Problème de réutilisabilité : comment écrire une classe qui possède tous les membres (attributs ou méthodes) d'une classe et d'autres en plus, sans tout réécrire?

```
class Salarie{
  String nom;
  int id;
  Contrat c;
  float salaire_fixe;
  void obtenirAugmentation(float f)...
  void changerContrat(Contrat c)...
}
```

```
class Commercial{
  String nom;
  int id;
  Contrat c;
  float salaire_fixe;
  float prime;
  void obtenirAugmentation(float f)...
  void changerContrat(Contrat c)...
  float ventesDuMois() ...
}
```

Problème de généralisation : comment éviter d'écrire plusieurs fois la même chose dans des classes qui ont des membres communs mais pas tous?

```
class Commercial{
  String nom;
  int id;
  Contrat c;
  float salaire_fixe;
  float prime;
  void obtenirAugmentation(float f)...
  void changerContrat(Contrat c)...
  float ventesDuMois() ...
}
```

```
class CadreDirigeant{
  String nom;
  int id;
  Contrat c;
  float salaire_fixe;
  float stockOptions;
  void obtenirAugmentation(float f)...
  void changerContrat(Contrat c)...
  void parachuteDore(float f) ...
}
```

Solution conforme aux principes du génie logiciel : l'héritage.

Une classe B hérite d'une classe A signifie que tout membre de A (sauf les constructeurs) est aussi membre de B.

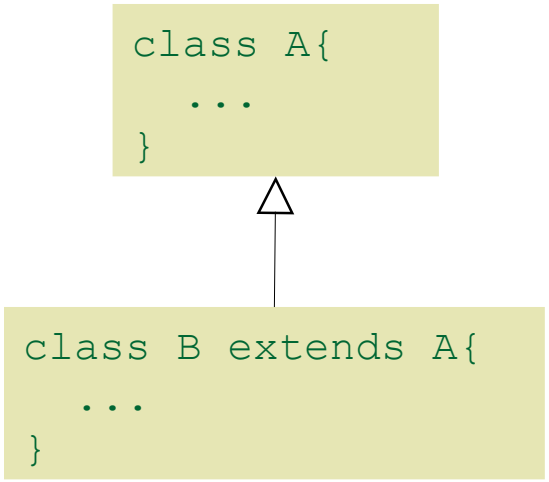
```
class Salarie{
  String nom;
  int id;
  Contrat c;
  float salaire_fixe;
  void obtenirAugmentation(float f)...
  void changerContrat(Contrat c)...
}
```

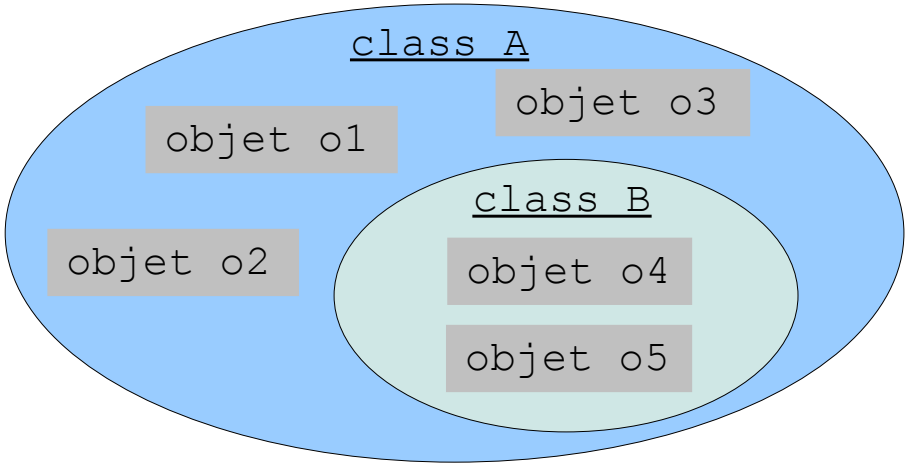
```
class Commercial extends Salarie{
  float prime;
  float ventesDuMois() ...
}
```

```
class CadreDirigeant extends Salarie{
  float stockOptions;
  void parachuteDore(float f) ...
}
```

Si B hérite de A, A est super-classe de B et B est sous-classe de A.

