Cours JAVA : Le polymorphisme en Java.

Version 3.01

Julien Sopena[1]

[1]julien.sopena@lip6.fr
Équipe REGAL - INRIA Rocquencourt
LIP6 - Université Pierre et Marie Curie

Licence professionnelle DANT - 2012/2013

Grandes lignes du cours

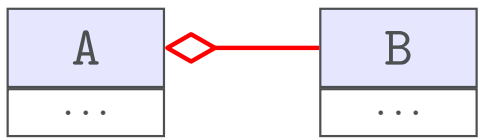
Outline

Outline

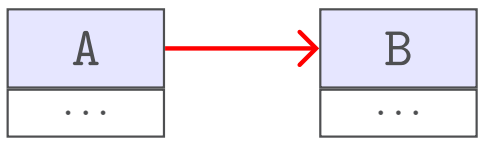
Rappel – principes de la POO

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

Définition

Le terme héritage désigne le principe selon lequel une classe peut hériter des caractéristiques (attributs et méthodes) d'autres classes.

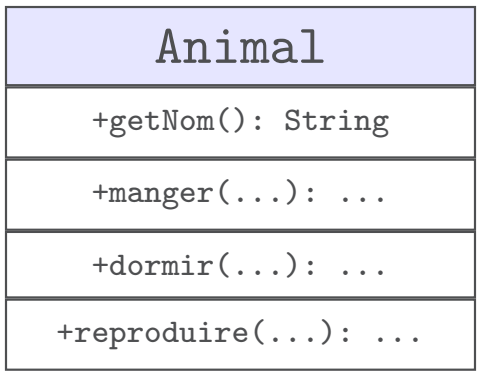
Soient deux classes A et B

- 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)



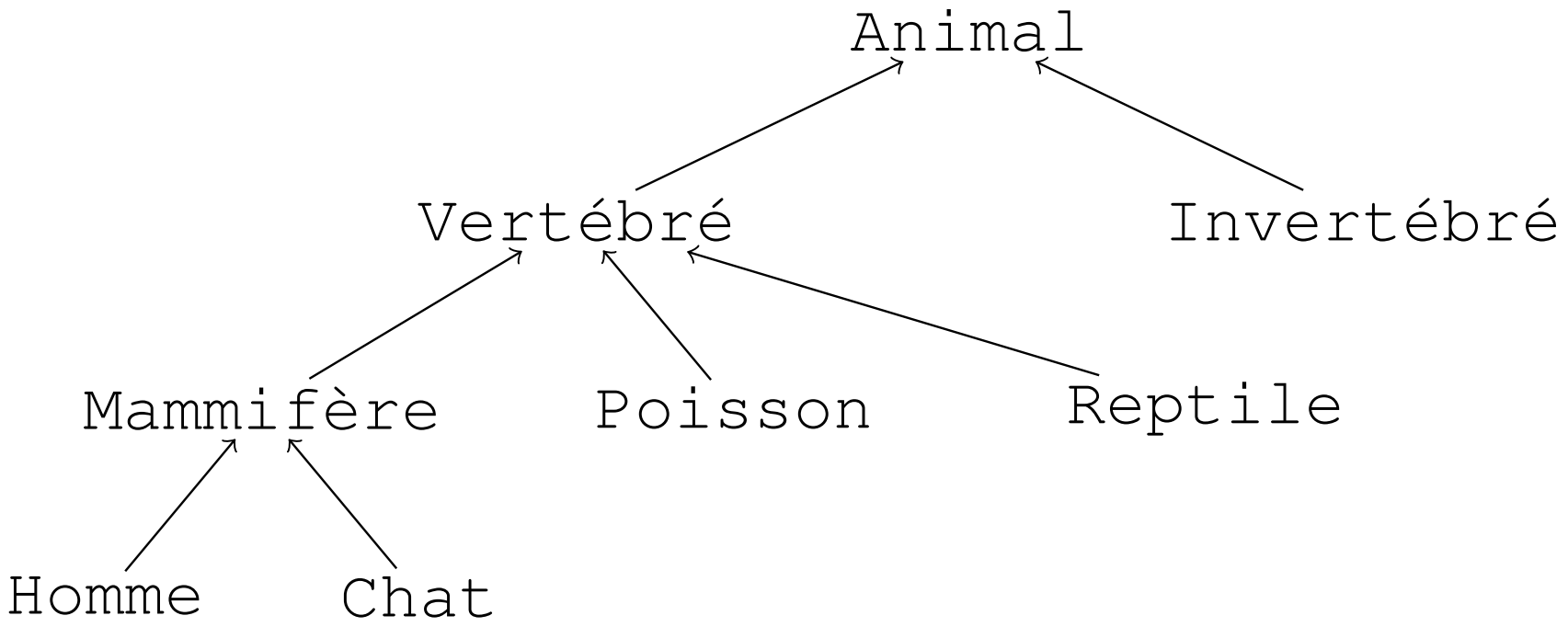
Problèmes pour implémenter cette spécification :

- Elle est trop générale (exemple : manger( ) est très différent suivant 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 factorisations perdues

