Héritage (2)

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille1

Université Lille1 Sciences et Technologies | IEEA - FIL Informatique

Appels ?

```
public class C { public void f() { System.out.println("f.c"); } }
public class A extends C { public void f() { System.out.println("f.a"); } }
public class B extends C {}
public class TestAppelMethodeEtSousClasses {
   public void appel(C c) { this.m(c); }
   public void m(C c) { c.f(); }
   public void m(B b) { System.out.println("appel avec type B"); }
   public void m(A a) { System.out.println("appel avec type A"); }
   public static void main(String[] args) {
      C c = new C(); A a = new A(); B b = new B();
      TestAppelMethodeEtSousClasses t = new TestAppelMethodeEtSousClasses();


      System.out.println("--------------");
      t.m(c); t.m(a); t.m(b);
      System.out.println("--------------");
      t.appel(c); t.appel(a); t.appel(b);
   }
}// TestAppelMethodeEtSousClasses
```

Appels ?

```
public class C { public void f() { System.out.println("f.c"); } }
public class A extends C { public void f() { System.out.println("f.a"); } }
public class B extends C {}
public class TestAppelMethodeEtSousClasses {
   public void appel(C c) { this.m(c); }
   public void m(C c) { c.f(); }
   public void m(B b) { System.out.println("appel avec type B"); }
   public void m(A a) { System.out.println("appel avec type A"); }
   public static void main(String[] args) {
      C c = new C(); A a = new A(); B b = new B();
      TestAppelMethodeEtSousClasses t = new TestAppelMethodeEtSousClasses();


      System.out.println("--------------");
      t.m(c); t.m(a); t.m(b);
      System.out.println("--------------");
      t.appel(c); t.appel(a); t.appel(b);
   }
}// TestAppelMethodeEtSousClasses
```

```
+----------------
| f.c
| appel avec type A
| appel avec type B
|----------------
| f.c
| f.a
| f.c
```

Appels ?

```
public class C { public void f() { System.out.println("f.c"); } }
public class A extends C { public void f() { System.out.println("f.a"); } }
public class B extends C {}
public class TestAppelMethodeEtSousClasses {
   public void appel(C c) { this.m(c); }
   public void m(C c) { c.f(); }
   public void m(B b) { System.out.println("appel avec type B"); }
   public void m(A a) { System.out.println("appel avec type A"); }
   public static void main(String[] args) {
      C c = new C(); A a = new A(); B b = new B();
      TestAppelMethodeEtSousClasses t = new TestAppelMethodeEtSousClasses();


      System.out.println("--------------");
      t.m(c); t.m(a); t.m(b);
      System.out.println("--------------");
      t.appel(c); t.appel(a); t.appel(b);
   }
}// TestAppelMethodeEtSousClasses
```

```
+----------------
| f.c
| appel avec type A
| appel avec type B
|----------------
| f.c
| f.a
| f.c
```

Pour les paramètres des méthodes, c'est le type de la référence qui compte.

Créer une classe d'exception

- dans la mesure du possible utiliser les exceptions existantes, sinon

1. définir une classe en lui donnant un nom explicite de la forme `QuelqueChoseException`
2. la faire hériter de `Exception` ou de l'une de ses sous-classes déjà définies

```
public class MatiereNotFoundException extends Exception {
   public MatiereNotFoundException(String msg) {
      super(msg):
   }
}
```

- les exceptions qui héritent de `RuntimeException` n'ont pas besoin d'être obligatoirement capturées

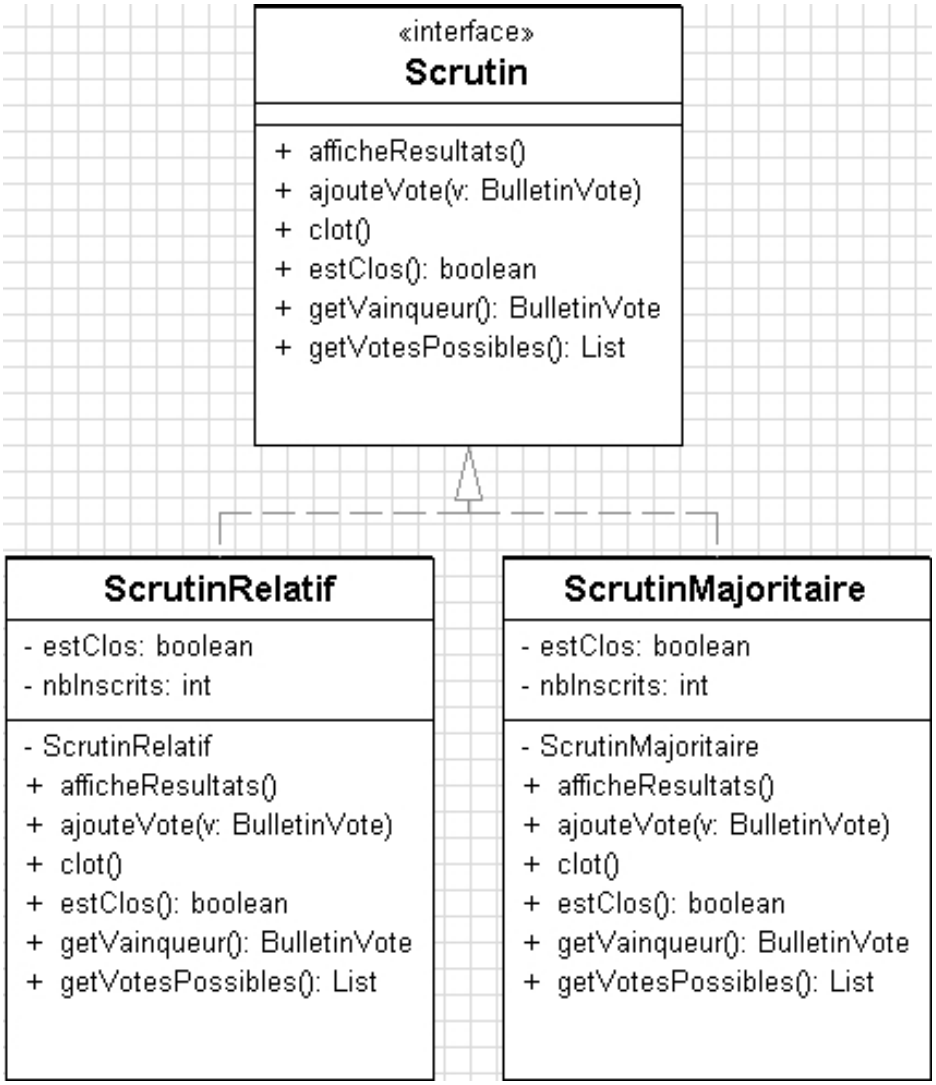
Méthodes surchargées et exceptions

Lors de la surcharge d'une méthode d'une super-classe, la signature doit être rigoureusement la même, **jusqu'aux** exceptions.
Avec cependant la possibilité d'affiner les exceptions levées par la méthodes par des sous-classes des exceptions originales.

```
public class ImmangeableException extends Exception { ...  }
public class Animal {
   public void mange(Nourriture n) throws ImmangeableException { ...  }
}
public class ViandeImmangeableException extends ImmangeableException { ...}
public class Herbivore extends Animal {
   public void mange(Nourriture n) throws ViandeImmangeableException { ...}
}
   // ...  utilisation ...
Animal animal = new Animal();
try {
   animal.mange();
}
catch(ViandeImmangeableException e) { ...  }
catch(ImmangeableException e) { ...  }
```

