# Programmation Objet Java – Héritage

Michail Lampis
michail.lampis@dauphine.fr

# Héritage

- Héritage = un système de réutilisation de code de manière claire et quasi-automatique
- **Motivation :**
  - Projet qui contient plusieurs classes similaires
  - Plusieurs classes qui sont des spécialisations d’autres classes
- L’héritage permet de déclarer qu’une classe B est un cas spécial de la classe A
  - Chaque objet de B est un objet de A (+-)

# Classe Fichier

- Ex : Un programme qui fait de la gestion des fichiers
  - On définit une classe **File**

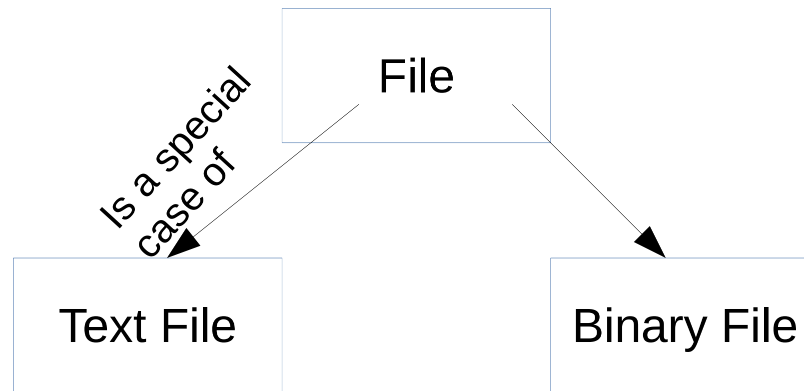
```
public class File {
    String name ;
    int size ;
    public void rename(String newName) ;
    ...
}
```

- Cette classe contient tous les champs (attributs) et toutes les méthodes pertinentes pour la manipulation d'un fichier

# Classe Fichier

- **Mais :** il existent des plusieurs types de fichier
  - Fichier texte : il faut ajouter de champs, par exemple pour le codage, méthodes pour l'affichage, etc.
  - Fichier binaire : Comment définir les champs/méthodes ci-dessus dans ce cas ?

