ABSTRACTION

Classes et méthodes abstraites
Polymorphisme
Sous-typage

Hiérarchie

- attributs et codes **répétés** ➡ factorisation de code possible
- comment concilier la factorisation de code et les différences ?

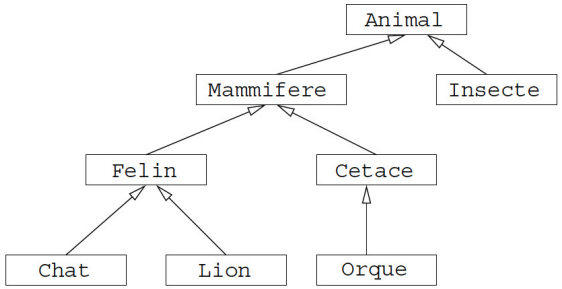
- **Héritage de classe**
- On peut définir une classe **héritant** d'une autre classe.
- la classe héritante
  - Récupère tous les comportements (accessibles) de la classe dont elle hérite,
  - peut modifier certains comportements hérités,
  - peut ajouter de nouveaux comportements qui lui sont propres.
- les instances de la classe héritante sont également du type de la classe héritée : polymorphisme
  - on parle de sous-classe (donc sous-type)
  - une instance de sous-classe est également du type de la classe mère (ou super-classe).

Hiérarchie

on fait souvent référence à l'héritage comme réalisant la relation "est un"

- Par exemple :
  - un `Mammifere` est un `Animal` vertébré
  - un `Insecte` est un `Animal` terrestre invertébré, à 6 pattes, le plus
  - souvent ailé,
  - un `Felin` est un `Mammifere` terrestre carnivore
  - un `Cetace` est un `Mammifere` qui vit dans l'eau
  - un `Chat` est un `Felin` qui miaule
  - un `Lion` est un `Felin` à pelage fauve qui rugit
  - un `Orque` est un `Cetace` carnivore

Hiérarchie



- Par héritage, une instance de `Chat` est aussi un `Felin`, un `Mammifere` et un `Animal`
- l'interface publique définie dans `Felin` fait partie de l'interface publique d'un objet `Chat`.
- idem avec les interfaces publiques de `Animal` et `Mammifere`.

Object

toutes les classes héritent par défaut de la classe `Object`
(soit directement soit via leur superclasse)

Donc
- tout objet peut se faire passer pour un objet de type `Object`
- tout objet peut utiliser les méthodes définies par la classe `Object`
  - exemples : `equals(Object o)`, `toString()`, `hashCode()`

Hiérarchie

Factorisation du comportement

Les comportements (accessibles) définis dans une classe sont directement disponibles pour les instances des classes qui en héritent (même indirectement).

```
public class Mammifere extends Animal {
   public String organeDeRespiration() {
      return "poumons";
   }
}
public class Felin extends Mammifere { ... }
Felin felix = new Felin();
String s = felix.organeDeRespiration();
```

`extends` signifie "hérite de".

- *// un Felin peut utiliser les*
- *// méthodes publiques de ses*
- *// super-classes*

- `Mammifere mamm = felix;`
- `Animal an = felix;`

- *// upcast autorisé vers Mammifere*
- *// ou vers Animal*

Hiérarchie

- factorisation également au niveau de l'"état"
  - les attributs des super-classes sont des attributs de la classe héritante

```
public class Animal { // sous-entendu : "extends Object"
public Habitat habitat;
public String nom;
}

public class Mammifere extends Animal {
}
public class Felin extends Mammifere {
}

// utilisation ...
Felin felix = new Felin();
felix.habitat = Habitat.TERRESTRE;
felix.nom =  "Felix";
```


Hiérarchie - Extension

Extension du comportement

la sous-classe peut ajouter des nouveaux comportements
la classe héritante est donc une extension de la classe héritée

```
public class Mammifere extends Animal {
   public String organeDeRespiration() {
      return "poumons";
   }
}
public class Felin extends Mammifere {
   public int getNbDePattes() {
      return 4;
   }
}
```

- un objet `Felin` peut invoquer `organeDeRespiration` et `getNbDePattes`
  `Felin  Mammifere  Animal`
- au sens où un `Felin` peut se faire passer pour un `Mammifere` ou un `Animal` :
- tout message envoyé à un `Mammifere` peut l'être à un `Felin`