Attention à l'ordre de capture des exceptions

Scrutins


«interface»
Scrutin
+ afficheResultats()
+ ajouteVote(v: BulletinVote)
+ clot()
+ estClos(): boolean
+ getVainqueur(): BulletinVote
+ getVotesPossibles(): List
ScrutinRelatif
- estClos: boolean
- nbInscrits: int
- ScrutinRelatif
+ afficheResultats()
+ ajouteVote(v: BulletinVote)
+ clot()
+ estClos(): boolean
+ getVainqueur(): BulletinVote
+ getVotesPossibles(): List
ScrutinMajoritaire
- estClos: boolean
- nbInscrits: int
- ScrutinMajoritaire
+ afficheResultats()
+ ajouteVote(v: BulletinVote)
+ clot()
+ estClos(): boolean
+ getVainqueur(): BulletinVote
+ getVotesPossibles(): List

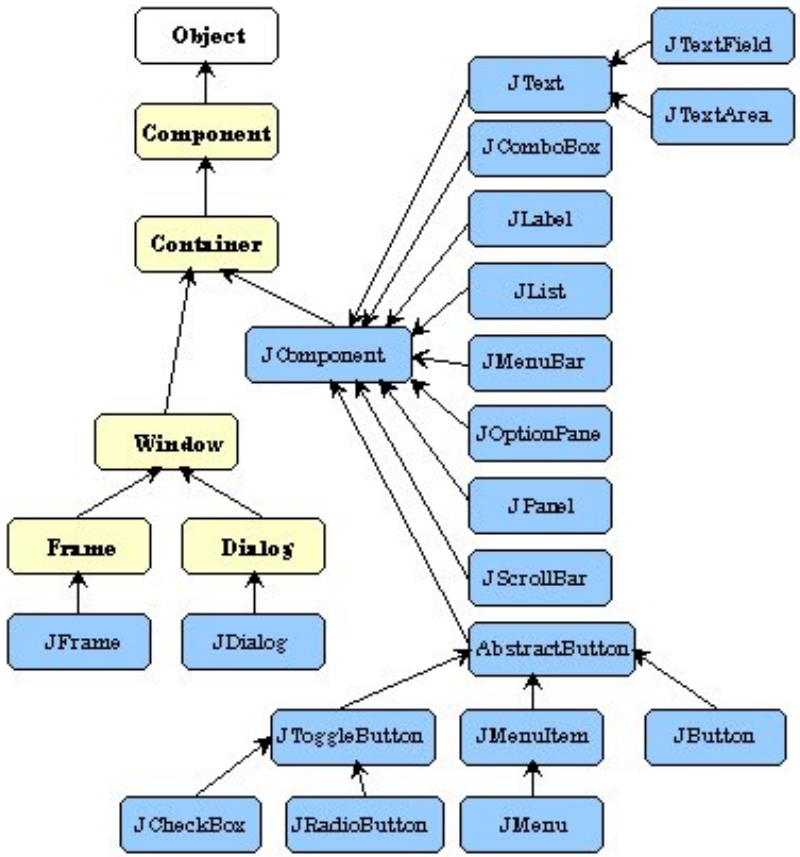
Comptes bancaires

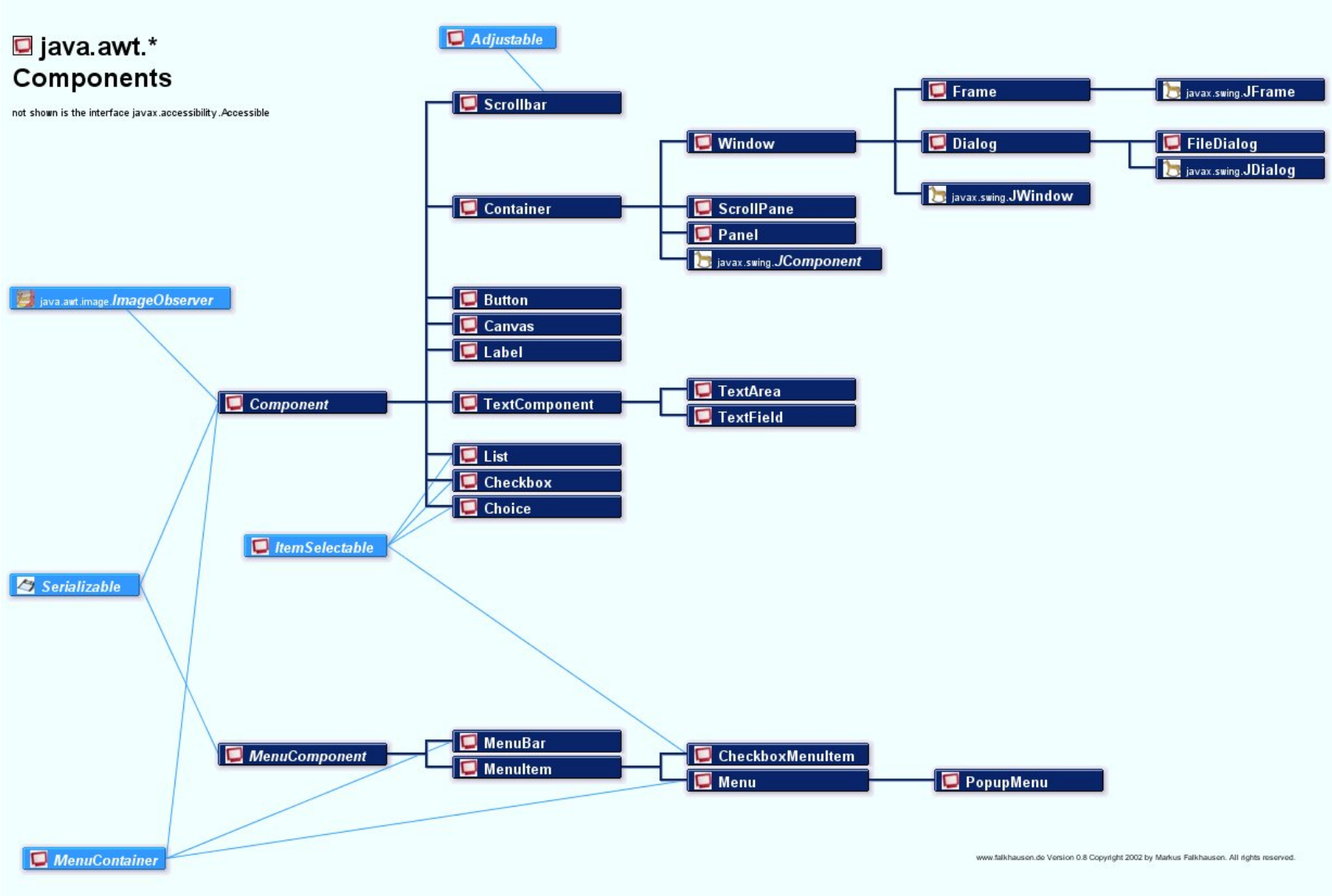
```
...
public class Compte {
   protected String numeroCompte;
   protected float solde;
   protected List<Ecriture> ecritures;
   public Compte(String numeroCompte) {
      this.numeroCompte = numeroCompte;
      this.solde = 0;
      this.ecritures = new ArrayList<Ecriture>();
   }
   public String getNumeroCompte() { return numeroCompte; }
   public float getSolde() { return solde; }
   public List getEcritures() { return ecritures; }
   public void addEcriture(Ecriture e) throws UnsupportedOperationException { ...  }
   public List<Ecriture> ecrituresDepuis(util.date.Date d, int nbJours) { ...  }
}

public class CompteCheque extends Compte {
   private float decouvertAutorise;
   public float getDecouvertAutorise() { return this.decouvertAutorise; }
   public void setDecouvertAutorise(float val) { this.decouvertAutorise = val; }
   public CompteCheque(String numeroCompte) {
      super(numeroCompte);
      this.decouvertAutorise = 0;
   }
}
public class CompteEpargne extends Compte { ...  }
```