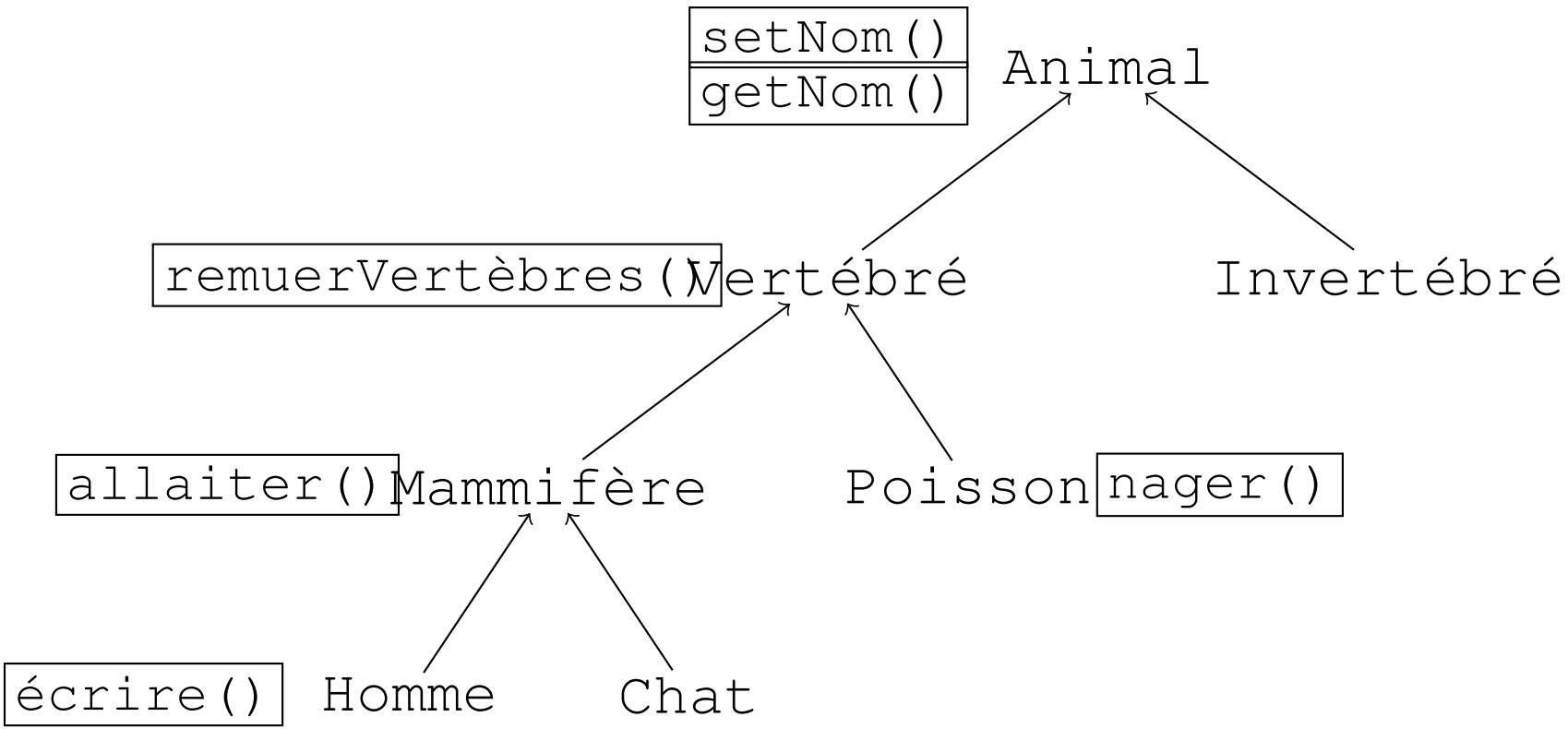
Classification hiérarchisée



- 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`

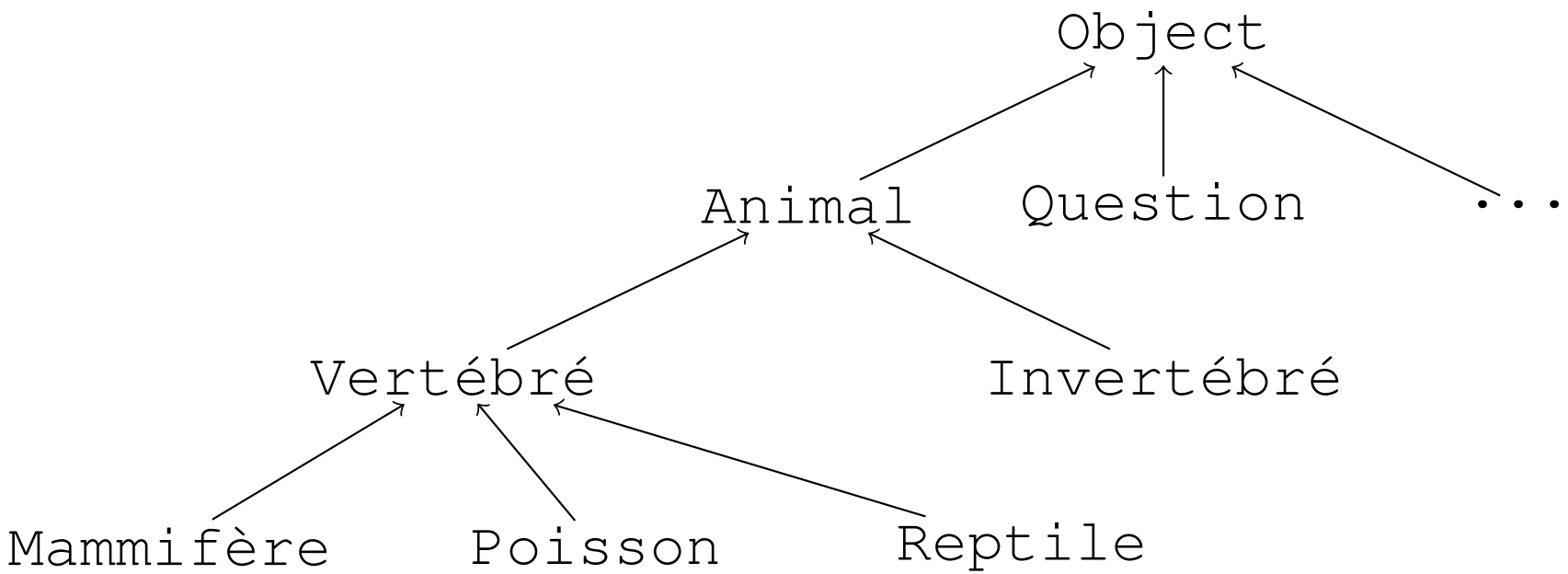
Héritage

- 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 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



Outline

Syntaxe

Pour indiquer qu'une classe hérite d'une autre classe, on utilise le mot-clé **extends**.

```
public class Appartement extends Logement
```

Important

La classe `Appartement` ainsi définie possède toutes les caractéristiques de la classe `Logement` (*i.e.*, ses éléments privés et publics).