Hiérarchie - Spécialisation

Spécialisation du comportement

une classe héritante peut redéfinir un comportement défini dans une super-classe
c'est ce comportement qui est utilisé par ses instances
on parle de surcharge de méthode

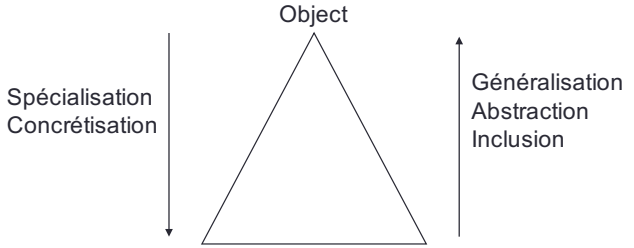
```
public class Mammifere extends
Animal {
  public int getNbDePattes() {
    return 4;
  }
}
public class Cetace extends
Mammifere {
  public int getNbDePattes() {
    return 0;
  }
}
```

```
// utilisation
Cetace cet = new Cetace();
System.out.println(cet.getNbDePattes());
// affiche 0

Mammifere mam1 = new Mammifere();
System.out.println(mam1.getNbDePattes());
// affiche 4

Mammifere mam2 = cet;
System.out.println(mam2.getNbDePattes());
// affiche 0
```

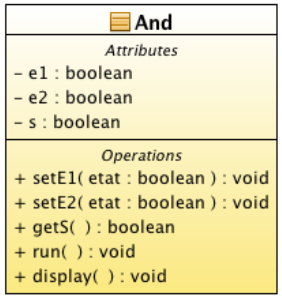

Hiérarchie (arbre) de classes

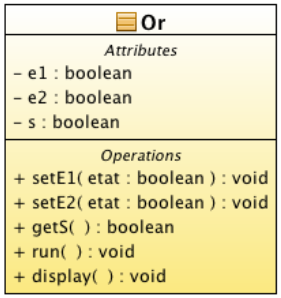


	classe	sous-classe de
Interprétation extensionnelle	ensemble (type)	inclusion (sous-type)
Interprétation intensionnelle	description d'objet	généralisation/ spécialisation abstraction/ concrétisation

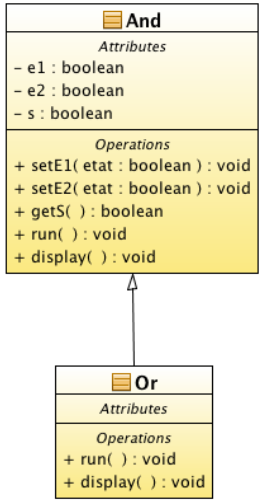
Abstraction

- A partir de plusieurs classes semblables, abstraire une sur-classe commune
- Factorisation de code



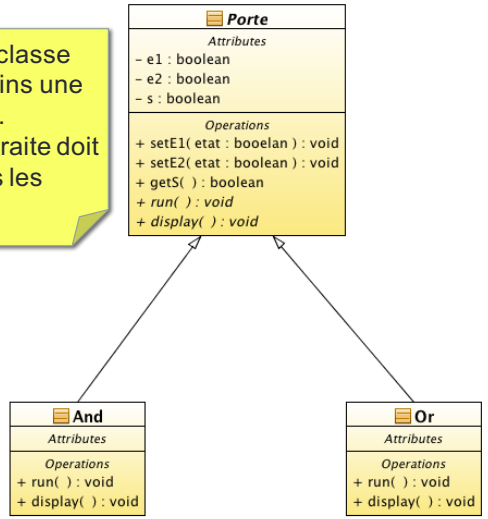


Erreur de conception…



Solution : surclasse abstraite

Classe abstraite : classe qui contient au moins une méthode abstraite.
Une méthode abstraite doit être redéfinie dans les classes filles



Classe abstraite

Une classe abstraite ne peut pas avoir d'instance. Son constructeur ne peut être appelé que via super().

```
abstract class Porte {

// variables d'instance
 boolean e1, e2, s;

// methodes
      void setE1(boolean etat) {e1 = etat;}
      void setE2(boolean etat) {e2 = etat;}
      boolean getS() {
            run();
            return s;}
      abstract void run();
      abstract void display();
}
```


Classe abstraite

```
class And extends Porte {
      void run() {
            s = e1 && e2;
      }
      void display() {...}
}

class Or extends Porte {
      void run() {
            s = e1 || e2;
      }
      void display() {...}
}
```

`extends` signifie "hérite de". Valable que la classe mère soit abstraite ou concrète.