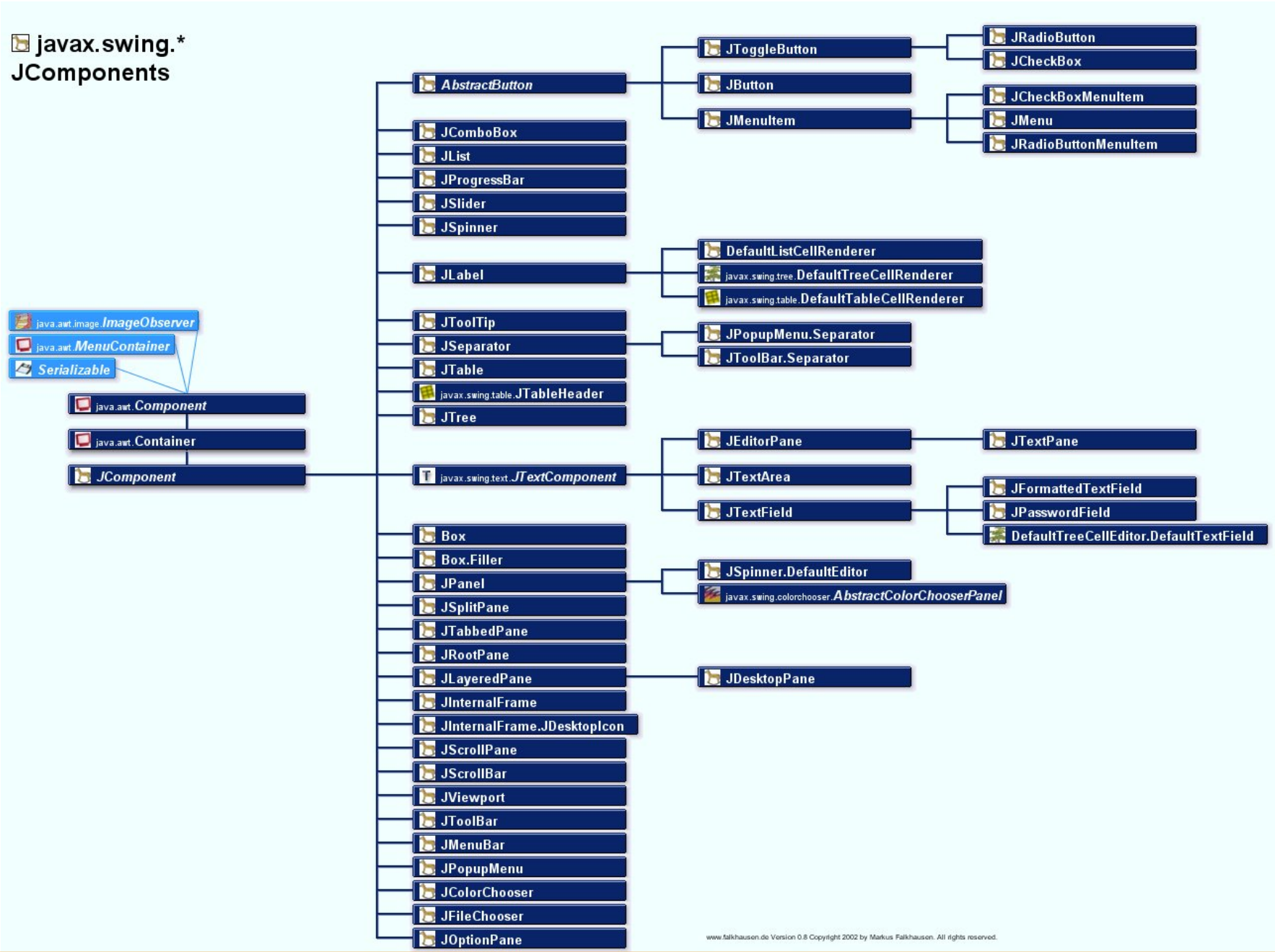
Hiérarchies de composants graphiques

`java.awt` et `javax.swing`

- factorisation des comportements et propriétés des différents éléments
  ↪ position, taille, visibilité, gestion des événements, etc.
- polymorphisme
  ↪ tout les composants swing sont du type “JComponent”




java.awt.*
Components
not shown is the interface javax.accessibility.Accessible
Adjustable
Scrollbar
Window
Frame
javax.swing.JFrame
Dialog
FileDialog
javax.swing.JDialog
javax.swing.JWindow
Container
ScrollPane
Panel
javax.swing.JComponent
java.awt.image.ImageObserver
Button
Canvas
Label
Component
TextComponent
TextArea
TextField
List
Checkbox
Choice
ItemSelectable
Serializable
MenuComponent
MenuBar
MenuItem
CheckboxMenuItem
Menu
PopupMenu
MenuContainer
www.falkhausen.de Version 0.8 Copyright 2002 by Markus Falkhausen. All rights reserved.