Il s'agit d'une inclusion ensembliste : toute instance de B est instance de A.





L'héritage est transitif : si B hérite de A et C hérite de B, alors C hérite de A.

```
objet o1 instance de Salarie

nom : "Toto"
id : 3
c : ...
salaire_fixe : 2000,00

void obtenirAugmentation(float f)...
void changerContrat(Contrat c)...
```

```
objet o2 instance de Commercial
(et donc aussi instance de Salarie)

nom : "Titi"
id : 5
c : ...
salaire_fixe : 2500,00
prime : 500,00

void obtenirAugmentation(float f)...
void changerContrat(Contrat c)...
float ventesDuMois() ...
```

```
objet o3 instance de CadreDirigeant
(et donc aussi instance de Salarie)

nom : "Tutu"
id : 21
c : ...
salaire_fixe : 5000,00
stockOptions : 50000,00

void obtenirAugmentation(float f)...
void changerContrat(Contrat c)...
void parachuteDore(float f) ...
```

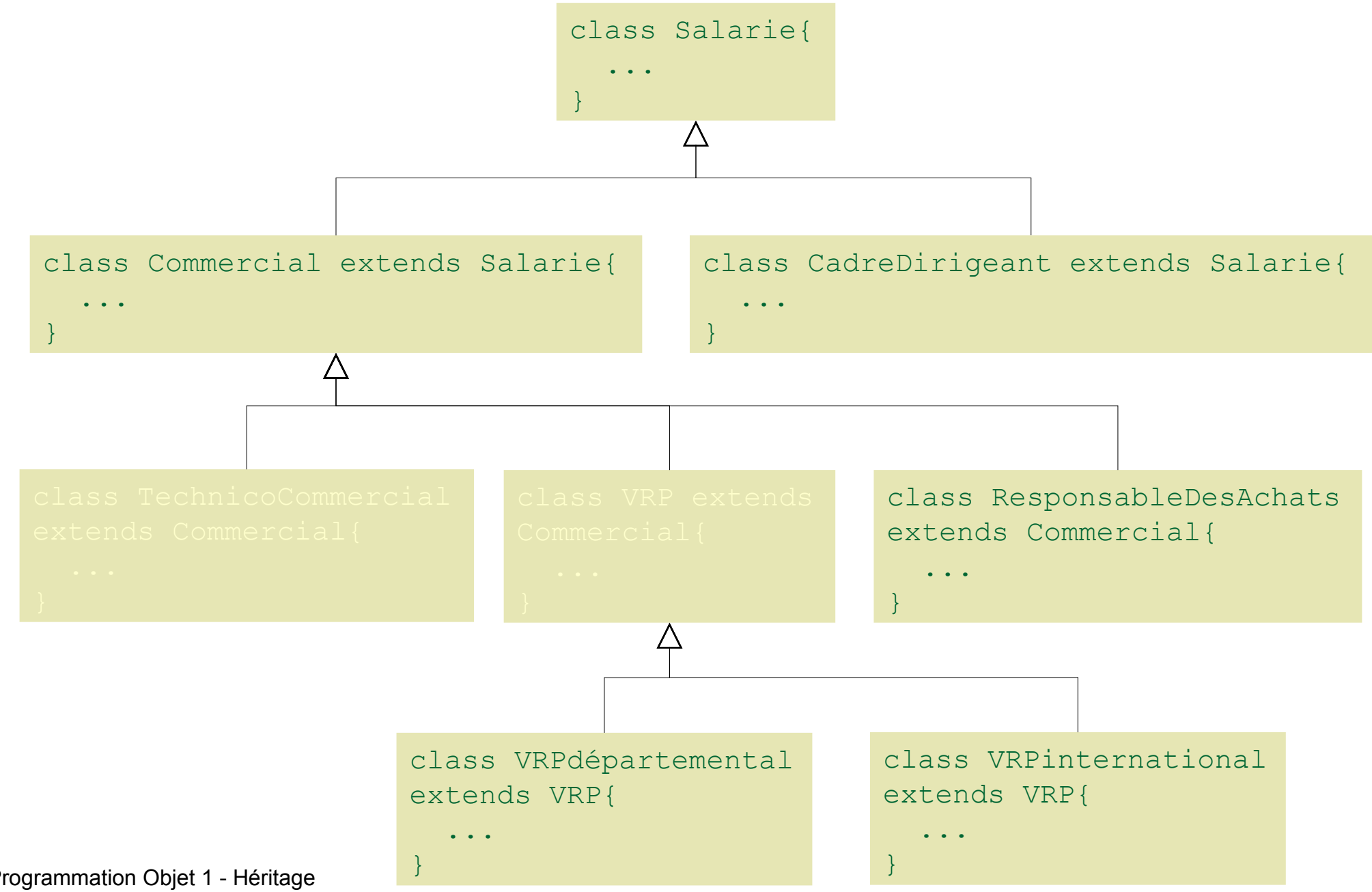
L'héritage est une généralisation, une relation sorte-de (ou est-un) entre classes.

