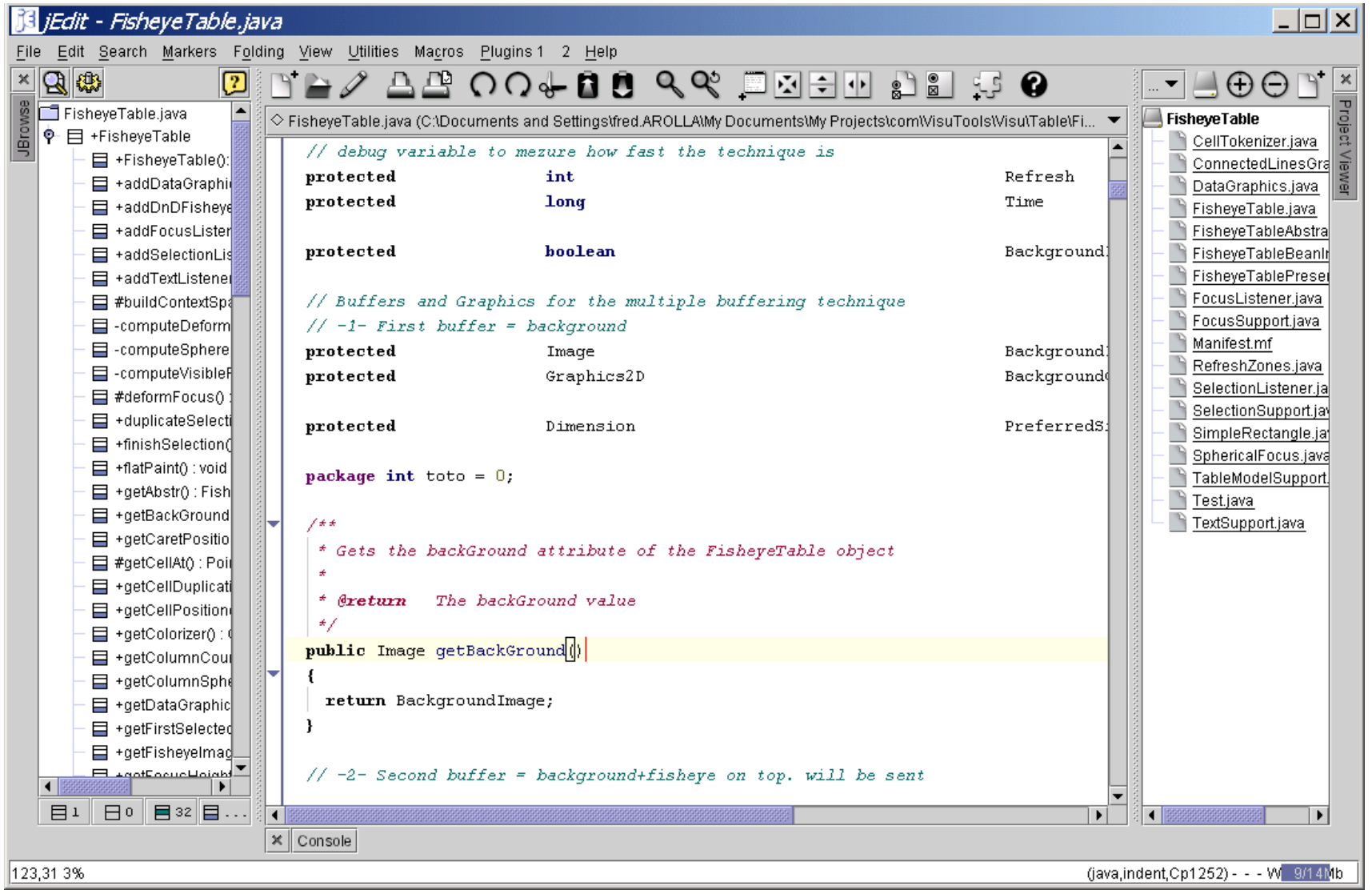
Base de programmation Objet en JAVA. 2ème partie.

Frédéric Vernier
(Université Paris-Sud / LRI / LIMSI-CNRS)
Frederic.Vernier,@limsi.fr

Ce cours reprend en grande partie le matériel pédagogique mis au point par Jérôme Nobécourt, Christian Jacquemin et Claude Barras pour l'enseignement de Java en 2001 et 2002 en FIIFO et celui trouvé sur Internet
(http://www.laltruiste.com/, etc.)