# Fichiers - Ontologie



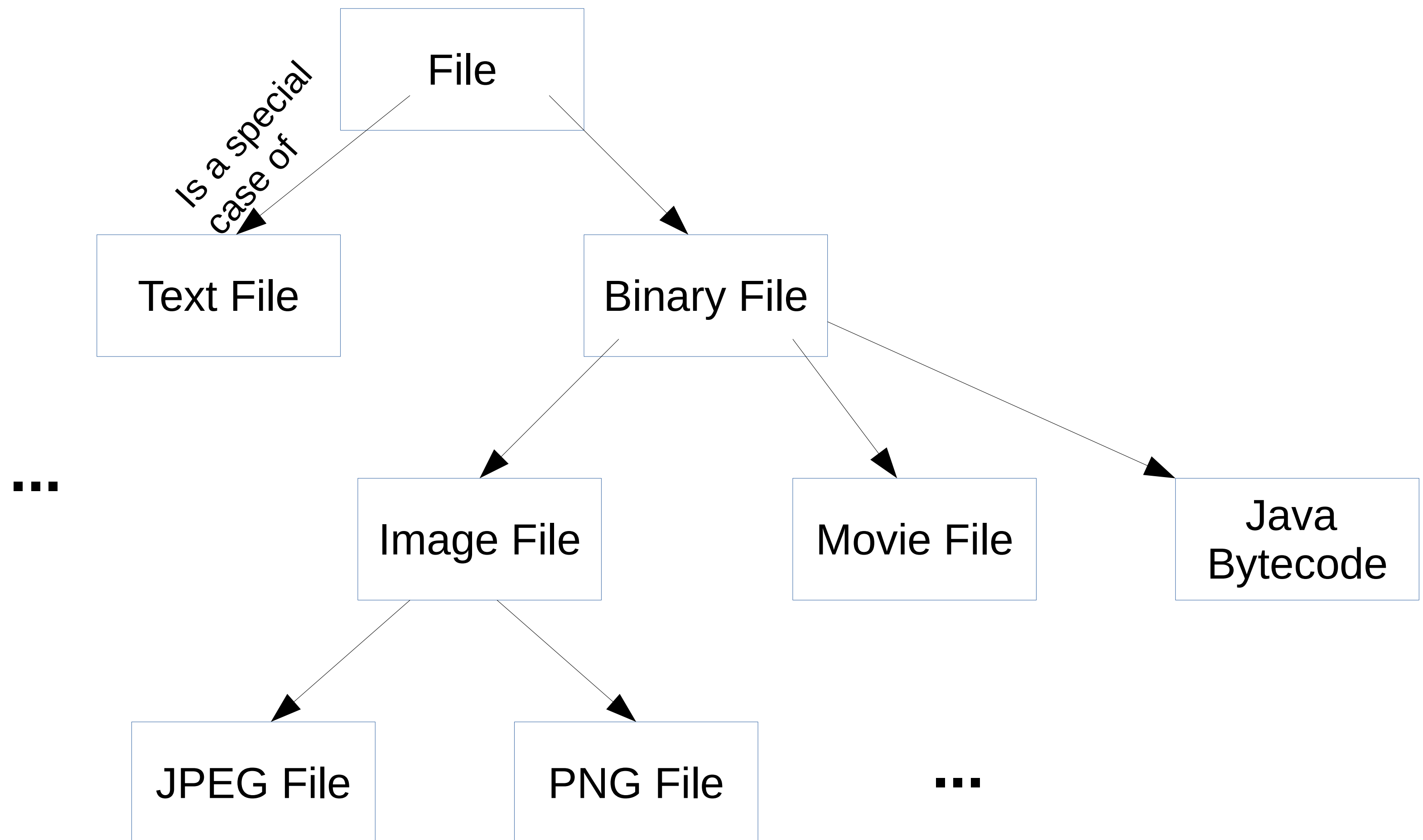
# Spécialisation

- Problème :
  - On a une classe de base (**File**)
  - On voulait ajouter de fonctionnalité pour quelques instances (objets) de cette classe, mais pas pour les autres
- Solutions classiques :
  - On ajoute le fonctionnalité dans la classe de base, et on fait un **if** au début de chaque fonction
  - On écrit une nouvelle classe

# Solution 1

```
public class File {
    String name ;
    int size ;
    boolean isText ;
    public void rename(String newName) ;
    public void affiche(){//que pour les Textes
        if(!isText) {return;}//Error ?
        ...
    }
    ...
}
```

# Fichiers - Ontologie



# Solution 1 ?

- Problème : « Solution doesn’t scale ! »
  - Solution ne respecte pas la séparation du programme
  - Mène à une classe énorme et méthodes avec beaucoup de sous-cas
    - Code difficile à lire/maintenir/modifier

# Solution 2

```
public class TextFile {
    String name ;
    int size ;
    public void rename(String newName) ;
    public void affiche(){//que pour les Textes
        //if(!isText) {return;} – Pas nécessaire, l'objet a type
        //TextFile
        ...
    }
    ...
}
```

# Solution 2 ?

- Problème : duplication du code
  - Si on commence une nouvelle classe indépendante pour représenter les fichiers texte on perd les méthodes existantes qui travaillent pour leurs objets
  - Mène à la solution « copy-paste »
    - À EVITER !!
    - Utiliser des copies du même code dans plusieurs classes est un signe de la mauvaise programmation
    - Bug-fix → il faut corriger toutes les copies
    - Nouvelle version → il faut changer toutes les copies
    - ...