Si `display()` n'est pas redéfinie, il y a erreur de compilation. C'est peut-être qu'elle ne devait pas faire partie de `Porte`

Abstraction

- **Classes et méthodes abstraites**
  - Classe abstraite = non instanciable
  - Spécifie des méthodes abstraites, implantées dans les sous-classes
- **Méthodes génériques**

  Dans la sur-classe, les méthodes qui font référence par this-message à des méthodes abstraites sont implicitement «génériques » pour les sous-classes.

Méthodes génériques

```
abstract class Porte {
 boolean getS() { // generique
      this.run();
      return s;
 }
 abstract void run();
...}
{// programme utilisateur
 And a = new And();
 Or o = new Or();
 a.setE1(true);
    a.getS() // --> false
 o.setE1(true);
    o.getS() // --> true...}
```

Type statique de `this` : la classe
Type dynamique de `this` : la classe d'instanciation de l'objet (→ sous classe)

Remarques sur l'héritage

- Le constructeur de la sous classe doit prendre en charge l'intégralité de la construction de l'objet
  - L'initialisation de certains champs de la super classe se fait par appel de fonction d'altération (s'ils sont bien encapsulés) ou par appel du constructeur de la super classe.
- En cas d'appel au constructeur de la super classe :
  - Il doit s'agir de la première instruction par le mot clé **`super`**
    - Pas d'appel de **`this`** et **`super`** possible dans le même constructeur
- Pas d'héritage multiple en Java mais notions d'interface
- Pour compiler une sous classe, il est nécessaire que la super classe ait déjà été compilée (ou appartienne au même fichier)

Remarques sur l'héritage

- Exemple 1

```
class A {
  public A() {…}          //constructeur 1
  public A (int n) {…} //constructeur 2
}
class B extends A {
 … //pas de contructeur
}
B b = new B() //
```

- Exemple 2

```
class A {
  public A (int n) {…} //constructeur 2
}
class B extends A {
 … //pas de contructeur
}
```

Erreur de compilation

- Exemple 3

```
class A {
  … //pas de contructeur
}
class B extends A {
 … //pas de contructeur
}
B b = new B()
```

Appel du constructeur par défaut de B qui appelle le constructeur par défaut de A.

Polymorphisme de surcharge

- Lors de la surcharge (également connue sous le nom de surdéfinition ou *overloading* en anglais) le nombre et le type des arguments fournis varient.
- Possible au sein d'une même classe
- Possible entre une sous classe et une super classe

```
class A{
  public void f (int n) {…}
}
class B extends A {
  public void f (float x) {…}
}
A a; B b;
int n; float x;
…
a.f(n) // appelle f (int) de A
a.f(x) // erreur de compilation : une seule méthode acceptable
       //(f(int) de A et on ne peut convertir x de float en int
b.f(n) // appelle f (int) de A
b.f(x) // appelle f (float) de B
```

Polymorphisme : une même opération peut être définie différemment éventuellement dans des classes distinctes.

Polymorphisme de redéfinition

- Lors de la redéfinition le nombre et le type des arguments fournis sont identiques.
- Possible entre une sous classe et une super classe

```
class Point {
  …
  public void affiche(){
    System.out.println("Je suis en " + x + " " + y);
  }
}
class PointColore extends Point {
  …
  public void affiche(){
    super.affiche();
    System.out.println("et ma couleur est " + couleur);
  }
}
```

Il peut y avoir redéfinition sans forcément appel à `super`

- Polymorphisme indirect

```
a.getS(); // Porte:getS() -> And:run()
o.getS(); // Porte:getS() -> Or:run()
```

- La redéfinition de méthode s'applique à une classe et toutes ses sous classes jusqu'à ce qu'éventuellement l'une d'elles la redéfinisse à nouveau