Spécialisation

```
public class Logement {
  final public double surface;
  public double prix;
  public String proprietaire;
  private boolean vendu;
}
```

```
public class Appartement extends Logement {
  final public int etage;
  private boolean cave;
}
```

Les attributs de la classe `Appartement` sont : `surface`, `prix`, `proprietaire`, `vendu`, `etage` et `cave`.

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`
- Puisqu'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

À retenir : la classe `Object`

Important

En Java, toutes les classes héritent de la classe `Object`.

La déclaration

```
public class A { ... }
```

doit donc être comprise comme

```
public class A extends Object { ... }
```

même si on ne l'écrit pas ainsi.

À retenir : le mot-clé **super**

Á retenir

Java fournit la référence **super** qui *désigne* pour chaque classe, sa classe mère.

Héritage d'attributs

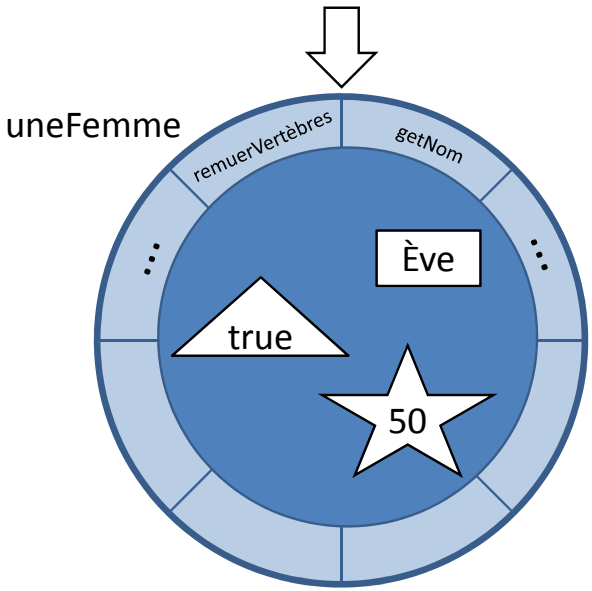
```
public class Animal {
  // attributs propres à tous les animaux
  private String nom; ...
  public String getNom(){ return nom; }
  public void setNom(String n) { nom = n; }
  ...
}
public class Vertébré extends Animal {
  // attributs propres aux Vertébrés
  private int nbVertèbres; ...
  public 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(50);
    this.setNom(unNom);
    this.droitier = droitier;
  }
  ...
}
```

Héritage d'attributs – fin

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




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

Outline

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 (et attribut) détermine sa visibilité :

	private	défaut	protected	public
la classe elle-même	OUI	OUI	OUI	OUI
une sous-classe, paquetage $=$	NON	OUI	OUI	OUI
pas une sous-classe, paquetage $=$	NON	OUI	OUI	OUI
une sous-classe, paquetage $\neq$	NON	NON	OUI	OUI
pas une sous-classe, paquetage $\neq$	NON	NON	NON	OUI

Rappel

Les attributs doivent toujours être déclarés **private**

Outline

Instantiation des attributs hérités

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

Problèmes

- Obligation d'introduire des modificateurs telles que `Vertébré.setNbVertèbres()`
- Redondance – la super-classe `Mammifère` doit avoir un constructeur

Héritage et construction – Correction

Pour initialiser les attributs hérités, le constructeur d'une classe peut invoquer un des constructeurs de la classe mère à l'aide du mot-clé **super**.

```
public class Homme extends Mammifère {
  private static final int NB_VERTEBRES = 50;
  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);
  }
}
```

Règles d'utilisation d'un constructeur de la classe mère.

Règles

1. L'appel d'un constructeur de la classe mère doit être la première instruction du constructeur de la classe fille.
2. Il n'est pas possible d'utiliser à la fois un autre constructeur de la classe et un constructeur de sa classe mère dans la définition d'un de ses constructeurs.

```
public class A {
  public A(int x) {
    super();
    this();
  }
}
```

INTERDIT

Constructeur implicite

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

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

Attention : Dans ce cas, le constructeur sans argument doit être défini dans la super-classe

Constructeur implicite : exemple

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

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

Le ligne de code suivante :

```
  B c = new B ();
```

produira l'affichage suivant :

```
java MonProg
  A
  B
```

Constructeur implicite : erreur fréquente

Etape 1 :
Jusqu'ici tout va bien.

```
public class A {
  public int x;
}

public class  B extends A {
  public int y;
  public B (int x,int y) {
    this.x = x;
    this.y = y;
  }
}
```

Etape 2 :
Puis, on ajoute un constructeur.

```
public class A {
  public int x;
  public A (int x) {
   this.x =x;
  }
}

public class  B extends A {
  public int y;
  public B (int x,int y) {
    this.x = x;
    this.y = y;
  }
}
```

Constructeur implicite : erreur fréquente

```
javac B.java
    B.java:3: cannot find symbol
    symbol  : constructor A()
    location: class A
    public B ( int x , int y ) {
                              ^
    1 error
    }
```

```
public class  B extends A {
  public int y;
  public B (int x,int y) {
     super(); \\ Ajout du compilateur
     this.x = x;
     this.y = y;
  }
}
```