# Héritage

- L’héritage permet de définir **la classe TextFile par différence**.
  - On dit : un TextFile est un File, sauf que,...
- Avantage :
  - On réutilise le code pour File (et ses modifications eventuelles)
  - La classe File n’est pas concerné

# Héritage – Java

**class File { …}**

**class TextFile extends File {…}**

- Class TextFile : **sous-classe** ou classe enfant
- Class File : **super-classe** ou classe parent
- Tous le membres de la super-classe sont hérités
  - Ils existent automatiquement dans la sous-classe
- Les deux classes ont une relation **is-a**
  - Un TextFile est un File…

# Is-a vs Has-a

- À comparer : Héritage vs Composition
  - Is-a vs Has-a

```
class Triangle {
    Point p1,p2,p3 ;
    …
}
```

- Vs

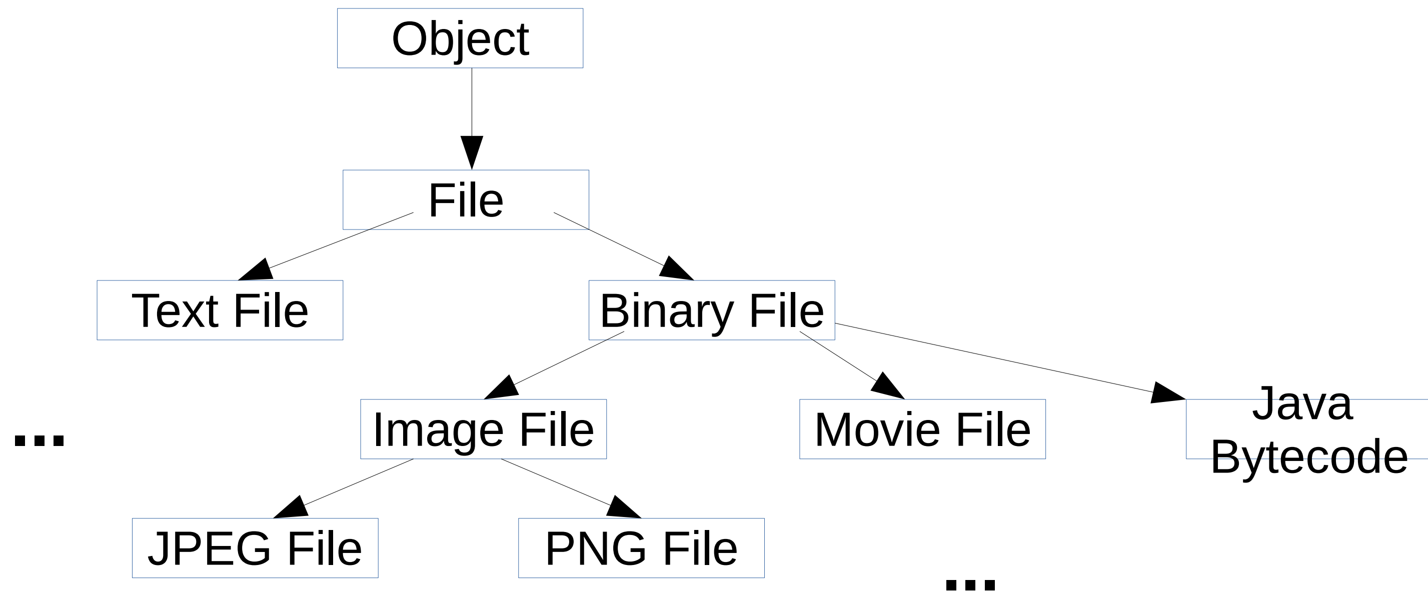
```
class TextFile extends File {…}
```

- Différence pas claire parfois, utilisez votre jugement…
- Question à poser : Va-t-on utiliser les objets de la sous-classe où on s’attend à un objet de la super-classe ?

# Hiérarchies de classes

- Une classe peut être une sous-classe d'une seule autre classe (avec **extends**)
- La relation sous-classe est transitive
  - C extends B, B extends A → C sous-classe de A
- Toutes les classes (non-primitives) sont des sous-classes de la classe **Object**
- → On a une hiérarchie arborescente, avec **Object** comme racine.