javax.swing.*
JComponents
java.awt.image.ImageObserver
java.awt.MenuContainer
Serializable
java.awt.Component
java.awt.Container
JComponent
AbstractButton
JToggleButton
JRadioButton
JCheckBox
JButton
JMenuItem
JCheckBoxMenuItem
JMenu
JRadioButtonMenuItem
JComboBox
JList
JProgressBar
JSlider
JSpinner
JLabel
DefaultListCellRenderer
javax.swing.tree.DefaultTreeCellRenderer
javax.swing.table.DefaultTableCellRenderer
JToolTip
JSeparator
JPopupMenu.Separator
JToolBar.Separator
JTable
javax.swing.table.JTableHeader
JTree
javax.swing.text.JTextComponent
JEditorPane
JTextPane
JTextArea
JTextField
JFormattedTextField
JPasswordField
DefaultTreeCellEditor.DefaultTextField
Box
Box.Filler
JPanel
JSpinner.DefaultEditor
javax.swing.colorchooser.AbstractColorChooserPanel
JSplitPane
JTabbedPane
JRootPane
JLayeredPane
JDesktopPane
JInternalFrame
JInternalFrame.JDesktopIcon
JScrollPane
JScrollBar
JViewport
JToolBar
JMenuBar
JPopupMenu
JColorChooser
JFileChooser
JOptionPane
www.falkhausen.de Version 0.8 Copyright 2002 by Markus Falkhausen. All rights reserved.

Héritage et javadoc

javax.swing

Class JComponent

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--javax.swing.JComponent
```

All Implemented Interfaces:
ImageObserver, MenuContainer, Serializable

Direct Known Subclasses:
AbstractButton, BasicInternalFrameTitlePane, Box, Box.Filler, JColorChooser, JComboBox, JFileChooser, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSpinner, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, JViewport

---

Fields inherited from class java.awt.Component

BOTTOM_ALIGNMENT, CENTER_ALIGNMENT, LEFT_ALIGNMENT, RIGHT_ALIGNMENT, TOP_ALIGNMENT

Fields inherited from interface java.awt.image.ImageObserver

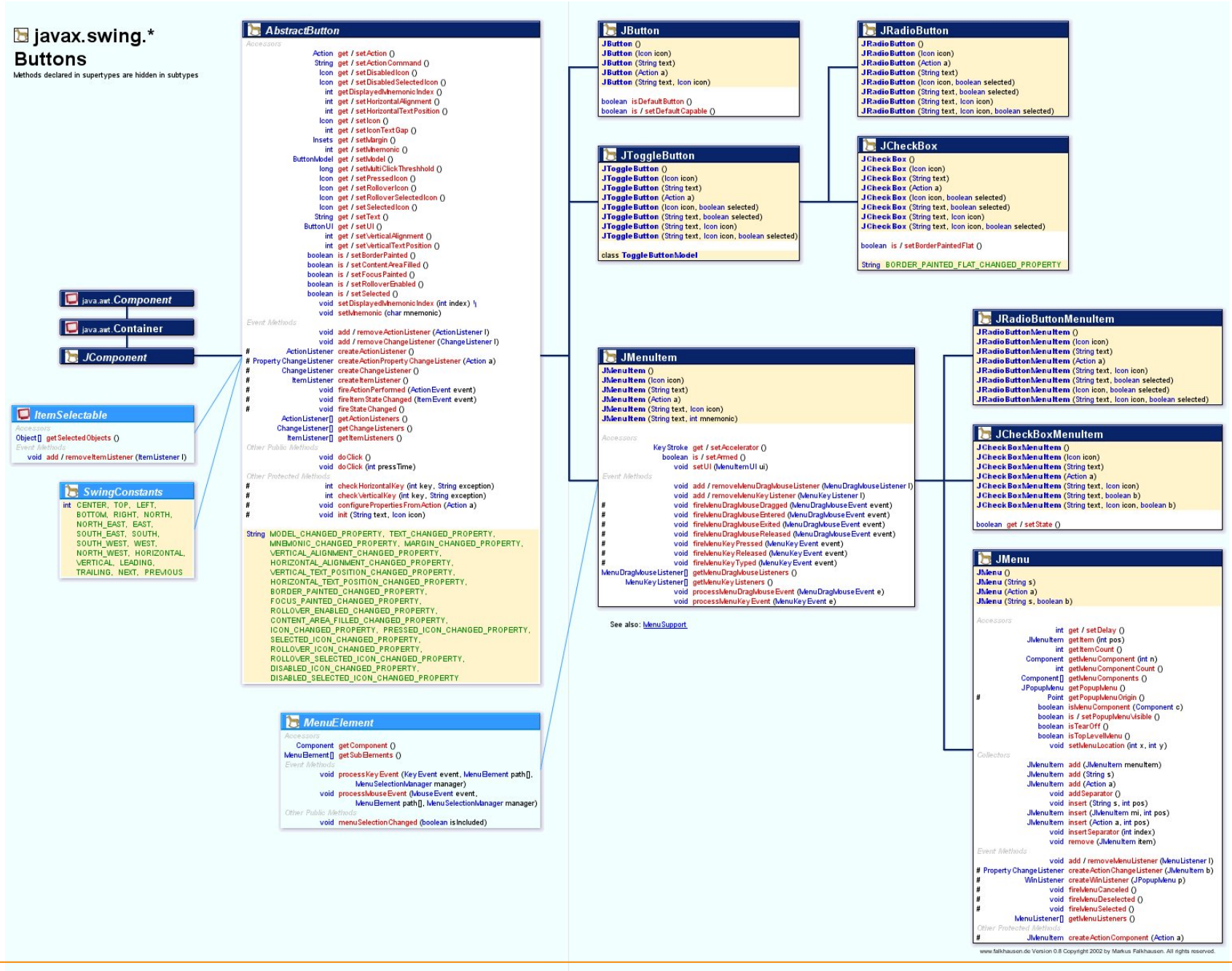
ABORT, ALLBITS, ERROR, FRAMEBITS, HEIGHT, PROPERTIES, SOMEBITS, WIDTH

Methods inherited from class java.awt.Container

add, add, add, add, add, addContainerListener, addImpl, applyComponentOrientation, areFocusTraversalKeysSet, countComponents, deliverEvent, doLayout, findComponentAt, findComponentAt, getComponent, getComponentAt, getComponentAt, getComponentCount, getComponents, getContainerListeners, getFocusTraversalKeys, getFocusTraversalPolicy, getLayout, insets, invalidate, isAncestorOf, isFocusCycleRoot, isFocusCycleRoot, isFocusTraversalPolicySet, layout, list, list, locate, minimumSize, paintComponents, preferredSize, printComponents, processContainerEvent, processEvent, remove, remove, removeAll, removeContainerListener, setFocusCycleRoot, setFocusTraversalKeys, setFocusTraversalPolicy, setLayout, transferFocusBackward, transferFocusDownCycle, validate, validateTree

Methods inherited from class java.awt.Component

action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener, addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener, bounds, checkImage, checkImage, coalesceEvents, contains, createImage, createImage, createVolatileImage, createVolatileImage, disableEvents, dispatchEvent, enable, enableEvents, enableInputMethods, getBackground, getBounds, getColorModel, getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners,


javax.swing.*
Buttons
Methods declared in supertypes are hidden in subtypes
AbstractButton
JButton
JRadioButton
JToggleButton
JCheckBox
java.awt.Component
java.awt.Container
JComponent
ItemSelectable
SwingConstants
MenuElement
JMenuItem
JRadioButtonMenuItem
JCheckBoxMenuItem
JMenu
See also: MenuSupport

Exemple : Dessin de Formes

- On souhaite réaliser une application de dessin permettant, entre autres, la manipulation de différentes formes graphiques : cercles, triangles, rectangles, polygônes, carrés, etc.
- Un objet `Feuille` gère l'état courant du dessin,
- L'application dispose d'outils permettant de déplacer la forme sélectionnée, d'en changer la taille, de lui appliquer une rotation, etc.

`Feuille` ou chacune des classes décrivant les outils doivent pouvoir être implémentées **indépendamment** des formes existantes
$\hookrightarrow$ on doit pouvoir **facilement** ajouter des formes à l'application.

(Open Close Principle)