Constructeur implicite : Solutions

Solution 1 :
Ajout d'un constructeur vide.

```
public class A {
  public int x;
  public A () {}
  public A (int x) {
    this.x = x;
  }
}

public class  B extends A {
  public int y;
  public B (int x,int y) {
    this.x = x;
    this.y = y;
  }
}
```

Solution 2 :
Appel explicite au **super(int)**.

```
public class A {
  public int x;
  public A (int x) {
    this.x = x;
  }
}

public class B extends A {
  public int y;
  public B (int x,int y) {
    super(x);
    this.y = y;
  }
}
```

Constructeur d'une classe dérivée

Un constructeur d'une classe dérivée commence toujours :

1. soit par l'appel explicite d'un autre constructeur de la classe.
2. soit par l'appel explicite ou implicite d'un constructeur de la classe mère.

Corollaire

Comme tout objet dérive (directement ou indirectement) de la classe **Object** : **Tout constructeur commence par exécuter le constructeur de l'Object**

Outline

Définition de la redéfinition.

Définition

*On appelle **redéfinition** (en anglais « overriding ») d'une méthode, la possibilité de définir le comportement d'une méthode selon le type d'objet l'invoquant, i.e., de donner une nouvelle implémentation à une méthode héritée sans changer sa signature.*

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

La signature d'une méthode Java

```
public class Homme extends Mammifère {
  ...
  public void dessiner () {
    System.out.println("Gribouillis");
  }
}

public class Leonard extends Homme {
  ...
  public void dessiner () {
    System.out.println("Joconde");
  }
}
```

Overloading vs Overriding

ATTENTION

Il ne faut pas confondre la **redéfinition** (en anglais « *overriding* ») et la **surcharge** (en anglais « *overloading* ») qui a été étudiée dans le cours d'introduction et qui correspond à la possibilité de définir des comportements différents pour la même méthode selon les arguments passés en paramètres.

Surcharge d'une méthode redéfinie

On peut aussi surcharger une méthode redéfinie.

```
public class Logement {
  public void vendre(String acquereur) {
    this.proprietaire = acquereur;
    this.vendu = true;
  }
}

public class Appartement extends Logement {
  public void vendre(String acquereur) {
    this(acquereur);
    this.cave = false:
  }
  public void vendre(String acquereur,boolean cave) {
    this(acquereur);
    this.cave = cave:
  }
}
```

Spécialiser une méthode héritée

La référence **super** permet de redéfinir une méthode `f` héritée en réutilisant sa définition dans la classe mère. On dit alors qu'on spécialise la méthode héritée `f`.

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

public class B extends A {
  public void f() {
    ...
    super.f()
    ...
  }
}
```

Spécialiser une méthode héritée : exemple

```
public class Point {
  private int x,y;
  ...
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}

public class PointCouleur extends Point {
  private byte couleur;
  ...
  public String toString() {
    return super.toString() +
           "\nCouleur : " + this.couleur ;
  }
}
```

Limite pour désigner une méthode redéfinie

Attention

On ne peut remonter plus haut que la classe mère pour récupérer une méthode redéfinie :

- pas de cast `(ClasseAncetre)m()`
- pas de **super**.**super**`.m()`

Annotation pour la redéfinition

Depuis Java 5, on peut annoter par **@Override** les redéfinitions de méthodes. Ceci est très utile pour repérer des fautes de frappe dans le nom de la méthode : le compilateur envoie un message d'erreur si la méthode ne redéfinit aucune méthode.

```
public class Point {
  private int x,y;
  ...
  @Override
  public String tostring() {
    return "(" + x + "," + y + ")";
  }
}
```

```
Point.java:3: method does not override or implement a method from a
supertype
      @Override
      ^
```

Outline

Les fondements de la covariance

Rappels

1. La **redéfinition** d'une méthode c'est le fait de donner une nouvelle implémentation en conservant la signature.
2. La **signature** d'une méthode est composée de son nom et des types de ses arguments.

Définition

*On appelle **covariance** le fait de modifier le type de retour d'une méthode lors de sa redéfinition. En Java, elle a été introduite dans la version : Java 1.5.*

Outline

classes finales

Lors de la conception d'une classe, le concepteur peut empêcher que d'autres classes héritent d'elle (classe finale).

```
final public class A { }
```

Remarque(s)

La classe `String` est une classe finale.

Méthodes finales

On peut empêcher la redéfinition d'une méthode d'instance d'une classe dans une de ses sous-classes en la déclarant **final**.

```
public class A {
  final public void f() {}
}

public class B extends A {
  // on ne peut pas redéfinir f() !
 }
```

Outline

Outline

Polymorphisme : définition

Définition

*Le **polymorphisme** peut être vu comme la capacité de choisir dynamiquement la méthode qui correspond au type réel de l'objet.*

Principe POO : Polymorphisme

Si la classe `B` hérite de la classe `A`

- 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`)

Le polymorphisme consiste à exploiter cela en fournissant un `B` dans les expressions "qui attendent" un `A`.