# Ontologie et Hiérarchie




**class File {..} //**extends Object Automatiquement
**class TextFile extends File {..}**
**class BinFile extends File {..}**
**class ImageFile extends BinFile {..}**
**..**

# Héritage – Exemples

- Toutes les classes étendent **Object**
  - Elles héritent ses méthodes
  - Ex : **String toString() ;**
  - Définition pour **Object**: retourne un String qui contient le nom de la classe et un Hash Code.
- Ex :

```
class Point { double x,y ; ..
public static void main(String[]args){
    Point p = new Point() ;
    System.out.println(p.toString()) ;//ou S.o.p(p) ;
    }
}
```

# Héritage – Exemples

- Fichier ColorPoint.java

```
class ColorPoint extends Point {
        int color;
        int getColor() { return color; }
}
```

- Un objet de la classe ColorPoint
  - Contient touts les champs et les méthodes de la classe Point
  - Contient en plus le champ color, et une méthode
  - Attn : La sous-classe n’hérite pas les constructeurs !!

```
ColorPoint p = new ColorPoint(2.2,3.3) ; // !!!!
ColorPoint p = new Point(2.2,3.3) ;// !!!
```

# Règles d'accès

- La sous-classe hérite tous les champs, mais :
  - Ne peut pas accéder aux attributs/méthodes **private**
  - Peut utiliser les attributs/méthodes **public** (disponible pour tous) et **protected** (disponible pour les sous-classes)
  - Peut utiliser les attributs/méthodes package-private (accès pas précisé) **seulement si** elle fait partie du même package.

```
class A { private int x ; int y ; protected z ; public w;}

class B extends A{

    void f(){ x=2 ;// !!!

    y = 3 ;// OK si dans le même package

    z = 2 ;//OK

    w = 5 ;//OK

}
```

# Règles d'utilisation

- On peut utiliser un objet d'une sous-classe où on peut utiliser un objet de la super-classe.
- Pas à l'autre sens !

**class Point{.. double distance(Point p){..} }**

**class ColoredPoint extends Point {..}**

**..**

**Point p = new ColoredPoint() ;**//OK, constr défaut

**ColoredPoint p = new ColoredPoint() ;**//OK

**ColoredPoint p = new Point() ;**// !!

# Règles d'utilisation

```
class Point{.. double distance(Point p){..} }
class ColoredPoint extends Point {..}
..
Point p1 = new ColoredPoint() ;
ColoredPoint p2 = new ColoredPoint() ;
p1.distance(p2) ;//OK
p2.distance(p1) ;//OK
```

- La méthode distance existe pour les objets ColoredPoint (méthode héritée), donc p2.distance est OK
- On peut donner comme paramètre un ColoredPoint où un Point est attendu.

# Redéfinition des Champs

- Une sous-classe peut réutiliser le même nom pour un attribut, même avec un autre type
  - Effet:l’attribut de la super-classe est occulté
- La sous-classe peut également augmenter l’accès à un champ
  - **public > protected >** (package-private) **> private**
- Pourquoi on n’est pas permis de diminuer l’accès ?
  - Contrat : on a promis que dans tous les objets de la super-classe (→ tous les objets de la sous-classe) un champ est public.

# Redéfinition des Champs

```
class A { int x; int y ;}
class B extends A{
    double x ;//OK de changer le type
    public int y ;//OK d'augmenter l'accès
    void f(){
        x = 2.3 ;//Nouveau x. x de super-classe occulté
        super.x = 2 ;//x de la super-classe
    }
```

# Le mot clé super

- Dans une sous-classe nous permet de faire référence à un champ de la super-classe (même si occulté)
- En dehors des classes, la version correcte du champ est décidé selon le type qui précède le .