Jedit

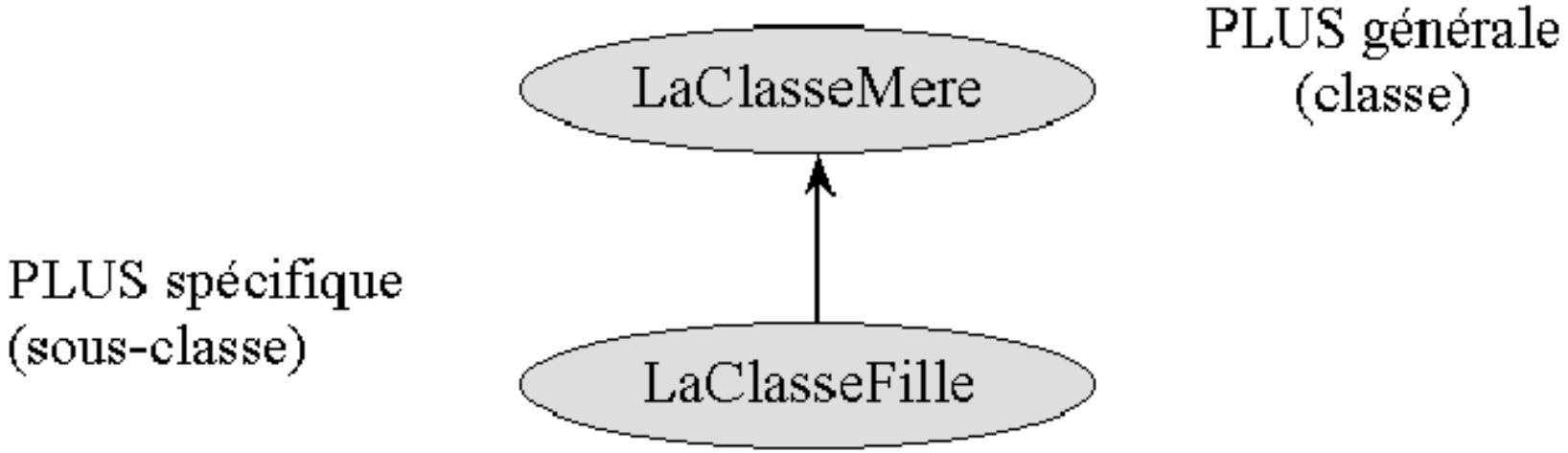

jEdit - FisheyeTable.java
File Edit Search Markers Folding View Utilities Macros Plugins 1 2 Help
FisheyeTable.java (C:\Documents and Settings\fred.AROLLA\My Documents\My Projects\com\VisuTools\Visu\Table\Fi...
// debug variable to mezure how fast the technique is
protected int Refresh
protected long Time
protected boolean Background
// Buffers and Graphics for the multiple buffering technique
// -1- First buffer = background
protected Image Background
protected Graphics2D Background
protected Dimension PreferredS
package int toto = 0;
/**
* Gets the backGround attribute of the FisheyeTable object
*
* @return The backGround value
*/
public Image getBackGround()
{
return BackgroundImage;
}
// -2- Second buffer = background+fisheye on top. will be sent
FisheyeTable
CellTokenizer.java
ConnectedLinesGra
DataGraphics.java
FisheyeTable.java
FisheyeTableAbstra
FisheyeTableBeanIr
FisheyeTablePrese
FocusListener.java
FocusSupport.java
Manifest.mf
RefreshZones.java
SelectionListener.ja
SelectionSupport.ja
SimpleRectangle.ja
SphericalFocus.java
TableModelSupport
Test.java
TextSupport.java
Console
123,31 3%
(java,indent,Cp1252) - - - W 9/14Mb

Plan 4: Héritage

1. Héritage
2. Concept
3. Hiérarchie
4. Exemple Javadoc 1&2
5. Exemple d'héritage
6. Utilisation des constructeurs
7. Constructeur & super
8. Blocage
9. Transtypage & héritage 1&2
10. Classe Object
11. Classe Class
12. Méthode equals()
13. Méthodes hashCode() & toString()
14. Classes enveloppes 1&2 &3
15. Classe Vector
16. Factorisation 1&2
17. Constructeur Père&Fils
18. Schémas d'instance
19. Schémas de classe
20. Conseils

Héritage

- class LaClasseFille extends LaClasseMere



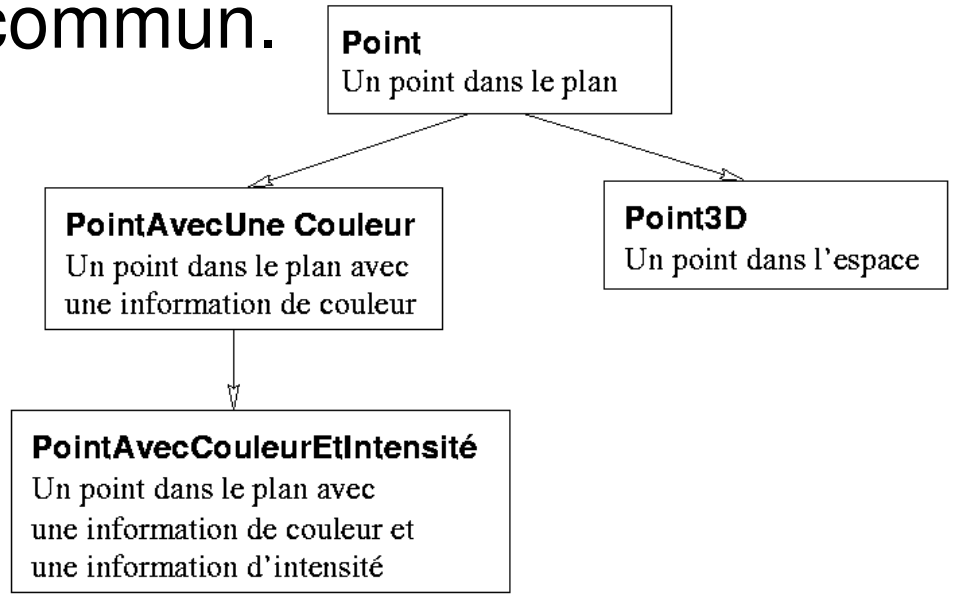
- class PointAvecUneCouleur extends Point
- java n'autorise que l'héritage simple

Concept d'héritage

- Concept de base des langages à objet
- Dériver une nouvelle classe à partir d'une classe existante en récupérant ses propriétés.
- Avantage
  - Réutilisation : factorisation, enrichissement de champs et méthodes de classes existantes.
  - Spécialisation d'une classe existante sans la modifier
- Contrainte
  - Héritage simple : une classe ne peut hériter que d'une seule classe.
  - Toutes les méthodes et champs ne sont plus regroupées dans le même fichier

Hiérarchie En Héritage

- L'héritage est une notion transitive
- Hiérarchie d'héritage : l'ensemble des classes dérivées d'un parent commun.



- Chaîne d'héritage : le chemin menant d'une classe vers ses ancêtres.