```
public class Feuille {
   protected List<Shape> lesFormes;
   public void repaint() {
      for(Shape s:  this.lesFormes) {
         s.draw();
      }
   }
}
public class MoveTool {
   public void move(Shape selectedShape) {
      selectedShape.move(...);
   }
}
public class RotateTool {
   public void rotate(Shape selectedShape) {
      selectedShape.rotate(...);
   }
}
public class ResizeTool {
   public void resize(Shape selectedShape) {
      selectedShape.resize(...);
   }
}
```

Il est nécessaire de disposer d'un abstraction **`Shape`**

Interface ou héritage ?

interface il n'est pas possible de définir un comportement pour les méthodes `draw`, `move`, `rotate`, `resize` ou d'autres de `Shape`.

héritage Un certain nombre de comportements et d'attributs vont pouvoir être définis de manière commune pour toutes les formes : couleur du trait, motif du remplissage, épaisseur du trait, gestion du plan de profondeur d'affichage, etc.

Autre possibilité, utiliser une approche hybride :

une **classe abstraite**

Classes abstraites

Une classe abstraite

- est une classe (comporte donc des attributs et des méthodes),
- déclare de méthodes sans comportement attaché, ces méthodes sont dites **méthodes abstraites**,
- ne peut pas créer d'instance (même si elle peut déclarer un constructeur),
- les classes qui en héritent doivent concrétiser le comportement de **toutes** les méthodes abstraites
  $\hookrightarrow$ sous peine d'être elles aussi abstraites

abstract

- déclaration classe abstraite : ajouter le qualificatif **abstract** avant `class` dans l'entête de déclaration.

```
public abstract class ClasseAbstraite ...  {
    ...

}
```

abstract

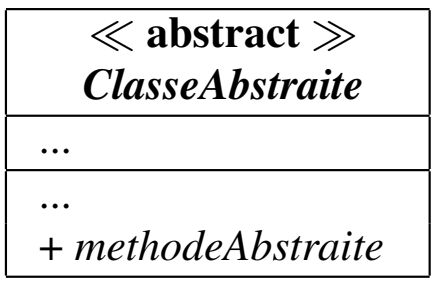
- déclaration classe abstraite : ajouter le qualificatif **abstract** avant `class` dans l'entête de déclaration.
- déclaration méthode abstraite : ajouter le qualificatif **abstract** avant le type de retour dans la signature de la méthode et ne donner aucun corps.

```
public abstract class ClasseAbstraite ...  {
   ...
   public abstract Type methodeAbstraite(...);
}
```

abstract

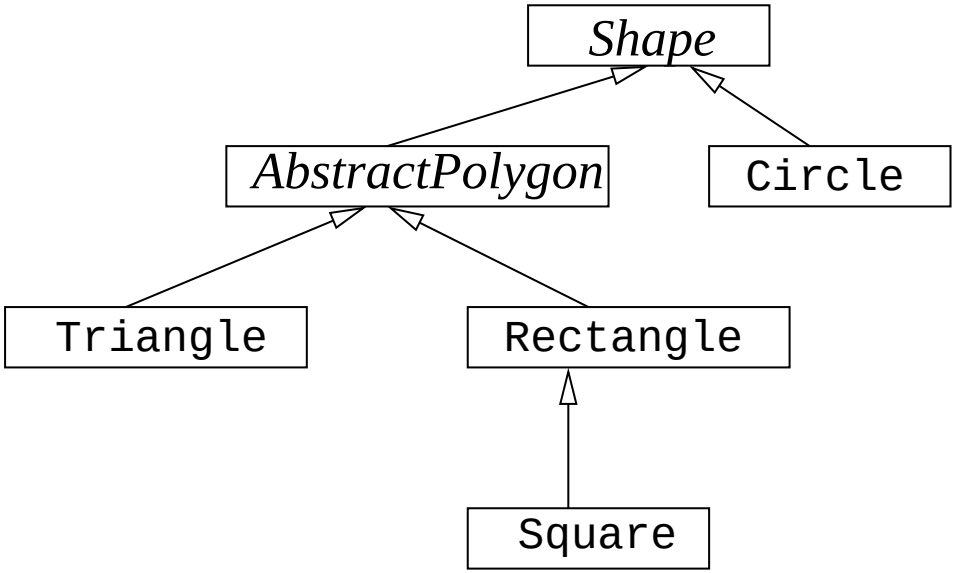
- déclaration classe abstraite : ajouter le qualificatif **`abstract`** avant `class` dans l'entête de déclaration.
- déclaration méthode abstraite : ajouter le qualificatif **`abstract`** avant le type de retour dans la signature de la méthode et ne donner aucun corps.