Attention : héritage et composition sont deux notions bien distinctes.

Ex.: un chat est composé de quatre pattes, mais il est une sorte d'animal

Attention : héritage et abstraction sont deux notions bien distinctes.

Un programme n'est jamais trop modulaire, il ne faut pas hésiter à créer de nombreuses classes, structurées horizontalement mais aussi verticalement.



Le mot-clé `super` permet d'accéder aux membres de la super-classe.

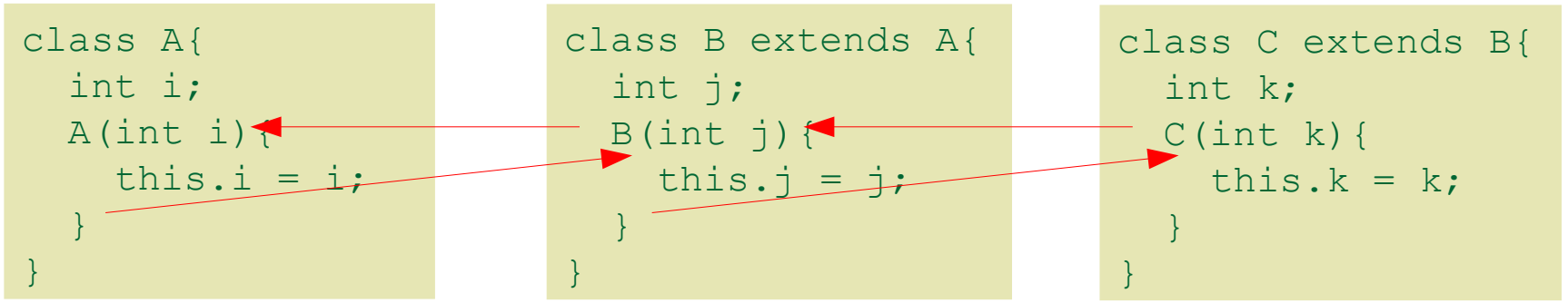
Par défaut, le préfixage est `this`. Le préfixage des membres n'est nécessaire qu'en cas d'ambiguité.

```
class Quadrupède{
  int nbPattes = 4;
  void marche(int vitesse)...
}
```

```
class Cheval extends Quadrupède{
  void galope(){
    super.marche(30);
  }
  void trotte(){
    marche(15);
  }
  void vaAuPas(){
    this.marche(5);
  }
  boolean peutGaloper(){
    return super.nbPattes == 4;
  }
}
```

`super.super` est interdit afin d'éviter de passer « par dessus » une classe lors d'un appel de constructeur.

Lors de la construction d'un objet, le constructeur de la super classe est d'abord appelé (automatiquement ou non), puis celui de la classe.



L'objet n'est totalement créé qu'une fois la chaine des constructeurs redescendue : `this` n'existe donc pas tant que l'enchainement des appels de constructeurs n'est pas terminé.

Un appel explicite au super-constructeur est forcément la première instruction du constructeur.

```
class B extends A{
  int j;
  B(int i, int j){
    super(i);
    this.j = j;
  }
}
```

Un appel au super-constructeur peut être implicite ou se faire dans un autre constructeur qui est appelé.

```
class B extends A{
  int j;
  B(int i){
    super(i);
    this.j = 0;
  }
}
```

Si dans le constructeur de la sous-classe le constructeur de la super-classe n'est pas explicitement appelé, le compilateur cherche dans la super-classe un constructeur sans paramètre (constructeur par défaut éventuellement).