Exemple en Javadoc 1/2

javax.swing

Class AbstractButton
java.lang.Object
|
+--java.awt.Component
|
+--java.awt.Container
|
+--javax.swing.JComponent
|
+--**javax.swing.AbstractButton**

All Implemented Interfaces:
ImageObserver, ItemSelectable, MenuContainer, Serializable, SwingConstants
Direct Known Subclasses:
JButton, JMenuItem, JToggleButton

public abstract class **AbstractButton**
extends JComponent
implements ItemSelectable, SwingConstants
Defines common behaviors for buttons and menu items. For further information see How to Use Buttons, Check Boxes, and Radio Buttons, a section in *The Java* …..

Exemple en Javadoc 2/2

Class java.lang.Object
…
  class java.awt.**Component** **(implements java.awt.image.**ImageObserver, …
    class java.awt.**Container**
      class javax.swing.**CellRendererPane** **(implements javax.accessibility.**Accessible)
      class javax.swing.**JComponent** **(implements java.io.**Serializable)
        class javax.swing.**AbstractButton** **(implements java.awt.**ItemSelectable, …
          class javax.swing.**JButton** **(implements javax.accessibility.**Accessible)
          class javax.swing.**JMenuItem** **(implements javax.accessibility.**Accessible, …
            class javax.swing.**JCheckBoxMenuItem** **(implements …**
            class javax.swing.**JMenu** **(implements javax.accessibility.**Accessible, …
            class javax.swing.**JRadioButtonMenuItem** **(implements …**
          class javax.swing.**JToggleButton** **(implements javax.accessibility.**Accessible)
            class javax.swing.**JCheckBox** **(implements javax.accessibility.**Accessible)
            class javax.swing.**JRadioButton** **(implements javax.accessibility.**Accessible)
        class javax.swing.**Box** **(implements javax.accessibility.**Accessible)
        class javax.swing.**Box.Filler** (implements javax.accessibility.Accessible)
        class javax.swing.**JColorChooser** (implements javax.accessibility.Accessible)
        class javax.swing.**JComboBox** (implements javax.accessibility.Accessible, …
          …

Exemple d'héritage

```
import Point;
public class PointAvecUneCouleur extends Point
{
     int couleur;

     public PointAvecUneCouleur(int x, int y, int couleur)  // un constructeur
     {
        super();        // appel au constructeur de la classe mère
        setX(x);
        setY(y);
        setCouleur(couleur);
     }

     public PointAvecUneCouleur(int couleur)    // un constructeur
     {
        super();        // appel au constructeur de la classe mère
        setCouleur(couleur);
     }

     public void setCouleur(int couleur)
     {
        this.couleur=couleur;    // désigne l'objet encours
     }
```

Exemple d'utilisation des constructeurs

- PointAvecUneCouleur p1,p2,p3;
  - x=10, y=100, couleur=1000

```
p1=new PointAvecUneCouleur(10,100,1000);
```

  - x=y=0 (valeur par défaut), couleur=5

```
p2=new PointAvecUneCouleur(5);
```

- Mais ce qui est impossible :

```
p3=new PointAvecUneCouleur();
p4=new Point (10,100,1000);
```

UNIVERSITÉ PARIS-SUD 11

Constructeurs

- Il est préférable d'appeler le super()
  - Même s'il est vide (pour versions ultérieures)
  - Même si on ne sait pas ce qu'il y a dedans (la convention c'est de mettre juste le code indispensable dans le constructeur)
- Le super() ne s'appelle qu'au début
- Dans le doute on peut créer une/des fonction init(…)
  - appel explicite (par celui qui fait le new)
  - Appel implicite (depuis le constructeur)

Blocage de l'héritage

- Mot-clé final
  - Le mot-clé final devant une classe indique qu'elle ne peut avoir de descendance.

```
final class Point3D {
   ...
}
```

- Avantage
  - Efficacité : implémentation statique possible.
  - Sécurité : pas de redéfinition possible des méthodes pour une descendance.
- Inconvénients
  - Impossibilité de sous-classer pour créer une classe de test indépendante

Transtypage et héritage 1/2

- Le transtypage consiste à convertir un objet d'une classe en objet d'une autre classe
- Vers super-classes toujours possible : on peut toujours transtyper vers une super-classe, plus pauvre en informations

```
Point3D unPoint3D = new Point3D( x, y, z );
Point unPoint = (Point) unPoint3D; // revient à une
   projection sur xOy
```

  – La mention du transtypage est facultative, mais recommandée pour la lisibilité.

Transtypage et héritage 2/2

- Vers sous-classes
  - Impossible sauf si appartenance : pour transtyper vers une sous-classe, utiliser le mot-clé réservé instanceof.

```
if( unPoint instanceof Point3D) {
      unPoint3D = (Point3D) unPoint;
      unPoint3D.projection3D(...);
}
```

- La mention du transtypage est obligatoire, dans ce cas.
- Ne pas transtyper systématiquement quand le compilateur retourne une erreur " bad class "
  - Réfléchir a l'erreur…
  - Ne devrait-on pas demander en argument un type plus précis ?
  - Dois-je tester le type avant de transtyper ?

La classe "Object"

- Toutes les classes Java sont des descendants de la classe Object.
- Object est parent par défaut des classes qui ne précisent pas leur ancêtres
- Object offre quelques services génériques

```
public final Class getClass();
```

- Renvoie un objet de type Class qui permet de connaître le nom d'une classe

```
public boolean equals(Object o);
```

- Compare les champs un par un (pas les pointeurs mémoire comme ==)

```
protected Object clone();
```

- Allocation de mémoire pour une nouvelle instance d'un objet et recopie de son contenu

```
public String toString();
```

- Renvoie une chaîne décrivant la valeur de l'objet

La classe "Class"

- C'est la classe qui identifie les types à l'exécution
- Permet la sélection des méthodes en fonction de la classe lors des appels

```
public String getName();
```

- Renvoie le nom d'une classe

```
public static Class forName(String s);
```

- La fonction réciproque de getName

```
public Object newInstance();
```

- Crée une nouvelle instance de même type

```
public String toString();
```

- Renvoie une chaîne décrivant la valeur de l'objet

La fonction equals()