Hiérarchie de classes et typage

- **Hiérarchie de classes => hiérarchie de types**
  - Tout objet instance d'une classe peut être considéré du type de ses sur-classes
  - Ou inversement : partout où l'on attend un objet d'une classe donnée, tout objet d'une sous-classe convient
- **Variable polymorphe**

  Soit x une variable de type C, x peut référencer :
  - tout objet instance de C (typage "fort" classique)
  - mais aussi tout objet instance d'une sous-classe de C (**typage souple**)

Affectation polymorphe : Exemple

```
Porte p;
And a1 = new And(), a2;
Or o;
Rectangle r;

// affectations valides : typage fort classique
a2 = a1 ;
// affectations non valides : '' horizontales ''
a2 = o;
p = r;

// typage souple
// affectations '' verticales '' toujours valides : upcast
p = a1; p = o;
// affectations ''verticales '' hypothétiques : downcast
a2 = p; // non valide en général sauf...
a2 = (And)p; // downcast valide si...
if (p instanceof And) a2=(And)p; // sinon ClassCastException
```


Variable et liaison dynamique

- Type statique d'une variable = type de la déclaration
  - il détermine, à la **compilation**, les opérations applicables (dont les abstract déclarées)
- Type dynamique d'une variable

  = type de la valeur à l'exécution

  = type de l'objet référencé
  - il détermine les opérations effectivement appliquées (parmi celles applicables)
  - liaison dynamique des méthodes
- Ceci s'applique à toute catégorie de variable (this, variable d'instance, locales, paramètres, indexée, ...)

Liaison dynamique sur this

```
abstract class Porte {
 boolean getS() { // generique
      this.run(); // type statique de this = Porte
      return s;
 }
 abstract void run();
 ...
}

{// programme utilisateur
 And a = new And();
 Or o = new Or();
 a.setE1(true);
    a.getS() // => this.run()
      // type dynamique de this = And
 o.setE1(true);
    o.getS() //=> this.run()
      // type dynamique de this = Or
```


Liaison dynamique sur paramètres

```
// exemple de procedure dans une application
// utilisatrice de Porte's ...
boolean test(Porte p) {
 p.setE1(true);
 return p.getS();
}

And a = new And();
Or o = new Or();
Porte p;
test(a); // --> false
test(o); // --> true
p = quellePorte();
test(p);
```


Liaison dynamique : SD

- Une structure de données (tableau, liste, ...) peut contenir des objets de toute sous-classe (type dynamique) de la classe déclarée (type statique) pour ses éléments.
- SD hétérognènes

```
Exemple
Porte[] circuit = new Porte[n]; // type statique

circuit[0] = new And(); // types dynamiques ...
circuit[1] = new Or();
circuit[2] = new Nand();...

for (int i=0;i<n;i++) {
 circuit[i].setE1(false);
 circuit[i].setE2(false);
 circuit[i].run();
}
```

```
for (Porte p : circuit) {
  p.setE1(false);
  p.setE2(false);
  p.run(); //polymorphe
}
```


Recherche de méthode

- Processus en deux étapes :
  1. La recherche débute dans la classe du receveur
  2. Si la méthode est définie dans la classe, elle est retournée

  Sinon, la recherche continue dans les superclasses de la classe du receveur. Si aucune méthode n'est trouvé et s'il n'y a plus de superclasse à explorer (classe Object), il y a une erreur.

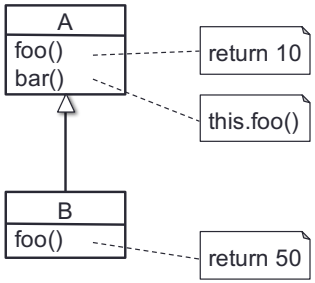
Recherche de méthode

- `ab = new A()`
- Que vaut `ab.foo()` ?

- `ab = new B()`
- Que vaut `ab.foo()` ?

- `ab = new A()`
- Que vaut `ab.bar()` ?

- `ab = new B()`
- Que vaut `ab.bar()` ?