```
public abstract class ClasseAbstraite ...  {
    ...
    public abstract Type methodeAbstraite(...);
}
```



Retour en Forme

```
public abstract class Shape {
   protected int lineWidth;
   protected FillStyle fillStyle;
   protected byte depth;
   public void setLineWidth(int width) { this.lineWidth = width }
   public void toFront() { depth = 0; }
   public FillStyle getFillStyle() { return fillStyle; }

   public abstract void draw();
   public abstract void rotate(double angle);
   public abstract void move(...);
   public abstract void resize(...);
}

public class Circle extends Shape {
   protected Point center;
   protected int radius;
   public void draw() { ...  code ...  }
   public void rotate(double angle) { ...  code ...  }
   public void move(...)  { ...  code ...  }
   public void resize(...)  { ...  code ...  }
   public Point getCenter() { ...  code ...  }
}
```




```
public abstract class Shape { ...  }

public class Circle extends Shape { ...  }
public abstract class AbstractPolygon extends Shape {
   protected List<Edge> edges;
   public abstract int getNumberOfEdges();
   ...
}
//  les deux classes suivantes fournissent une définition pour getNumberOfEdges()
// (et les autres méthodes abstraites de Shape)
public class Triangle extends AbstractPolygon { ...  }
public class Rectangle extends AbstractPolygon { ...  }
// réutilise le getNumberOfEdges() de Rectangle
public class Square extends Rectangle { ...  }
```

Retour sur les enum

Méthodes différenciées pour chaque constante

```
package operation;
import java.util.*;
public enum Operation {
   plus {
      public double eval(double x, double y) { return x + y; }
   },
   minus {
      public double eval(double x, double y) { return x - y; }
   },
   times {
      public double eval(double x, double y) { return x * y; } },
   divided_by {
      public double eval(double x, double y) { return x / y; } };

   public abstract double eval(double x, double y);
   //-------------------------------------------------------------------
   public static void main(String args[]) {
      double x = Double.parseDouble(args[0]);
      double y = Double.parseDouble(args[1]);
      for (Operation op :  Operation.values()) {
         System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
}}}
```

Retour sur les enum

Méthodes différenciées pour chaque constante

```
package operation;
import java.util.*;
public enum Operation {
   plus {
      public double eval(double x, double y) { return x + y; }
   },
   minus {
      public double eval(double x, double y) { return x - y; }
   },
   times {
      public double eval(double x, double y) { return x * y; } },
   divided_by {
      public double eval(double x, double y) { return x / y; } };

   public abstract double eval(double x, double y);
   //-------------------------------------------------------------------
   public static void main(String args[]) {
      double x = Double.parseDouble(args[0]);
      double y = Double.parseDouble(args[1]);
      for (Operation op :  Operation.values()) {
         System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
}}}
```

```
java operation.Operation 6 3
6.0 plus 3.0 = 9.0
6.0 minus 3.0 = 3.0
6.0 times 3.0 = 18.0
6.0 divided_by 3.0 = 2.0
```

Des exemples

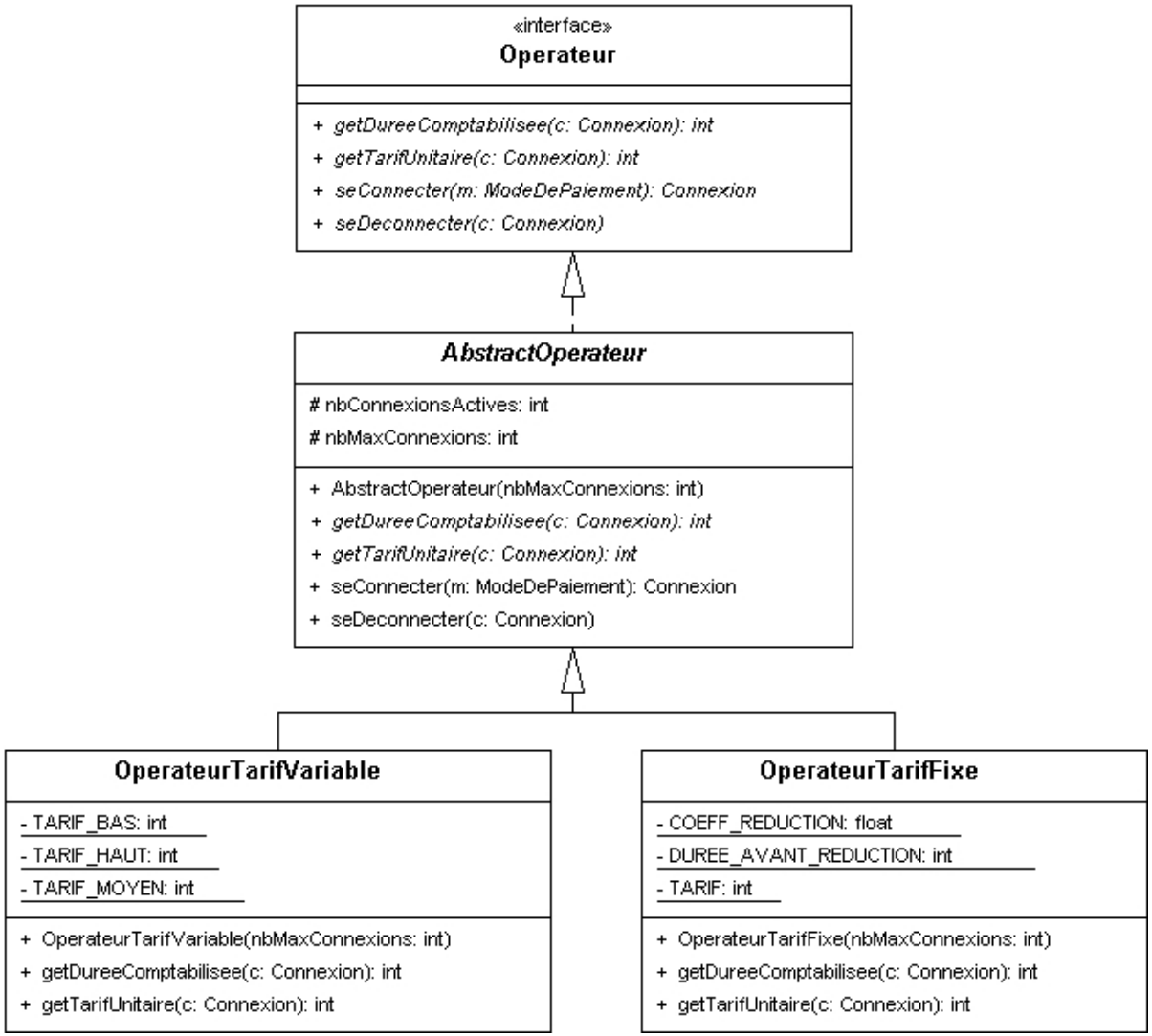
- dans `java.util`