S'il n'en trouve pas, il refuse de compiler.

Si la sous-classe ne définit aucun constructeur, la super-classe doit avoir un constructeur sans paramètre.

Conclusion : il est préférable de toujours définir explicitement un constructeur sans paramètre.

Si la classe B hérite de la classe A, tout objet instance de B est instance de A.

Le transtypage ascendant (upcasting) : il consiste à manipuler un objet comme instance d'une classe dont hérite (directement ou non) la classe qui a créé l'objet.

Transtypage implicite :

```
B toto = new B();
A titi = toto;
```

Transtypage explicite : on transtype (cast) l'objet en le faisant précéder du nom de la super classe entre parenthèses.

```
B toto = new B();
((A) toto).m();
```

Transtypage descendant (downcasting) : il consiste à manipuler un objet comme instance d'une classe qui hérite (directement ou non) de celle qui a créé l'objet.

Le transtypage descendant est forcément explicite.

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

Pour que le transtypage descendant fonctionne à l'exécution, il faut que l'objet soit bien du type de la classe dans lequel on le transtype.

On peut tester sa classe réelle à l'aide du mot clé `instanceof`

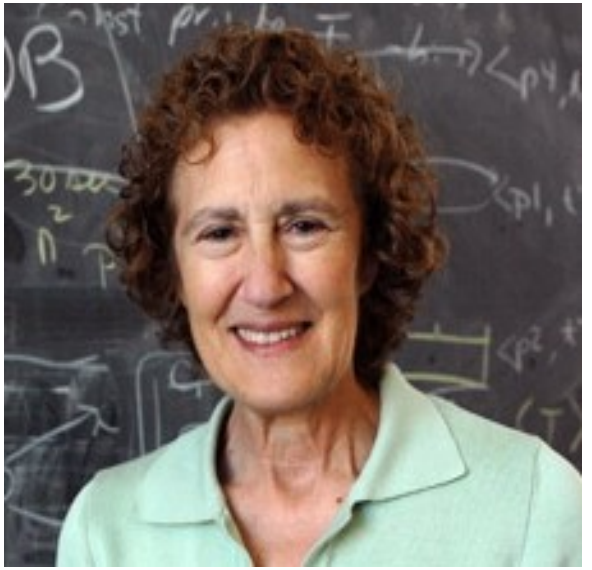
```
A a = new B();
if(a instanceof B){
  B b = (B)a;
}
```

L'API Java offre à la fois des types primitifs (`byte, short, int, long, float, double, boolean, char`) et les classes correspondantes (`Byte, Short, Integer, Long, Float, Double, Boolean, Character`).

Il est possible d'affecter, sans transtypage, une valeur de type primitif à une variable de type la classe correspondante, et vice-versa.

```
int i = 2;
Integer j = i;
i = j;
```

Principe de substitution de **Barbara Liskov** : une sous-classe doit pouvoir être utilisée dans toutes les situations où sa classe parente peut l'être, sans qu'il puisse être possible de percevoir une différence.



Ce principe de bonne programmation est violé par la redéfinition d'attribut ou de méthode.

```
class Chien{
  ...
  public String cri(){
    return "ouahouah";
  }
}
```

```
class Caniche extends Chien{
  ...
  public String cri(){
    return "wifwif";
  }
}
```

La définition dans une sous-classe d'attributs portant les mêmes noms que des attributs de la super-classe crée des ambiguités au niveau de l'appel de ces attributs.
La définition dans une sous-classe de méthodes portant les mêmes noms que méthodes de la super-classe peuvent créer des ambiguités au niveau de l'appel de ces méthodes.

```
class A{
  int i = 0;
  String s;

  void toto()...
  void toto(String chaine) ...
}
```

```
class B extends A{
  int i = 1;
  float s, toto;

  void toto() ...
  int toto(String chaine) ...
}
```

```
objet o instance de B (et donc de A)

i (de A) : 0
i (de B) : 1
s (de A) : "Truc"
s (de B) : 12,5
toto (de B) : 56,4

void toto() (de A) ...
void toto (String chaine) (de A) ...
void toto() (de B) …
int toto (String chaine) (de B) ...
```

```
B b = new B();
b.toto();
b.i;
A a = new B();
a.toto();
a.i;
```

Masquage d'attribut : la sous classe redéfinit un attribut avec le même nom qu'un attribut défini dans la super-classe (ou plus haut). Le type n'importe pas.