- Très utile pour les String
  - `If (myString == "Hello") {…} Toujours Faux !!!`
  - `If (myString.equals("Hello") ) {…} OK`
- Si les 2 objets o1 et o2 ont des références r1 et r2 comme champs, il faut que r1==r2 pour que o1.equals(o2) et non pas que r1.equals(r2)
- Réflexive, symétrique, transitive et consistent
- null.equals(null) mais aucune autre instance oi telle que oi.equals(null) if (!myString.equals(null))

Les fonctions hashCode() et toString()

- hashCode() renvois un chiffre différent pour toute instance différente

```
Si o1.equals(o2) alors o1.hashCode()==o2.hashCode()
```

  - Le contraire n'est pas assuré mais préférable (@mémoire)

- toString() donne une version "lisible" de la Class et du hashcode

```
getClass().getName()+'@'+Integer.toHexString(hashCode())
```

  - Nom unique utile pour tracer la vie d'une instance

Classes enveloppes 1/3

- Les classes enveloppes fournissent une contrepartie sous forme de classes aux types primitifs

- Les classes enveloppes sont final, donc non surchargeables

Nombres entiers	int	Integer
Nombres entiers longs	long	Long
Nombres décimaux	float	Float
Nombres décimaux en double précision	double	Double
Booléens	boolean	Boolean
Octets	byte	Byte
Caractères (Unicode)	char	Character
Type vide	void	Void

Classes enveloppes 2/3

- Quelques méthodes de la classe Integer par exemple :

```
public int intValue();
```

- Renvoie la valeur entière de l'objet (type int).

```
public static int parseInt(String s);
```

- Renvoie la valeur entière de l'objet si l'objet spécifié représente la chaîne d'un nombre entier en base 10.

```
public static Integer valueOf(String s)
```

- Renvoie un objet Integer si la chaîne s représente un nombre entier en base 10.

- Exemple :

```
Integer unInteger = new Integer(100);
int d = unInteger.intValue();
```

Classes enveloppes 3/3

- Pas de modificateur
  - Pas de Float F1 = new Float();
    car pas possible de le changer ultérieurement
  - Fonction equals() pour comparer
  - Beaucoup de fonctions statiques (pas besoin de faire de new pour les utiliser)
  - Float, Byte, Integer peuvent tous retourner une valeur int, float, etc. car héritage de Number
- Comparable
  - Fonction compareTo()

La classe Vector

- Une liste indexée
  - Comme un tableau de Object mais de taille variable
  - Gère sa capacité d'accueil en fonction d'une stratégie d'optimisation de la mémoire
  - addElement(…), removeElement(..), removeAllElement(), elementAt(int i), size(), toArray()
  - Accepte des instances de n'importe quel type
  - Retourne des Objects a vous de transtyper !
  - elements() permet retourner une Enumeration

Enumeration

- Permet d'énumérer des éléments et de faire des boucles avec une notation simple
- Sun dit …

```
for (Enumeration e = v.elements() ; e.hasMoreElements() ;) {
   System.out.println(e.nextElement());
}
```

- Fred préférerait …

```
for (Enumeration Enumeration1 = Vector1.elements() ; Enumeration1
   .hasMoreElements() ;) {

  MaClasse MonInstance = (MaClasse)Enumeration1.nextElement();
  MonInstance.maFonction(...);
}
```

Exemple présidentiel 1/2

```
public class USA {
  protected    Homme          president    = null;
  private      Vector         etats        = new Vector();
  public final int            NB_ETATS     = 50;

  public static void main(String[] arg) {
    USA TheUSA = new USA();
  }

  public USA(){
    etats.add(new Population("Alabama" , 4319000));
    etats.add(new Population("Alaska" , 609000));
    etats.add(new Population("Arizona" , 4555000));
    etats.add(new Population("Arkansas" , 2523000));
    etats.add(new Population("California" , 32268000));
  }
}
```

Exemple présidentiel 2/2

```
  // ne doit etre appelle que par l'objet AssembleeNationale
  public elirePres(Homme Cand1_Arg, Homme Cand1_Arg)
  {
    int Resultat1, Resultat2 = 0;

   for (Enumeration enum = etats.elements(); enum.hasMoreElements();) {
      Population Population1 = (Population)enum.nextElement());
      int Choix1= Population1.choose(Cand1_Arg,Cand2_Arg);
      Resultat1+=Choix1;
      Resultat2+=(Population1.getVotants()-Choix1);
    }

    if (Resultat1>=Resultat2)
      setPresident(Cand1_Arg);
    else
      setPresident(Cand2_Arg);
  }
}// Fin class USA
```

Exemple de factorisation 1/2

- Lors de la factorisation de 2 classes,
  - D'autres classes pourront s'insérer dans votre nouveau schéma
  - Renommer les fonctions en terme plus générique

```
Public class Bouton {
  public void setActionName(String ...)
  {...}
}

Public class Label {
  public void setLabelText(String ...)
  {...}
}
```

Exemple de factorisation 2/2

```
Public class TextComponent {
  public void setText(String ...)
  {...}
  public void paint () {}
}
```

Taille, couleur, police, etc.

```
Public class Bouton extends TextComponent {
  public void paint () {...}
}
```

Action souris, état, bordure, ombrage, etc.

```
Public class Label extends TextComponent {
   public void paint () {...}
}
```

Multi-ligne ???