```
// 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 ;
```

Choix de la méthode : liaison dynamique

```
public class Point {
  private int x,y;
  ...
  public String toString() {
    return "(" + x + "," + y + ")";
  }
}

public class PointCouleur extends Point {
  private byte couleur;
  ...
  public String toString() {
    return super.toString() +
           "\nCouleur : " + this.couleur ;
  }
}
```

Choix de la méthode : liaison dynamique (suite)

```
public class TraceurDeLigne {

  public static void main(String[] args) {

    Point[] tab = new Point[3];

    tab[0] = new PointCouleur(2,3,bleu);
    tab[1] = new PointCouleur(2,3,vert);
    tab[2] = new PointCouleur(2,3,rouge);

    System.out.println("Ma ligne : " ) ;
    for (Point p : tab)
      System.out.println(" - " + p.toString());
    }
  }
}
```

Choix de la méthode : liaison dynamique (suite)

Si `p` référence une `PointCouleur`, quelle méthode `toString()` va être invoquée ?

1. La méthode `toString()` est définie dans `Point`
2. Mais elle est *spécialisée* dans `PointCouleur`

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

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

Lorsqu'une méthode est spécialisée dans une sous-classe, sa visibilité peut être augmentée (ex. **protected** $\rightarrow$ **public**) mais elle ne peut pas être réduite (ex. **public** $\not\rightarrow$ **private**)

Écrire du code polymorphe : le problème

On veut coder une classe Zoo qui aura plusieurs cages permettant d'acceuillir différents types d'animaux. Il faut donc implémenter une classe Cage permettant de contenir tous ces annimaux.

```
public class Zoo {
  public static void main (String[] args) {
    Cage uneCage1 = new Cage(...);
    Cage uneCage2 = new Cage(...);

    // On ajoute un lion
    Lion unLion = new Lion(...);
    uneCage1.accueillir(unLion);

    // On ajoute un singe
    Singe unSinge = new Singe (...);
    uneCage2.accueillir(unSinge);
  }
}
```

Écrire du code polymorphe : la mauvaise solution

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

Ici, la surcharge est une très mauvaise solution

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

Écrire du code polymorphe : la bonne solution

La bonne solution consiste à utiliser le **polymorphisme**, en implémentant une méthode `acceuillir` générique pour tout les animaux.

Son paramètre étant de type `Animal` on pourra l'appeler avec une référence de `Lion` ou de `Singe`.

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

Outline

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

```
public class Zoo {
  public static void main (String[] args) {
    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;
    ...
  }
}
```

Très mauvaise solution

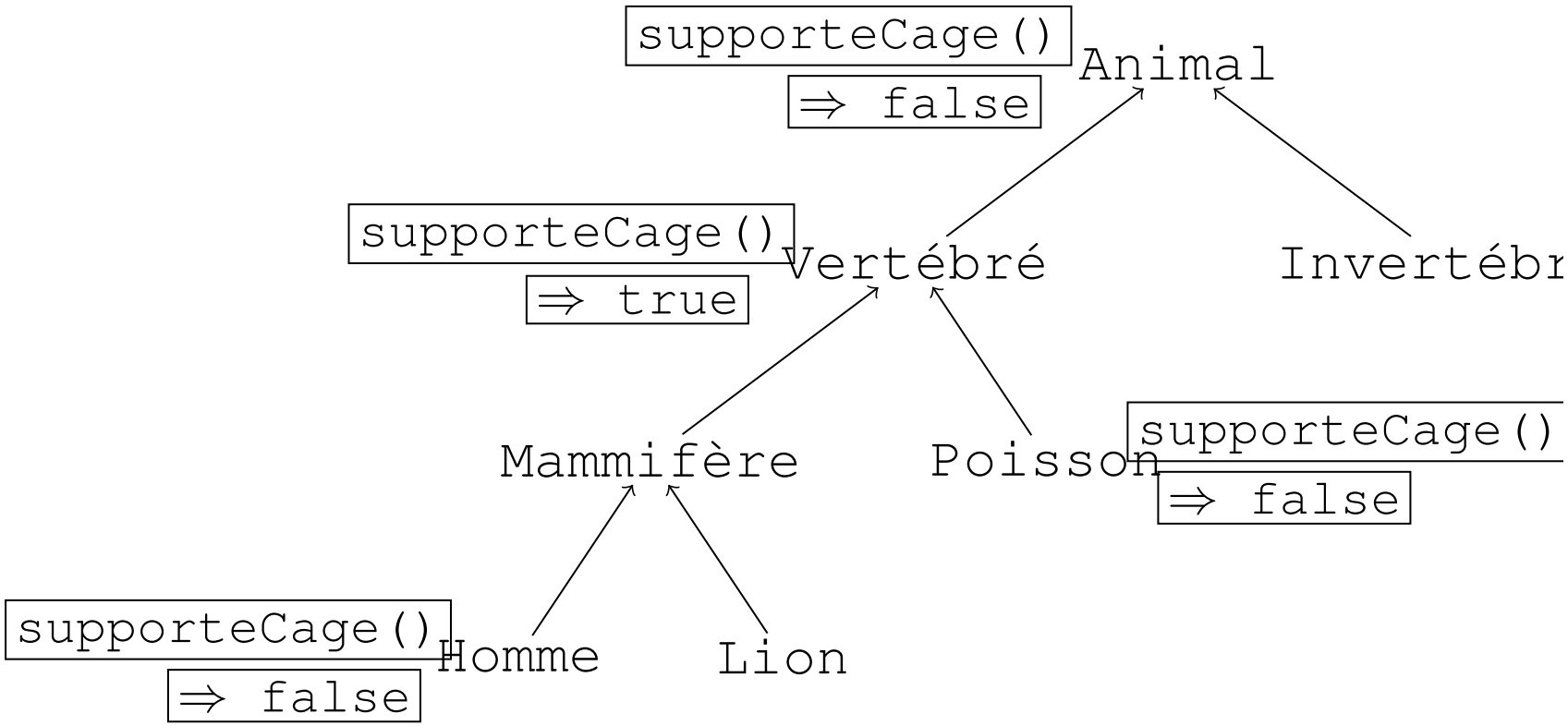
- Des connaissances propres à la classe `Homme` sont dans la classe `Cage`
- si une nouvelle espèce animale refuse d'aller en cage, il faudra modifier le code de la classe `Cage`

Mise en place d'un protocole