**class A { public String x;}**

**class B extends A { public double x;}**

**..**

**B b1 = new B() ; A b2 = new B() ;**

**b1.x = 2.2 ;**//OK

**b2.x = 2.2 ;**// !!!!

# Type-Casting

- Pour l’exemple du slide précédent, l’objet b2 est en réalité de type B, mais ce n’est pas connu pendant la compilation !

  Ex :

  **A b2 ; if(..) b2 = new A() ; else b2=new B() ;**

- → Logique que le compilateur n’accepte pas une ligne qui suppose que b2 est vraiment un B.
  - On peut forcer une conversion

    **((B)b2).x = 2.2**//OK mais à eviter !

- Le compilateur accepte. On a promis que, bien que b2 soit déclaré comme A, il a type B.
  - Si pas le cas → Run-time Exception !!

- Attn : On peut utiliser le type-casting que pour des classes qui ont une relation de sous-classe (même indirectement)

    **String s ; ((B)s).x=2.2 ;// !!! ???** Ne compile pas

# Addition des méthodes

- Le compilateur utilise le type déclaré pour décider si un appel est correcte.

  **class A { … } ;**

  **class B { void f(){…} } ;**

  **..**

  **B b = new B() ; A b2 = new B() ;**

  **b.f() ;**//OK

  **b2.f() ;**// !!! Type-casting ?

# Redéfinition des méthodes

- Une sous-classe peut donner une version adaptée d’une méthode héritée.

  Ex :

  ```
  class Point {
      double x,y ;
      String toString(){ return ‘’Coords =’’+x+’’,’’+y ; }
  }
  class ColorPoint extends Point {
      int color ;
      String toString() { return ‘’Coords =’’+x+’’,’’+y+..color ; }
  }
  ```

# Redéfinition des méthodes

- Une sous-classe peut donner une version adaptée d'une méthode héritée.

  Ex :

```
class Point {
    double x,y ;
    String toString(){ return ‘’Coords =’’+x+’’,’’+y ; }
}
class ColorPoint extends Point {
    int color ;
    String toString() { return super.toString()+y+..color ; }
}
```

Super nous permet d'accéder aux champs/méthodes occultés de la super-classe

# Redéfinition vs Surcharge

- Redéfinition (Over-ride) :
  - Définition dans une sous-classe d'une méthode avec la même signature d'une méthode de la super-classe
- Surcharge (Over-loading) :
  - Définition, dans une sous-classe ou la même classe, d'une méthode avec le même nom avec une méthode existante, mais autre nombre/type de paramètres
- Attn :
  - Pour la redéfinition, le type de retour **peut** changer. Il doit rester un sous-type du type de retour originale.

# Redéfinition

```
class Point {
    Point makeCopy() { return new Point(x,y) ; }
    ..
}
class ColorPoint extends Point{
    ColorPoint makeCopy() { return new ColorPoint();}
}
```

- OK, parce que ColorPoint est un sous-type de Point

# Constructeurs

- La sous-classe hérite toutes les méthodes sauf les constructeurs
- Il faut définir les constructeurs de la sous-classe
- Ceux-ci peuvent appeler les constructeurs de la super-classe avec le mot-clé **super**
  - Si on utilise **super** pour appeler le constructeur de la super-classe, il faut que ce soit la première instruction de notre constructeur
  - Si on utilise **super** on ne peut pas utiliser **this**
  - Si on n'utilise ni l'un ni l'autre, en fait, **super()** et automatiquement appelé (constructeur par défaut de la super-classe)
    - Si le constructeur par défaut n'existe pas → Erreur de compilation !

# Interdiction de Redéfinition

- On peut utiliser le mot-clé **final** pour indiquer qu’une méthode ne peut pas être redéfinie.

```
class A {
    final int i = 2 ;
    final void f() {…}
}
class B extends A {
    int i = 3 ;//OK ! Occulte super.i
    void f() {…} //Erreur de compilation !
}
```

# Polymorphisme

- Un objet de la sous-classe peut être vu comme un objet de la super-classe.
- Quand les définitions des deux classes sont différentes, quelle définition a priorité ?