- Autre solution : Bouton hérite de Label
  - Label factorise et propose des valeurs par défaut

Constructeur et père-fils 1/3

- Le construit n'est pas le fils du celui qui le construit : Les instances ne sont pas père-fils (seule une classe peut être père de ...)

```
Public class Travailleur {
  Object Createur = null;
  public Travailleur (Object Createur_Arg) {
    Createur = Createur_Arg;
  }
  public void faireBoulot () {
    if (Createur instanceOf CreateurAttentif)
      ((CreateurAttentif)Createur).informeBoulot(...);
  }
}
```

- Apres construction, 2 instances sont égales

Constructeur et père-fils 2/3

- "Qui créer qui" est un problème indépendant de qui est fils de qui
- Il est plus facile de garder une référence sur sa création que sur son créateur
- Mais on peut passer un pointeur de soi-même au constructeur de sa création

```
Public class DonneurDOrdre {
  Travailleur Travailleur1 = null;
  public void faireBoulot () {
    ...
    Travailleur1 = new Travailleur(this);
  }
}
```

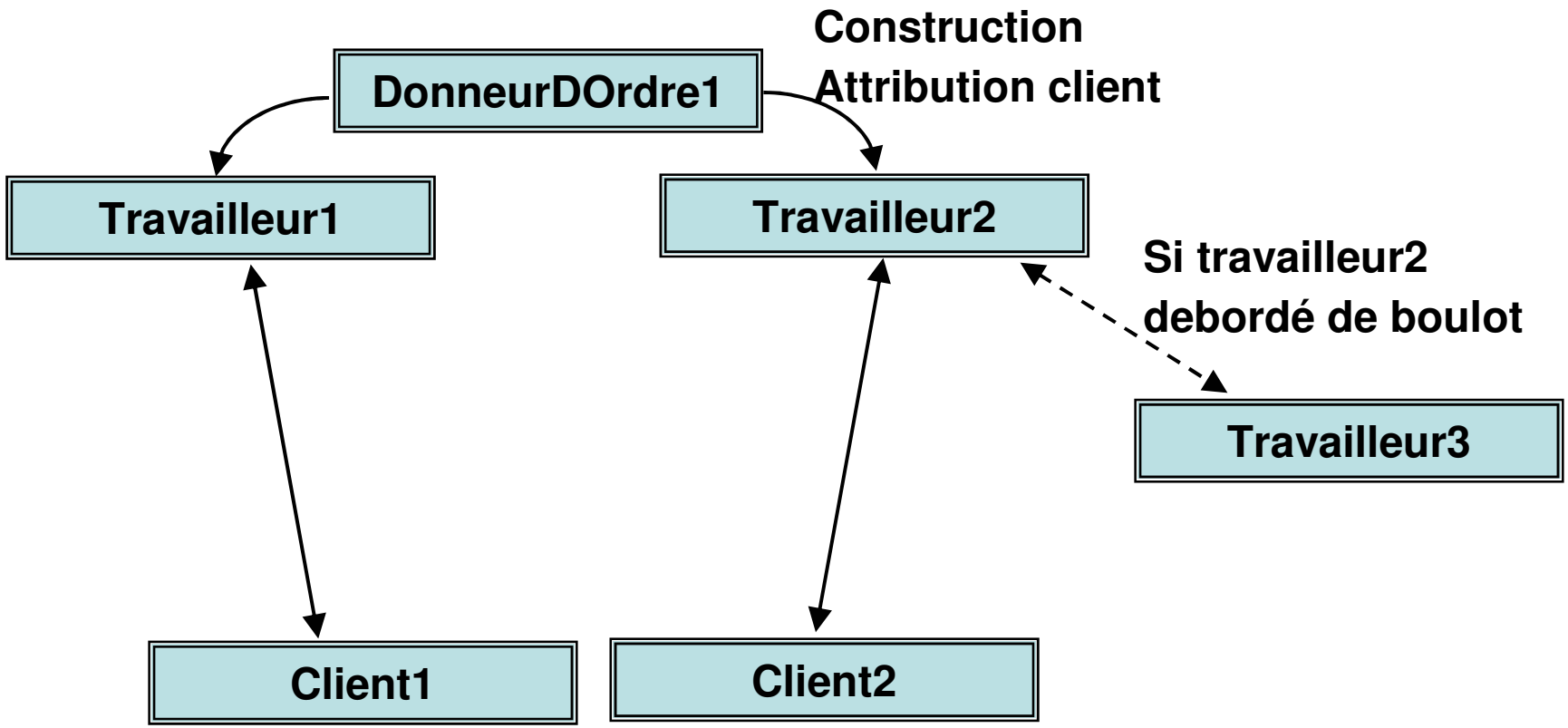
Constructeur et père-fils 3/3

- Mais la création peut faire l'hypothèse du type (de la classe) de qui la créée

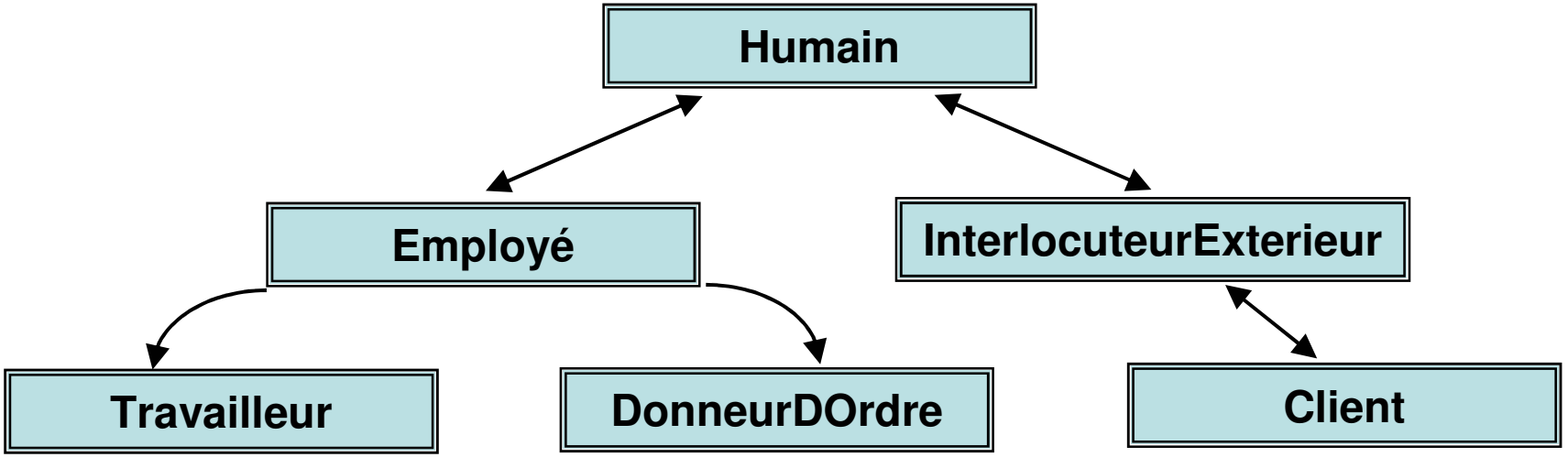
```
Public class Travailleur {
  DonneurDOrdre DonneurDOrdre1 = null;
  public Travailleur (DonneurDOrdre DonneurDOrdre_Arg) {
    DonneurDOrdre1 = DonneurDOrdre_Arg;
  }
  public void faireBoulot () {
    DonneurDOrdre1.informeBoulot(...);
  }
}
```