```
public class Cage {
  ...
  public void accueillir (Animal a) {
    if (!a.supporteCage()) {
      System.out.print(a.getNom());
      System.out.println(" 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; }
}

public class Zoo {
  public static void main (String[] args) {
    Cage uneCage1 = new Cage(...);
    uneCage1.accueillir(new Lion(...));
    // Génère un appel à Vertébré.supporteCage();
  }
}
```

Outline

Retour sur les protocoles standards

```
public class Zoo {
  public static void main (String[] args) {
    Cage[10] lesCages ;
    Lion unLion = new Lion(...);
    lesCages[0].accueillir(unLion);
    Singe unSinge = new Singe (...);
    lesCages[1].accueillir(unSinge);

    for (Cage c : lesCages) {
      System.out.println(c.contenu());
      if (unLion.equals(c.contenu())
          System.out.println("J'ai trouve le lion");
    }
  }
}
```

- La classe `Lion` ne dispose pas de méthode `equals()`, pourquoi n'y a-t-il pas d'erreur de compilation ?
- Où est l'appel la méthode `toString()` de `Animal` ?

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 `Animal` 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 Animal  {
  ...

  @Override
  public String toString() {
    return this.getNom() ;
  }

}
```

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 Extrait de la documentation de `java.util.Arrays` :

```
public static boolean equals(Object[] a, Object[] a2)
```

Returns true if the two specified arrays of `Objects` are equal to one another. The two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal.
Two objects `e1` and `e2` are considered equal if

```
(e1==null ? e2==null : e1.equals(e2))
```

In other words, the two arrays are equal if they contain the same elements in the same order. Also, two array references are considered equal if both are null.

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);
  }
}
```

Outline

Upcasting : classe fille → classe mère

Définition

*On appelle **surclassement** ou **upcasting** le fait d'enregistrer une référence d'une instance d'une classe `B` héritant d'une classe `A` dans une variable de type `A`. En java, cette opération est implicite et constitue la base du polymorphisme.*

```
public class A { ... }

public class B extends A { ... }

A a = new B() // C'est de l'upcasting (surclassement).
```

On dit que a1 est une référence **surclassée** (elle est du type `A` et contient l'adresse d'une instance d'une sous-classe de `A`).

Downcasting : classe mère → classe fille

Définition

*On appelle **déclassement** ou **downcasting** le fait de convertir une référence « surclassée » pour « libérer » certaines fonctionnalités cachées par le surclassement. En java, cette conversion n'est pas implicite, elle doit être forcée par l'oppérateur de **cast** :* ***(<nomClasse>)***.

```
public class A { ... }

public class B extends A { ... }

A a = new B(); // surclassement, upcasting
B b = (B) a; // downcasting
```

Pour que la conversion fonctionne, il faut qu'à l'exécution le type réel de la référence à convertir soit `B` ou une des sous-classe de B !

Downcasting : mise en garde !

Attention

Le downcasting ne permet pas de convertir une instance d'une classe donnée en une instance d'une sous-classe !

```
class A { A() {}}

class B extends A { B() {}}

A a = new A();
B b = (B) a ;
```

Ça compile, mais ça plantera à l'exécution :

```
Exception in thread "main" java.lang.ClassCastException:
      A cannot be cast to B
```

Downcasting : utilisation de instanceof

On peut utiliser le mot-clé **instanceof** pour savoir si une référence d'une classe donnée ou d'une de ces sous-classes.

Si `a` est une référence de type `B` alors l'expression `a` **instanceof** `A` renvoie **true** si `B` est une sous-classe de `A` et **false** sinon.

```
a = new B() ;
if (a instanceof A) {
  System.out.print   ("a référence un A,") ;
  System.out.println ("ou une sous classe de A.");
}
```

Remarque(s)

Attention, l'utilisation d'**instanceof** est souvent la marque d'un défaut de conception et va à l'encontre du polymorphisme.

Exemple : classe rendant un service générique.

```
public class Garage {
  Vehicule[] tab;
  int nbVehicules;

  Garage (int taille) {
    tab = new Vehicule[taille];
    nbVehicules = 0;
  }

  void entrerVehicule (Vehicule v) {
    if (nbVehicules<tab.length)
      tab[nbVehicules++] = v ;
  }

  Vehicule sortirVehicule () {
    if (nbVehicules>0)
      return tab[--nbVehicules];
    else
      return null;
  }

}
```

Exemple : polymorphisme d'héritage

```
public class Vehicule {
  ...
}

public class Moto extends Vehicule {
  ...
}

public class Voiture extends Vehicule {
  int temperature ;
  ...
  void augmenterChauffage(int deg) {
    temperature += deg;
  }
}
```

Exemple : le downcast c'est la fin du polymorphisme

```
public class Main {
  public static void main (String[] args) {
    Individu moi = new Individu(...);
    Garage monGarage = new Garage(2) ;
    // Le polymorphisme permet de garer des
    // véhicules sans s'occuper de leurs types.
    monGarage.entrerVehicule (new Moto());
    monGarage.entrerVehicule (new Voiture());
    ...
    Vehicule unVehicule = monGarage.sortirVehicule();
    if (moi.avoirFroid()) {
      // Si on ne partage pas le garage, on sait
      // qu'on vient de sortir une voiture
      ((Voiture)unVehicule).augmenterChauffage(10);
    }
  }
}
```

Outline

Ce qu'on ne peut pas faire et ne pas faire

Pour permettre le polymorphisme :

$\Longrightarrow$ une sous classe doit pouvoir se faire passer pour une sur classeur ;

$\Longrightarrow$ une fille doit pouvoir le faire tout ce que fait sa mère ;

On ne peut rien supprimer :

- On ne peut pas supprimer un membre ;
- Changer le type d'un attribut ;
- On ne peut pas réduire la visibilité d'un membre ;
- La covariance doit maintenir la compatibilité (voir suivants).

Par contre on peut toujours ajouter/modifier des choses :

- On peut ajouter des membres ;
- On peut redéfinir des méthodes ;
- On peut augmenter la visibilité d'un membre.

Problème de la covariance

Dans cet exemple de **covariance**, peut-on choisir `Y` librement ?

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

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

Problème de la covariance

Dans cet exemple de **covariance**, peut-on choisir `Y` librement ?

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

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

Limite de la covariance