L'attribut de la classe mère n'est pas écrasé par celui de la classe fille, mais masqué

```
class A{
  int i = 0;
  ...
}
```

```
class B extends A{
  String i = "a";
  ...
}
```

```
B b = new B();
System.out.println(b.i); // affiche a
```

Il y a ambiguité si l'objet de type B est stocké dans une variable de type A.

```
A a = new B();
a.i ?
```

En Java, le polymorphisme d'attribut est résolu par liaison statique : le type des objets est vérifié à la compilation.

L'attribut considéré est donc celui de la classe de déclaration de la variable contenant l'objet.

```
A a = new B();
a.i // vaut 0
```

En C++, la résolution du polymorphisme d'attribut est également réalisé par liaison statique.

Surcharge de méthode (overloading ou polymorphisme ad hoc) : on définit dans une même classe ou dans des classes différentes des méthodes ayant le même nom mais pas les mêmes signatures (paramètres différents).

```
class Parser{
  ...
  int parseFile(String url){
    ...
  }
  boolean parseFile(String name,String directory){
    ...
  }
}
```

```
class TextParser extends Parser{
  ...
  int parseFile(InputStream s){
    ...
  }
}
```

Les types des paramètres passés lors de l'appel des méthodes suffisent pour lever les ambiguïtés si les signatures sont incompatibles.

En cas d'ambiguité lors de l'appel d'une méthode surchargée, le principe de liaison statique est appliqué aux paramètres : le type des valeurs des paramètres est donné par celui des variables qui les contiennent.

```
class TextInputStream extends InputStream{...}
```

```
class Parser{
  int parseFile(InputStream s){
    ...
  }
}
```

```
class TextParser extends Parser{
  int parseFile(TextInputStream s){
    ...
  }
}
```

```
Parser tp = new TextParser();
InputStream is = new TextInputStream("machin.truc");
tp.parseFile(is);
```

Si une valeur de paramètre n'est pas stockée dans une variable, c'est le type de son constructeur qui compte :

```
tp.parseFile(new TextInputStream("machin.truc"));
```

Redéfinition de méthode (overriding) : on définit dans une classe une méthode ayant le même nom et la même signature qu'une méthode de la super-classe.

```
class Parser{
  int parseFile(InputStream s){
    ...
  }
}
```

```
class TextParser extends Parser{
  int parseFile(InputStream s){
    ...
  }
}
```

```
Parser tp = new TextParser();
InputStream is = new TextInputStream("machin.truc");
tp.parseFile(is);
```

En Java, le polymorphisme de méthode est résolu par liaison dynamique (à l'exécution) : la méthode considérée est celle du type réel (type du construteur) de l'objet sur lequel la méthode est appelée.

Si le type réel de l'objet ne contient pas la méthode appelée, on remonte dans la hiérarchie des classes pour trouver la méthode.

```
class A{
  int toto(int i){
    ...
  }
}
```

```
class B extends A{
  int toto(int i){
    ...
  }
}
```

```
class C extends B{
}
```

```
A c = new C();
c.toto(3);
```

En C++, le redéfinition de méthode est résolue de façon statique, mais on peut imposer la liaison dynamique en qualifiant une méthode de virtuelle.

On parle aussi de surcharge de méthode lorsque les noms et signatures des méthodes sont les mêmes mais qu'elles sont définies dans des classes non liées par héritage.

Le polymorphisme ad hoc est surtout utile pour uniformiser les traitements d'objets de classes non liées par héritage.

```
class Carré{
  void afficher(Graphics g){
    ...
  }
  ...
}
```

```
class Image{
  void afficher(Graphics g){
    ...
  }
  ...
}
```

```
MachinGraphique[] mg = ...
...
for(int i = 0;i<mg.length;i++){
  mg[i].afficher(mongraphique);
}
```

```
class Triangle{
  void afficher(Graphics g){
    ...
  }
  ...
}
```

La covariance consiste à redéfinir une méthode en changeant son type de retour uniquement.