- Le travailleur et le donneur d'ordre bossent ensemble
  - La question de l'initialisation (qui créer qui) est secondaire

Schémas d'instance



Schémas de classe



Conseils pour toucher votre héritage

- Placer les informations communes dans la superclasse : principe d'économie et de non redondance de l'information

- N'utiliser l'héritage que lorsque c'est pertinent : c'est-à-dire lorsque toutes les informations de la surclasse sont bien utilisées par une sous-classe

- Utiliser le polymorphisme plutôt que des tests sur le type : éviter un code tel que :

```
if( x instanceof Classe1 ) {
    x.methode1();
}
else if( x instanceof Classe2 ) {
    x.methode2();
}
```

Plan 5: Héritage 2 : le retour

1. Paquetage 1&2
2. Utilisation
3. Exemple : Point / unPoint
4. Conseils
5. Membres et modificateurs
6. Droit d'accès
7. Tableau d'accessibilité
8. Polymorphisme
9. Algorithme
10. Polymorphisme(re)
11. Exemple
12. Polymorphisme de classe
13. Polymorphisme super
14. Polymorphisme contraint
15. Classe abstraite 1 et 2
16. Interface 1 et 2
17. Exemple
18. Jar
19. Jar et manifest
20. Jar suite et fin

Paquetage (1/2)

- Pour regrouper des classes " proches "
- Prédéfinis :
  - java (rangement des classes essentielles à JAVA)
  - javax (rangement des classes étendues de JAVA)
  - sun (rangement des classes propriétaires de SUN)
  - ...
- Classes Fondamentales : java.lang
- Classes des I/O (entrées-sorties) : java.io
- Sous-paquetages possibles
  - java.util.zip
- paquetages industriels
  - com.nom_de_domaine_internet.nom_de_projet …
- En minuscule

Paquetage (2/2)

- Définition d'un paquetage
  - au début du fichier .java
  - avant les `import` et le commentaire de classe
  - package NomPackage.NomSousPackage;

```
 //fichier Toto.java
package mesPackages.excours.bidon;

public class Toto{
  ...
};
```

Attention au répertoire !

- Pour lancer la commande se trouver dans le répertoire au dessus de mespackages/excours/bidon/
- Créez des scripts (Unix ou dos) au bon endroit

- Compilation (sous windows / devient \ ):
  - Toto.java est dans MesPackages/ExCours/Bidon/
  - javac MesPackages/ExCours/Bidon/Toto.java
- Exécution :
  - java MesPackages.ExCours.Bidon.Toto

Utilisation : classes et paquetages

- Une classe appartient à UN paquetage
  - Par défaut : le répertoire courant
- Une classe a le droit d'accéder aux classes du même paquetage (faire des news, appeler fcts de classes)
- Une classe a toujours 2 noms
  - nom court : NomDeLaClasse
  - nom long : NomPackage.NomDeLaClasse (à utiliser en cas d'ambiguïté)
- import permet de spécifier le chemin des classes utilisées par rapport aux paquetages
  - import java.io.File;
  - import java.io.*;

Exemple : Classe Point

```
package fr.u-psud;

public class Point                          // une classe
{
  int x;                                    // variable d'instance
  int y;                                    // variable d'instance

  public int getX() {                       // accès
    return x;
  }

  public void setX(int abscisse) {          // altération
    x = abscisse;
  }

  public int getY() {
    return y;
  }

  public void setY(int ordonnee) {
    y = ordonnee;
  }
}
```

Exemple d'instance : unPoint

```
import java.awt.*;          // on a besoin de la classe
   Point

public class TestPoints {
  public static void main(String[] arg) {
    Point unPointAWT = new Point(10, 15);

    fr.u-psud.Point unPoint=new fr.u-psud.Point();
    unPoint.setX(10);
    unPoint.setY(100);
    System.out.println("point\n\t d'abscisse " +
                       unPoint.getX() +
                       "\n"+"\t et d'ordonnée "+
                       unPoint.getY());
  }
}// Fin class TestPoints
```

Paquetage : conseils

- Il n'y a pas d'inclusions récursives
  - N'hésitez pas à importer des paquetages entiers
  - import java.awt.* n'importe pas java.awt.event.*
  - 2 fois le même import ne gène pas
- Rangez dans des paquetages : Tout (gros projets) ou Rien (petit projet)
- Si vous avez besoin de gérer dynamiquement vos paquetages

```
Package Package1 = Class1.getPackage();
Class1.getName()...
```