```
public class Main {
  static void g (A a) {
    ...
    X x = a.f(3);
    ...
  }
  public static void main (String[] args) {
    A a = new A();
    g(a);
    a = new B();
    // Quelle version f sera executée dans cet appel ?
    g(a);
  }
}
```

C'est la verion de `B` qui sera executée :

$\Longrightarrow$ La référence de `Y` retournée doit être "compatible" avec `X`

$\Longrightarrow$ La classe `Y` doit être un descendant de de la classe `X`

Limite de la covariance

Règle de covariance

Lorsque l'on fait de la redéfinition avec **covariance**, le nouveau type de retour doit toujours être un sous-type du type de retour original.

Outline

Outline

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

Définition

*Un classe contenant au moins une méthode abstraite est appelée une **classe abstraite** et cela doit être explicitement précisé dans la déclaration avec : **abstract class***

Remarque(s)

- Une classe abstraite peut contenir des méthodes concrètes.
- Une classe peut être déclarée abstraite sans contenir de méthode abstraite.

Classe abstraite : instanciation et spécialisation.

- Une classe abstraite constitue un type à part entière :

```
Animal unAnimal; // OK
```

- Une classe abstraite ne peut pas être instanciée, en effet son comportement n'est pas complètement défini :

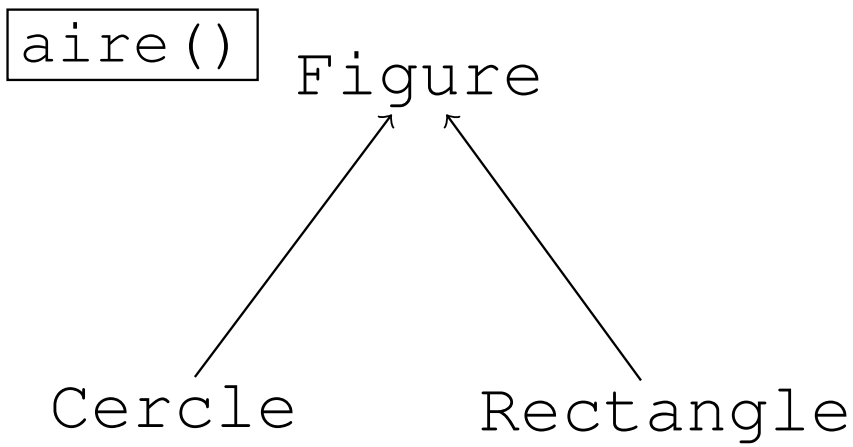
```
Animal unAnimal = new Animal (...); // ERREUR
```

- Une sous-classe d'une classe abstraite peut
  - **implémenter** toutes les méthodes abstraites de sa super-classe. Elle pourra alors être déclarée comme concrète et donc instanciée.
  - **ne pas implémenter** toutes ces méthodes abstraite. Elle reste alors nécessairement abstraite (abstract class) et ne pourra être instanciée.
  - **ajouter** d'autre(s) méthode(s) abstraite(s). Elle reste alors nécessairement abstraite (abstract class) et ne pourra être instanciée.

Outline

Exemple : besoin

Considérons la hiérarchie de classes suivantes :



Cahier de charge : on veut que toutes les classes disposent de la méthode `aire()` retournant l'aire de la figure géométrique définie par la classe.

Exemple : déclaration de la classe abstraite.

```
public abstract class Figure {
  private String nom;

  // Une classe abstraite ne peut pas être instanciée,
  // mais elle peut avoir un constructeur :
  public Figure(String nom) {
    this.nom=nom;
  }

  // Voilà la méthode abstraite à compléter :
  public abstract double aire();

  // Toutes les méthodes ne sont pas abstraites :
  public String quiSuisJe() {
    System.out.println("Je␣suis␣un␣" + this.nom);
  }

}
```

Exemple : première implémentation.

```
public class Cercle extends Figure {
  private double rayon;

  public Cercle(double rayon) {
    super("cercle");
    this.rayon = rayon;
  }

  public double aire() {
    return Double.PI * this.rayon * this.rayon;
  }

}
```

Exemple : deuxième implémentation.

```
public class Rectangle extends Figure {
  private double largeur;
  private double longueur;

  public Rectangle(double largeur,double longueur) {
    super("rectangle");
    this.largeur = largeur;
    this.longueur = longueur;
  }

  public double aire(){
    return this.largeur * this.longueur;
  }

}
```

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();
...
```

Outline

Outline

Préambule

Problèmes à résoudre :

- assurer qu'un ensemble de classes offre un service minimum commun.
- faire du polymorphisme avec des objets dont les classes n'appartiennent pas à la même hiérarchie d'héritage.
- utilisation d'objets sans connaître leur type réel.

Solution :

- la définition d'un type complètement abstrait nommé **interface** (notion de **contrat**).

Définition

Quand **toutes les méthodes** d'une classe sont **abstraites** et qu'il n'y a **aucun attribut**, on aboutit à la notion d'interface.