Pour qu'il y ait redéfinition de méthode, le type de retour de la méthode redéfinie doit être un sous-type du type de retour de la méthode de départ. Sinon il y a surcharge de méthode.

```
class Chef{
  Salarie souffreDouleur(){
    ...
  }
}
```

```
class ChefAtelier extends Chef{
  Ouvrier souffreDouleur(){
    ...
  }
}
```

Certains langages utilisent le typage dynamique : le type des objets n'est connu qu'à l'exécution (Smalltalk, CLOS, Python, PHP).

→ la résolution du polymorphisme est alors toujours dynamique

Dans d'autres langages, les types sont spécifiés par le programmeur (Java, C++, C#), ou déduits du contexte à la compilation (OCaml, Haskell).

→ la résolution du polymorphisme peut être dynamique ou statique

En C++, le polymorphisme de méthode est résolu par liaison statique. Pour que la résolution du polymorphisme soit dynamique, il faut utiliser des méthodes « virtuelles ».

Le polymorphisme ne doit être utilisé que lorsqu'il répond à une réelle exigence de représentation des données.

Un attribut `final` (`const` en C++) est une constante (sa valeur est fixée par la première affectation). Par convention, un attribut `final` est écrit en majuscule.

```
class Insecte{
 final int NB_PATTES = 6;
  ...
}
```

Un attribut `final` peut cependant être masqué. On peut contourner ce problème en remplaçant l'attribut par un “faux” accesseur impossible à redéfinir.

Le mot clé `final` sur une méthode empêche qu'elle soit redéfinie dans une sous-classe.

```
class Insecte{
 final int getNB_PATTES(){
    return 6;
  }
  ...
}
```

On peut interdire de spécialiser une classe en la déclarant `final`.

```
final class Maïs{
   ...
}
```

```
class Maïs fantastique extends Maïs{
   ...
}
```

Le compilateur et la machine virtuelle peuvent optimiser l'utilisation de telles classes en évitant la gestion du polymorphisme.

Une bonne partie des classes de l'API Java sont `final` : les types de base (`String`, `Integer`, `Long`, `Float`, etc), des classes système (`System`, `Console`, etc), des classes réseaux (`URI`, `IDN`, etc), des classes pour la sécurité (`AccessControler`, `Permissions`, etc), …

En Java, toutes les classes héritent de la classe `Object` (même si on ne le précise pas dans la déclaration de la classe!).

```
public class Object{
  public Object(){...}
  protected Object clone(){...}
  public boolean equals(Object o){...}
  protected void finalize(){...}
  public Class getClass(){...}
  public int hashCode(){...}
  public String toString(){...}
  public void notify(){...}
  public void notifyAll(){...}
  public void wait(){...}
  public void wait(long timeout){...}
  public void wait(long timeout, int nanos){...}
}
```

Par défaut, l'appel de la méthode `o1.equals(o2)` renvoie vrai si les objets `o1` et `o2` ont la même adresse mémoire dans la JVM.

Par défaut, l'appel `o.toString()` renvoie l'adresse de l'objet `o` dans la JVM.

Ces méthodes sont destinées à être redéfinies :

```
public class Carre{
  int hauteur;
  int largeur;
  public boolean equals(Object o){
    return (this.hauteur == ((Carre) o).hauteur) &&
                              (this.largeur == ((Carre) o).largeur);
  }
  public String toString(){
    return "Ceci est un carré de hauteur " + hauteur +
                                          " et de largeur " + largeur;
  }
}
```

En Java (et Smalltalk, C#, etc), une classe ne peut hériter que d'une seule autre classe (en plus de Object). On parle alors d'héritage simple.

Certains langages (C++, Python, Eiffel, etc) permettent l'héritage multiple, qui complique la résolution du polymorphisme.

```
class Chose{
  Bidule b;
}
```

```
class Machin{
  Bidule b;
}
```

```
class Truc extends Chose, extends Machin{
  Truc(){
    this.b; // ??
  }
}
```

En C++, le polymorphisme est résolu par liaison statique, mais on peut forcer la liaison dynamique sur les méthodes « virtuelles ».