```
public interface Collection<E> { ... }
public abstract class AbstractCollection<E> implements Collection<E> { ... }
public abstract class AbstractSet<E> extends AbstractCollection<E> { ... }
public abstract class AbstractList<E> extends AbstractCollection<E> { ... }
public class ArrayList<E> extends AbstractList<E> { ... }
public class LinkedList<E> extends AbstractList<E> { ... }
```

- dans `javax.swing`

```
public abstract class JComponent extends java.awt.Container { ... }
public class JPanel extends JComponent { ... }
public abstract class AbstractButton extends JComponent { ... }
public class JButton extends AbstractButton { ... }
public class JMenuItem extends AbstractButton { ... }
```

Opérateurs


«interface»
Operateur
+ getDureeComptabilisee(c: Connexion): int
+ getTarifUnitaire(c: Connexion): int
+ seConnecter(m: ModeDePaiement): Connexion
+ seDeconnecter(c: Connexion)
AbstractOperateur
nbConnexionsActives: int
nbMaxConnexions: int
+ AbstractOperateur(nbMaxConnexions: int)
+ getDureeComptabilisee(c: Connexion): int
+ getTarifUnitaire(c: Connexion): int
+ seConnecter(m: ModeDePaiement): Connexion
+ seDeconnecter(c: Connexion)
OperateurTarifVariable
- TARIF_BAS: int
- TARIF_HAUT: int
- TARIF_MOYEN: int
+ OperateurTarifVariable(nbMaxConnexions: int)
+ getDureeComptabilisee(c: Connexion): int
+ getTarifUnitaire(c: Connexion): int
OperateurTarifFixe
- COEFF_REDUCTION: float
- DUREE_AVANT_REDUCTION: int
- TARIF: int
+ OperateurTarifFixe(nbMaxConnexions: int)
+ getDureeComptabilisee(c: Connexion): int
+ getTarifUnitaire(c: Connexion): int

Scrutins

```
...
public abstract class AbstractScrutin implements Scrutin {
   protected boolean estClos;
   public boolean estClos() {
      return this.estClos;
   }
   ...
   public abstract BulletinVote getVainqueur();
}

public class ScrutinMajoritaire extends AbstractScrutin {
   public BulletinVote getVainqueur() {
      ...  TRAITEMENT Majoritaire...
   }
}

public class ScrutinRelatif extends AbstractScrutin {
   public BulletinVote getVainqueur() {
      ...  TRAITEMENT Relatif...
   }
}
```

Clients bancaires

```
...
public abstract class Client {
   protected String adresse;
   protected List<Compte> comptes;
   public Client(String adresse) {
      this.adresse = adresse; this.comptes = new ArrayList<Compte>();
   }
   public abstract Identite getIdentite();
   public String getAdresse() { return this.adresse; }
   public List<Compte> getComptes() { return this.comptes; }
   public void ajouteCompte(Compte nouveau) { this.comptes.add(nouveau); }
   public void addEcriture(...)  throws UnsupportedException { ...  }
}
public class ClientPhysique extends Client {
   protected PersonneId id;
   public ClientPhysique(String adresse, PersonneId id) {
      super(adresse);
      this.id = id;
   }
   public Identite getIdentite() { return this.id; }
   ...
}
public class ClientEntreprise extends Client {
   protected EntrepriseId id;
   protected String SIRET;
   public Identite getIdentite() { return this.id; }
   ...
}
```

Multi-héritage ?

- faire hériter une classe simultanément de classes et récupérer ainsi les comportements de chacune

```
public class Mammifere extends Animal {
   public String organeDeRespiration() {
      return "poumons";
   }
}
public class Nageur {
   public void nage() { ... }
}
public class Cetace extends Mammifere, Nageur { // interdit en Java
}
   // ...utilisation...
Cetace cet = new Cetace();
System.out.println(cet.organeDeRespiration());
cet.nage();
Mammifere mam = cet;                    // upcast possible vers l'une ou
Nageur nageur = cet;                    // l'autre des classes
```

Attention

L'héritage multiple de classes **n'existe pas** en JAVA

Problème

- Quelle définition choisir en cas d'une déclaration **multiple** dans les super-classes ?

```
public class Mammifere extends Animal {
   public int getSize() { ...  }
}
public class Nageur {
   public int getSize() { ...  }
}
public class Cetace extends Mammifere, Nageur {
}
   // ...utilisation...
Cetace cet = new Cetace();
System.out.println(cet.getSize());// Quelle définition de méthode ?
```

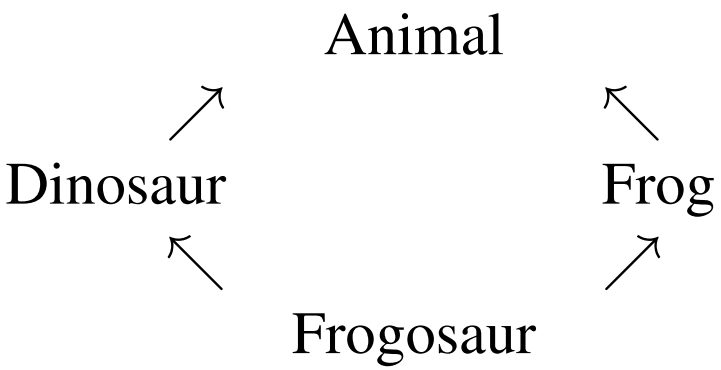
- une solution (?) : imposer le "cast"

```
Cetace cet = new Cetace();
System.out.println((Mammifere) cet.getSize());
ou
System.out.println((Nageur) cet.getSize());
```

Le problème du diamant

ou “*le losange de la mort*”

(d’après B. Venners)



Jurassic Park (M. Crichton)