Membres et modificateurs

- Les variables (attributs/champs) et les méthodes = membres d'une classe
- Le modificateur se met avant le type du membre
- Ne rien mettre : il faut une instance de la classe pour pouvoir l'utiliser
  - Utilisation de new AVANT
- **static** :
  - Variable : variable de classe partagée par toutes les instances de la classe (ex : compteur)
  - Méthode : une instance n'est pas nécessaire pour l'utiliser ex:main(String[] args)
- **final** :
  - classe : interdit la définition de sous-classes
  - variable et méthode : interdit les redéfinitions dans les sous-classes
- **static final** : combine les deux
  - variable : création d'une constante

```
static final boolean EN_MAJUSCULE = true;
```

Droit d'accès

- Il faut toujours préciser les droits d'accès aux:
  - variables d'instance
  - variables de classe
  - méthodes
  - classes
- **public** : tout le monde utilise (faire des nez, app fcts)
- **protected** : les membres d'une même famille (héritage, paquetage) ont le droit d'utilisation
- **private** : c'est personnel. PAS TOUCHE !
- (**package**) Ne rien mettre : les classes du même paquetage peuvent l'utiliser
- L'accès à un paquetage dépend des protections du système

UNIVERSITÉ PARIS-SUD 11

Tableau d'accessibilité

Accessible à	Visibilité du membre			
	public	protected	*paquetage*	private
Classe de définition	OUI	OUI	OUI	OUI
Classe du même paquetage	OUI	OUI	OUI	NON
Sous-classe	OUI	OUI *	NON	NON
Autre classe	OUI	NON	NON	NON

*** Si et seulement si la sous-classe est dans le même package !**

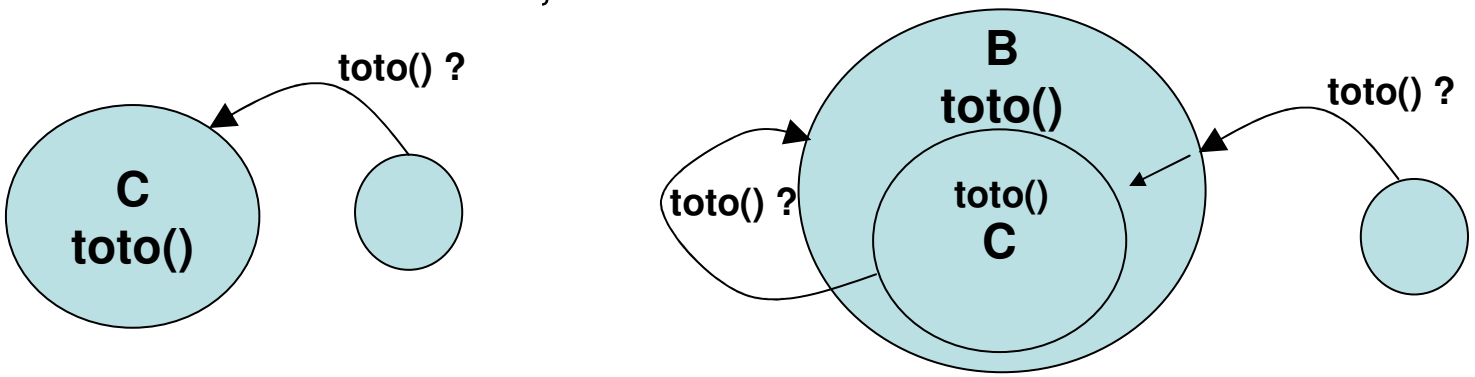
Polymorphisme

- La signature d'une méthode correspond au nombre et au type de ses paramètres plus le type retourné par la méthode
- Des classes différentes peuvent avoir des méthodes différentes de même nom et de même signature
- Le polymorphisme consiste pour *une instance* à retrouver quelle méthode il doit appeler

```
Point3D unPoint3D = new Point3D( x, y, z );
Point unPoint     = (Point) unPoint3D;
unPoint.projette(2, 5); // Point3D.projette() !
```

Algorithme

- Algorithme de recherche pour une classe C:
  - Soit MyRef une référence à une instance construite
    - B MyRef = new C(...);
  - si C a été déclarée avec une méthode de même nom et de même signature
  - alors appeler cette méthode
  - sinon si C est une sous classe de B
  - alors appliquer l'algorithme de recherche pour la classe B
  - sinon " Erreur, méthode non trouvée



Polymorphisme (re)

- Si on définit dans une classe C (sous-classe de B) une méthode m et avec la même signature que la méthode m de la classe B alors cette méthode m de la classe B est masquée pour toutes les instances de C
- On ne peut pas appeler directement la méthode m de B à partir d'une référence de C
- Exemple :

```
Public class C extends B {...}

C unC = new C();
B unB = unC; // Transtypage
unB.m(); //c'est la méthode m() de C si elle existe sinon de B
```

Exemple

- Je connais Robert, un agriculteur
- Je demande à Robert de préparer son Champ et j'attend en retour le temps que ça prend (temps = largueur du tracteur *vitesse /surface, etc.)
- Or en fait ce que je ne sais pas c'est que Robert est Viticulteur (un genre un peu spécial d'agriculteur) Pour lui préparer = tailler
- Que dois me répondre Robert ?
  - Robert sait-il que je ne sais pas qu'il est viticulteur ?
  - Dois-je faire une différence entre les viticulteurs et les autres ?