Définition

- *Une **interface** est un prototype de classe. Elle définit la signature des méthodes qui doivent être implémentées dans les classes construites à partir de ce prototype.*
- *Une **interface** est une "classe" purement abstraite dont toutes les méthodes sont abstraites et publiques. Les mots-clés **abstract** et **public** sont optionnels.*

Outline

Déclaration : syntaxe

La définition d'une interface se présente comme celle d'une classe, en utilisant le mot-clé **interface** à la place de **class**.

```
public interface Comparable {
  public abstract boolean plusGrand(Object o);
}

public interface Tracable {
  void dessineToi();
  void deplaceToi(int x,int y);
}
```

Remarque(s)

Dans les déclarations des méthodes de la classe `Tracable`, les mots clés **public** et **abstract** sont implicites, mais ce n'est pas recommandé.

Déclaration : règles

Comme les classes abstraites, les interfaces ne sont pas instanciables :

- Une interface ne possède pas d'attribut instance.
- Une interface n'a pas de constructeur.

Le but des interfaces est définir des API :

- toutes leurs méthodes sont **public**. Elles ne définissent pas les mécanismes internes

  public abstract `int f();`

- tout leurs attributs de classe sont des constante, c'est-à-dire définir des attributs déclarées comme **public static final** et ayant une valeur constante d'affectation. Exemple :

  public static final `float PI = 3.14f;`

Implémentation d'une interface

Définition

*On dit qu'une classe **implémente** une interface si elle définit l'ensemble des méthodes abstraites de cette interface. On utilise alors, dans l'entête de la classe, le mot-clé **implements** suivi du nom de l'interface implémetée.*

```
class Personne implements Comparable {
  private String nom;
  private String prenom;
  ...
  public boolean plusGrand(Object o) {
    return this.nom.compareTo(((Personne)o).nom) ||
           this.prenom.compareTo(((Personne)o).prenom) ;
  }
}
```

Implémentation d'une interface

Attention

La classe doit implémenter toutes les méthodes de l'interface, sinon elle doit être déclarée **abstract**.

```
abstract class Cercle implements Tracable {
  private int xCentre, yCentre;
  private int rayon;
  ...
  public void deplaceToi(int x, int y) {
    xCentre += x ;
    yCentre += y ;
  }
}
```

Outline

Utilisation comme type de données

Une interface peut remplacer une classe pour déclarer :

- un attribut
- une variable
- un paramètre
- une valeur de retour

A l'exécution, la donnée correspond à une référence d'un objet dont la classe implémente l'interface.

C'est suivant le type réel de cet objet que l'on choisira le code des méthodes à exécuter.

Exemple d'utilisation comme type de données.

```
public class Main {
  public static void main (String[] args) {
    Comparable var1, var2;
    ...
    if (var1.plusGrand(var2)) {
      System.out.println("Var1 est plus grand que var2");
    } else {
      System.out.println("Var1 est plus petit que var2");
    }
  }
}
```

Ici les variables `var1` et `var2` contiennent des références vers des objets dont les classes :

- peuvent être différentes
- implémentent l'interface `Comparable`

Exemple de polymorphisme dans un tableau

```
public class ListeElements {
  Comparable[] tab;
  int nbElements = 0;
  ...

  public void addElement (Comparable e) {
    tab[nbElements] = e;
    nbElements++ ;
  }

  public boolean croissant() {
    for (int i = 1; i < nbElements ; i++) {
      if (tab[i-1].plusGrand(tab[i]))
        return false;
    }
    return true;
  }
}
```

Outline

Choix entre classe et interface : principe

Une interface peut servir à faire du polymorphisme comme l'héritage, alors comment choisir entre classe et interface ?

1. **Choix dicté par l'existant :** L'héritage n'est plus possible, la classe hérite déjà d'une autre classe. Il ne reste plus que celui l'interface.
2. **Choix à la conception :** On étudie la relation entre `A` et `B` ?
   - Un objet de classe `B` **"EST UN"** `A`
     $\Longrightarrow$ **Héritage** : `B` **extends** `A`.
   - Un objet de classe `B` **"EST CAPABLE DE FAIRE"** `A`
     $\Longrightarrow$ **Interface** : `B` **implements** `A`.

Choix entre classe et interface : exemple

Une grenouille **"EST UN"** amphibien :

```
class Grenouille extends Amphibien {
   ...
}
```

Une grenouille **"EST CAPBLE DE FAIRE"** amphibie :

```
class Grenouille implements Amphibie {
   ...
}
```

Outline

Implémentation de plusieurs interfaces

Une classe peut hériter d'une autre classe et implémenter une interface :

```
class Grenouille extends Batracien implements Amphibie {
    ...
}
```

Une classe peut implémenter plusieurs interfaces :

```
class Grenouille implements SurTerre, SurEau {
   ...
}
```

Extension d'une interface

Lorsqu'une capacité s'exprime comme une extension d'une autre capacité, on peut étendre une interface en utilisant le mot clé **extends** :

```
interface SavoirConduire {
   ...
}

interface SavoirPiloter extends SavoirConduire {
  public abstract void deraper (int degre) ;
}
```

Composition de plusieurs interfaces

Lorsqu'une capacité s'exprime comme un ensemble d'autres capacités, on peut faire de la composition d'interfaces en utilisant le mot clé **extends** :

```
interface Amphibie extends SurTerre, SurEau {
   ...
}
```

Attention

Même si l'on utilise le même mot clé que pour l'héritage, ce n'est pas de l'héritage multiple qui n'existe pas en Java.

Polymorphisme d'interfaces

Comme pour les classes, on peut faire du **polymorphisme d'interfaces**.

```
public class Chauffeur extends Homme implements SavoirConduire {
  ...
}

public class Pilote extends Homme implements SavoirPiloter {
  ...
}

public class Course {
  public static void main (String[] args) {
    SavoirConduire[10] participants;

    Chauffeur julien = new Chauffeur () ;
    Pilote sebastienLoeb = new Pilote () ;

    participants[0] = julien ;
    participants[1] = sebastienLoeb ;
  }
}
```