C++, par défaut, les méthodes d'une classe ne sont pas spécialisables dans ses sous-classes
- Pour être spécialisable, une méthode d'une classe C++ doit être déclarée `virtual`

- En Java, par défaut, les méthodes d'une classe sont spécialisables dans ses sous-classes
- Pour ne pas être spécialisable, une méthode d'une classe Java doit être déclarée `final`
- Une classe Java ne peut pas être étendue si elle est déclarée `final` (c'est le cas des classes `String`, `Integer`, `Long`, etc.)

La classe Personne corrigée

```
class Personne {
private:
  std::string prenom, nom;
public:
  Personne(const std::string& prenom = "",
           const std::string& nom = "");
  virtual void afficher(std::ostream& os) const;
  const std::string& getPrenom() const;
  const std::string& getNom() const;
};

std::ostream& operator<< (std::ostream& os,
                          const Personne& p);
```

```
Quel étudiant que Albert Einstein 1
Quel étudiant que Albert Einstein 1
```


Méthodes abstraites (*virtuelle pure*)

```
// travailleur.h
#include "personne.h"

class Travailleur : public Personne {
public:
  Travailleur(const std::string& prenom = "",
              const std::string& nom = "");
  virtual void travailler() = 0;
};

class Enseignant : public Travailleur {
public:
  Enseignant(const std::string& prenom = "",
             const std::string& nom = "");
  void travailler();
};
```

Implémentation

```cpp
// travailleur.cpp
#include <iostream>
#include "travailleur.h"

Travailleur::Travailleur(const std::string& p,
          const std::string& n) : Personne(p, n) {
}

Enseignant::Enseignant(const std::string& p,
           const std::string& n) : Travailleur(p, n) {
}

void Enseignant::travailler() {
  std::cout << *this << " nous l'affirme : ";
  std::cout << "Vive la POO !" << std::endl;
}
```


Utilisation (polymorphe)

```cpp
// main.cpp
#include "travailleur.h"

int main() {
  Enseignant dg("Daniel", "Guillaume");
  dg.travailler();

  Travailleur& bosseur = dg;
  bosseur.travailler();
}
```

```
Daniel Guillaume nous l'affirme :  Vive la POO !
Daniel Guillaume nous l'affirme :  Vive la POO !
```