Polymorphisme de classe

- Les méthodes de classe (static) ne peuvent pas être masquées
- Exemple :

```
C unC=new C();
A unA=unC;
unA.methodeDeClasse(); // c'est la méthode de classe A qui est appelée
```

- Remarque : il est très fortement conseillé de toujours appeler les méthodes de classe par le nom de la classe plutôt que par une instance de la classe

```
A.methodeDeClasse();
```

Polymorphisme super

- On peut appeler dans la méthode m (que redéfinit la classe C) la méthode m de même signature immédiatement supérieure en la préfixant par le mot clef super
- Exemple :
  - dans un constructeur : super();
  - dans la méthode m redéfinie (ou ailleurs) : super.m();

```
private void faitBoulot(int mesure_Arg) {
  if (mesure_Arg>0)
    super.faitBoulot(mesure_Arg);
}
private float faitDecoupe(int mesure_Arg) {
  if (mesure_Arg>0)
    return super.faitDecoupe(mesure_Arg);
  else
    return super.prepareDecoupe();
}
```

- La notation super.super.m() n'est pas permise

Polymorphisme contraint

- On ne peut pas surcharger une méthode par une méthode (qui a donc la même signature) plus restreinte mais ou peut être plus laxiste

```
Public class Pere {
  protected void faitBoulot(int mesure_Arg) {
    if (mesure_Arg>0)
      super.faitBoulot(mesure_Arg);
  }
}

Public class Fils extends Pere {
  public void faitBoulot(int mesure_Arg) {
    if (mesure_Arg>0)
      super.faitBoulot(mesure_Arg);
    else
      ...
  }
}
```

Attention aux droits

- public est plus laxiste que protected
- Le fils peut appeler son père car protected le lui permet, private non !

Classe abstraite 1/2

- Une classe est dite abstraite si elle n'est pas entièrement implémentée
  - Ses sous classes doivent terminer son implémentation
  - Si une classe a (au moins) une méthode " abstract " alors cette classe est abstraite
  - Une classe abstraite ne peut pas être instanciée (new)
  - Une classe abstraite peut avoir des données
  - Pas de constructeur généralement
- Si une sous classe d'une classe abstraite n'implémente pas toutes les méthodes " abstract " de sa classe mère, c'est aussi une classe abstraite

Classe abstraite 2/2

- Les méthodes dites " static ", " private " et " final " ne peuvent pas être abstraites
  - Static : méthode de classe, or une classe abstraite n'est pas directement utilisable (partiellement implémentée).
  - Private : méthode interne à cette classe, non héritable, donc on ne peut pas la redéfinir.
  - Final : aucune surcharge de la méthode possible.
- Une classe déclarée final ne peut être abstraite
- Une classe déclarée abstraite est abstraite même si aucune méthode abstraite n'y figure
- Une classe déclarée abstraite qui ne déclare aucune méthode ni aucune donnée sert de classe de marquage. On peut ainsi déclarer des pointeurs compatibles avec toutes ses sous classes.

Interface 1/2

- Une interface déclare les signatures de méthodes " public " sans en donner une implémentation.
  - Une interface se déclare comme une classe mais avec le mot clef interface.

```
(public) interface InterfaceExemple {...}
```

  - Une interface n'a pas de variable.
  - Une interface correspond à une liste de services.
  - La signature d'une méthode est suivie d'aucun code.

```
(public) TypeRetour nomMethode(type arg) ;
```

  - Une interface peut spécialiser une autre interface en lui ajoutant des services. On peut ainsi faire des hiérarchies d'interface.

```
interface Interf1 extends Interf0;
```

- A une interface contenant plusieurs fonctions peut être associe un Adaptateur, classe implémentant les fonctions de l'interface par des fonctions vides

Interface 2/2

- Une classe peut utiliser 0 ou plusieurs interfaces.

```
public class A extends B implements NomI1, NomI2 ...
```

- Une classe qui déclare répondre aux services d'une interface (`implements NomI`)
  - DOIT donner une implémentation pour tous ces services
- Une classe abstraite peut donner une implémentation pour 0 ou plusieurs méthodes d'une interface.
  - Les autres méthodes sont déclarées par défaut comme abstraites

Exemple

```
interface I1 {  // Déclaration d'interface
      public void m1();              // Service
}

interface I2 {   // Déclaration d'interface
      public void m2();              // Service
      public int m1(String[] args); // Service
}

abstract class B {   // Une classe abstraite
      int v=0;
      public abstract void m();
}

class A extends B implements I1, I2 {
// Implémentation des services de I1 et I2 et de la méthode abstraite de B.
      public void m() {System.out.println(v);}
      public void m1() {System.out.println("1");}
      public void m2() {System.out.println("2");}
      public int  m1(String[] args) {return args.length;}
}
```

UNIVERSITÉ PARIS-SUD 11

Jar 1/3

- Un fichier JAR est un conteneur de fichiers (comme tar ou zip) pour stocker et regrouper les fichier .class
  - contient en plus un fichier nommé META-INF/MANIFEST.MF
  - peut être compressé ou non
  - Syntaxe de «tar» plus le m pour le fichier manifest.mf
  - création : `jar cmf toto.jar mymanifest.mf *.class`
  - mise à jour : `jar uf toto.jar *.java`
  - listing du contenu : `jar tf toto.jar`
  - v=verbose, 0=pas de compression
- Chemins des fichiers relatifs
  - pas de c:\blah ou
  - /usr/var/blah
  - Un fichier jar est comme un autre disque dur avec sa propre racine

Jar 2/3 Manifest

- Le fichier manifest.mf est un fichier texte qui peut contenir des meta-informations
  - `Author: prenom nom`
  - `Class-Path : MonAutreLibrairie.jar`
  - `version: 1.0`
  - `Main-Class: classname`
- Winzip lit les jars (mais ne rajoute pas des fichiers avec un chemin relatif facilement)
- Exécution = double click ou Tag applet : Main-Class
- Un jar peut être signé pour authentifier l'origine de son contenu (voir doc. SUN)

Jar 3/3

- Un jar peut contenir les sources (java), les autres ressources (gif, jpg, html, etc.)
- Les fichiers à l'intérieur du jar sont accessibles depuis le code java par

```
java.net.url URL1 =
   getClass().getResource("/Images/PleaseWait.jpg");
```

- A partir de cette URL on peut par exemple récupérer une image

```
if (URL1!=null)
   ImagePleaseWait = (new ImageIcon(URL1)).getImage();
```