  **Ex :**

  **class A{ int x ; void f() {…};}**

  **class B extends A{String x ; void f() {…};}**

  **..**

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

  **a.x = ‘’Hi’’ ;**//Erreur !

  **b.x = ‘’Hi’’ ;**//OK, décision prise selon type declaré

# Polymorphisme

- Même quand un objet de type déclaré A est en réalité un objet de type B, le compilateur le traite comme un objet du type déclaré (A).
- Seule exception (très importante!), l'appel aux méthodes de la classe
  - Si la classe B a redéfini une méthode, le programme va tenter d'exécuter la version de la méthode du **vrai type**
- On appelle ce processus linkage dynamique (dynamic binding).
  - NB : C'est une décision qui doit être prise pendant le temps d'exécution ! (Pourquoi?) → Dynamic binding a un coût en termes de temps d'exécution.

# Polymorphisme

- Un objet de la sous-classe peut être vu comme un objet de la super-classe.
- Quand les définitions des deux classes sont différentes, quelle définition a priorité ?

**Ex :**

**class A{ int x ; void f() {…};}**

**class B extends A{String x ; void f() {…};}**

**..**

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

**a.f() ;**

**b.f() ;**//appellent la même méthode B.f()!

# Polymorphisme – Exemple

```
class Point{
    double x,y ;
    String toString() { return ‘’Coords=’’+x+’’,’’+y;}
}
Point p = new Point(1.1,2.2) ;
S.o.p(p.toString()) ;//OK, évidemment
Object o = new Point(2.2,3.3) ;
S.o.p(o.toString()) ;//appelle Point.toString()
```

# Polymorphisme – Exemple

```
class Point{
    double x,y ;
    String toString() { return ‘’Coords=’’+x+’’,’’+y;}
}
class ColorPoint { int color ;
    String toString() { return super.toString()+’’,’’+color;}
}
Point p = new ColorPoint() ;
S.o.p(p.toString()) ;//appelle ColorPoint.toString
```

# Polymorphisme – Exemple

```
class Point{
    double x,y ;
    String toString() { return ‘’Coords=’’+x+’’,’’+y;}
}
class ColorPoint { int color ;
    String toString() { return super.toString()+’’,’’+color;}
}
Point [] p = new Point[2] ;
p[0] = new Point(2.2,3.3);
p[1] = new ColorPoint();
for(int i=0;i<2;i++) System.out.println(p[i]);
```

# Polymorphisme

- Intérêt : permet de traiter de manière unifiée des objets différents, en prenant automatiquement en compte leurs différences (la version de la méthode appelée est la version « correcte »).
- Désavantages :
  - Temps d'exécution
  - Lisibilité (?)
  - Pas appliquée de manière consistente
- Le linkage dynamique ne s'applique **que** pour l'appel des méthodes non-statiques
  - Pas pour les variables
  - Pas pour les méthodes statiques
  - Pas pour les paramètres
- Static Binding = Décision prise selon type déclaré (compile-time)

# Static Binding

```
class A { … }
class B extends A {
    void f(A a) {S.o.p(‘’A’’);}
    void f(B b) {S.o.p(’’B’’);}
}
..
B b = new B() ; A a = new B() ;
b.f(b) ;//B
b.f(a) ;//A
```

# instanceof

- Vous pouvez tester pendant l'exécution du programme quel est le type effectif d'un objet avec le mot clé **instanceof**

  **Point p = new ColorPoint() ;**

  **if(p instanceof ColorPoint) S.o.p(''CP'') ;**

- À éviter !
  - Normalement, c'est une meilleure idée de s'appuyer sur le linkage dynamique, si possible
- Exception :
  - Pour vérifier avant de tenter un type-casting

  **if(p instanceof ColorPoint){**

      **ColorPoint cp = (ColorPoint)p;** //conversion sûre dans ce cas

  **}**