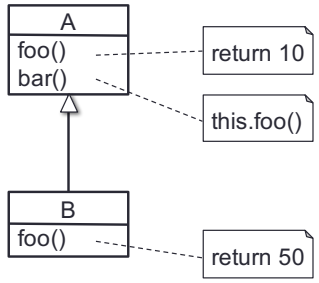
Recherche de méthode

- `ab = new B()`
- Que vaut `ab.foo()` ?

1. `ab` est de classe B
2. `foo()` est-elle définie dans B ?
3. `foo()` est exécutée ➡ 50

- `ab = new B()`
- Que vaut `ab.bar()` ?

1. `ab` est de classe B
2. `bar()` est-elle définie dans B ?
3. `bar()` est-elle définie dans A ?
4. `bar()` est exécutée
5. `this` est de classe B ?
6. `foo()` est-elle définie dans B ?
7. `foo()` est exécutée ➡ 50



`this` représente **TOUJOURS** le receveur.

Sémantique de super

- Similairement à this, super est une pseudo variable qui fait référence au receveur du message.
- Il est utilisé pour invoquer des méthodes surchargées.

- Quand on utilise this, la recherche de la méthode commence dans la classe du receveur.

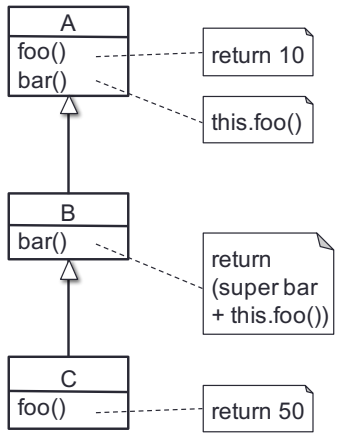
- Quand on utilise super, la recherche de la méthode commence dans la superclasse de la classe de la méthode contenant l'expression super.

Sémantique de super

- `ab = new A()`
- Que vaut `ab.foo()` ?
- Que vaut `ab.bar()` ?

- `ab = new B()`
- Que vaut `ab.foo()` ?
- Que vaut `ab.bar()` ?

- `ab = new C()`
- Que vaut `ab.foo()` ?
- Que vaut `ab.bar()` ?



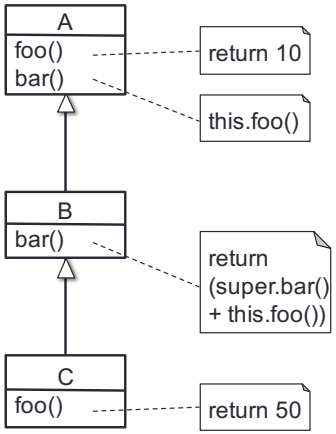
Sémantique de super

- `ab = new A()`
- Que vaut `ab.bar()` ?
- 10

- `ab = new B()`
- Que vaut `ab.bar()` ?
- 10 + 10

- `ab = new C()`
- Que vaut `ab.bar()` ?
- 50 + 50

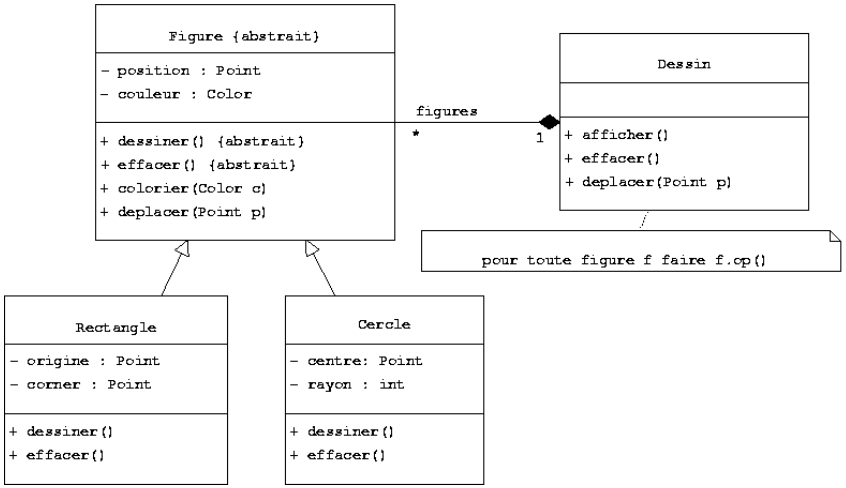
`super` **n'est pas** la super classe du receveur.



Etude de cas

- *Décrire des figures (rectangles, cercles, ...)*
  - *colorées, et dont on doit pouvoir changer la couleur*
  - *positionnées, et que l'on doit pouvoir effacer et déplacer.*
- Identification des objets
  - Rectangle
  - Cercle, …
- Munis du même protocole :
  - dessiner()
  - effacer()
  - colorier(Color c)
  - deplacer(Point p)
- Mêmes spécifications
  => sur-type commun : Figure

Etude de cas (suite)



Etude de cas (suite)

```
abstract class Figure {
// champs
 Point position;
 Color couleur;
// methodes
      abstract void dessiner ();
      abstract void effacer ();
      void colorier(Color c) { // generique
            couleur=c;
            this.dessiner();}
      void deplacer(Point p) { // generique
            this.effacer();
            position.translater(p);
            this.dessiner();}
}
```

Etude de cas (suite)

```
class Rectangle extends Figure {
 Point origine, corner;
 void dessiner() {...}
 void effacer() {...}
}

class Cercle extends Figure {
 Point centre;
 int rayon;
 void dessiner() {...}
 void effacer() {...}
}

...
```


Etude de cas (suite)

- *Un dessin est formé de figures. On doit pouvoir afficher, effacer et déplacer un dessin.*
- Identification

  Dessin :
  - afficher()
  - effacer()
  - deplacer(Point p)
- Structuration
  - Un dessin => une liste de Figure.
  - Algorithmes génériques sur les opérations afficher, effacer, deplacer :

```
pour toute Figure f faire f.operation() fait
```


Etude de cas (suite)

```
class Dessin {
// structure de donnees a voir...
 Figure figures[] = new Figure[N];
 int nbFigures;
// methodes
 void afficher() {
      for(int i=0;i<nbFigures; i++)
            figures[i].dessiner(); // polymorphe
 }
 void deplacer(Point p) {
      for(int i=0;i<nbFigures; i++)
            figures[i].deplacer(p); // polymorphe
 }
}
```


Qualités logicielles

- Extensibilité
  - Ajout d'un nouveau type de figure (Triangle)
  - Incrémental et modulaire (sans retouche du code existant)
- Réutilisation
  - Le code de Figure est réutilisable dans le nouveau sous-type
  - Programmation synthétique
- Généricité
  - Les portions de codes (applications) écrites à un niveau de la hiérarchie de classes sont applicables à toutes les sous-classes
  - les programmes restent applicables à toute nouvelle sous-classe

Si l'héritage n'existait pas…

- 1ère solution
  - Ensemble de types à plat et définir des opérations différentes: {Rectangle, Cercle, …} X {dessiner, effacer, …}
  - Pas de sur-type Figure => on ne peut regrouper les entités de types différents dans une même SD
- 2ème solution
  - structures à champs variants
  - record Pascal ou ADA, unions C
  - type et programmation «tagués»
- Qualités
  - Permet de simuler « à la main » le polymorphisme et la généricité
  - Peu efficace et risque d'erreur
  - Peu modulaire, maintenable et extensible

Programmation « taguée »

```
type Tags = (rectangle, cercle);

type figure (tag : Tags) = structure
 position : Point;
 couleur : Color;
 cas tag =
     cercle : rayon : int; centre : Point;
     rectangle : origine, corner : Point;

type Figures = tableau[n] de Figure;
```


Programmation « taguée »

```
// polymorphisme « à la main »

procedure dessiner (x : Figure)
 cas x.tag =
     rectangle : ... code ....
     cercle : ... code …

procedure effacer (x : Figure)
 cas x.tag =
     rectangle : ... code ....
     cercle : ... code ...
```


Programmation « taguée »

```
// généricité de Figure « simulée »
procedure deplacerFigure(f : Figure, p : Point)
 effacer(f);
 translater(f, p);
 dessiner(f)
procedure colorierFigure ...

// généricité de l'application Dessin « simulée »
procedure afficherDessin (f: Figures)
      pour i de 1 a n faire dessiner(f);
procedure deplacerDessin (f: Figures, p: Point)
      ...
```