```
public abstract class Animal {
   public abstract void talk();
}
public class Frog extends Animal {
   public void talk() {
      System.out.println("Ribit, Ribit");
   }
   public void jump() { ...  }
}
public class Dinosaur extends Animal {
   public void talk() {
      System.out.println("I'm a poor lonesome dinosaur");
   }
   public boolean isCarnivorous() { ...  }
}
public class Frogosaur extends Frog,Dinosaur {
   // impossible en Java
}
   // ...  utilisation ...
Animal animal = new Frogosaur();
animal.talk();                          // !!!  ambiguïté !!!
```

Le problème du diamant est lié au multi-héritage de 2 classes qui descendent de la même super-classe, lors de l'upcast vers la super-classe commune (ici "super-super-classe")

Quelle définition de `talk()` choisir ?

Le problème apparaît aussi pour d'éventuels attributs dans `Animal` :

Quelle copie de l'attribut considérer ? (y a-t-il 2 copies ?)

en JAVA

- héritage **simple** de classe
- implémentation **multiple** des interfaces.

Utiliser les interfaces

Choisir l'héritage de l'une des deux classes et définir une interface pour la partie spécifique à l'autre et que l'on souhaitait récupérer.

Utiliser les interfaces

Choisir l'héritage de l'une des deux classes et définir une interface pour la partie spécifique à l'autre et que l'on souhaitait récupérer.

```
public abstract class Animal {
   public abstract void talk();
}
class Frog extends Animal {
   public void talk() {
      System.out.println("Ribit, Ribit");
   }
   public void jump() { ...  }
}
interface DinosaurInterface {
   public boolean isCarnivorous();
}
public class Frogosaur extends Frog
      implements DinosaurInterface {
   public boolean isCarnivorous() { ...  }
}
   // ...  utilisation...
Frogosaur frogosaur = new Frogosaur();
Animal animal = frogosaur;
animal.talk();
Frog frog = frogosaur;
DinosaurInterface din = frogosaur;
```

Utiliser les interfaces

Choisir l'héritage de l'une des deux classes et définir une interface pour la partie spécifique à l'autre et que l'on souhaitait récupérer.

```
public abstract class Animal {
   public abstract void talk();
}
class Frog extends Animal {
   public void talk() {
      System.out.println("Ribit, Ribit");
   }
   public void jump() { ...  }
}
interface DinosaurInterface {
   public boolean isCarnivorous();
}
public class Frogosaur extends Frog
      implements DinosaurInterface {
   public boolean isCarnivorous() { ...  }
}
   // ...  utilisation...
Frogosaur frogosaur = new Frogosaur();
Animal animal = frogosaur;
animal.talk();
Frog frog = frogosaur;
DinosaurInterface din = frogosaur;
```

```
public abstract class Animal {
   public abstract void talk();
}
interface FrogInterface {
   public void jump();
}
class Dinosaur {
   public void talk() {
      System.out.println("Ribit, Ribit");
   }
   public boolean isCarnivorous() { ...  }
}
public class Frogosaur extends Dinosaur
      implements FrogInterface {
   public void jump() { ...  }
}
   // ...  utilisation ...
Frogosaur frogosaur = new Frogosaur();
Animal animal = frogosaur;
animal.talk();
FrogInteface frog = frogosaur;
Dinosaur din = frogosaur;
```

Combiner interfaces et héritage

Accroître le polymorphisme
tout en évitant les problèmes liés au multi-héritage.

Permettre de “multi-typer” les objets
et
d’accroître l’**abstraction** grâce à l’upcast

```
public interface Aquatique {
   public void nage();
}
public class Cetace extends Mammifere implements Aquatique {
   public void nage() { ... }
}
public class Zoo {
   public void addAnimal(Animal animal) { ... }
}
public class Aquarium {
   public void addAquatique(Aquatique aquatique) { ... }
}
public class ZooAvecAquarium extends Zoo {
   protected Aquarium aquarium;
   public void addCetace(Cetace cetace) {
                              // à la fois un Animal et un Aquatique
      this.addAnimal(cetace);
      this.aquarium.addAquatique(cetace);
} }

   // ... utilisation ...
ZooAvecAquarium zoo = new ZooAvecAquarium();
Cetace cet = new Cetace();
zoo.addCetace(cet);
```

Exemple

```
public interface Talkative {
   public void talk();
}
public abstract class Animal implements Talkative { ...  }
public class Frog extends Animal {
   public void talk() {
      System.out.println("Ribit, Ribit");
   }
}
public class Dog extends Animal {
   public void talk() {
      System.out.println("Ouah, Ouah");
   }
}
public class Interrogator {
   public void makeItTalk(Talkative subject) {
      subject.talk();
   }
} // Interrogator
```

```
public class CuckooClock extends Clock implements Talkative {
   public void talk() {
      System.out.println("Ah que coucou !!!");
   }
   public Time getTime() {
      ...
   }
   public static void main(String[] args){
      (new Interrogator()).makeItTalk(new CuckooClock());
   }
} // CuckooClock
```

- **N'importe quelle classe** peut potentiellement implémenter `Talkative` et ainsi être passée en argument de `makeItTalk`

Héritage d'interfaces

- Rappel : le multi-héritage est possible avec les interfaces.
  (dans ce cas pas de conflit en cas de définitions multiples)

```
public interface Aquatique {
   public void nage();
}
public interface Marin extends Aquatique {
}
public interface Predateur {
   public void chasse();
}
public interface PredateurMarin extends Marin, Predateur {
   public int kiloPoissonsMangesParjour();
}
```

Composition et Héritage

Deux manières de réutiliser

- Héritage relation *is-a*
- Composition relation *has-a*

Réutiliser : héritage

Avantages :

- liaison dynamique et polymorphisme (utilisation de upcast)

Intérêts :

- modification de code en remplaçant un objet de classe *C* par une instance d'une sous-classe de *C*

```
AbstractList<T> l = new ArrayList<T>();
```

peut devenir

```
AbstractList<T> l = new LinkedList<T>();
```

- Mais seul ce type de modification par introduction de nouvelles sous-classes est facilité.

fragilité l'interface de la super-classe

super-classe **fragile**

⇒ La plus légère des modifications dans la superclasse peut corrompre le comportement des sous-classes.

En fait c'est l'**interface** de la super-classe qui est sensible/fragile

- On dit que l'héritage réalise une *encapsulation faible* :
  un changement de l'interface de la super-classe corrompt le code qui utilise la sous-classe

Exemple

```
public class Fruit {
   public int peel() {
      System.out.println("peel done");
      return 1;
   }
}

public class Apple extends Fruit { }    // héritage

public class Example1Fruit {
   public static void main(String[] args) {
      Apple apple = new Apple();
      int pieces = apple.peel();
   }
} // Example1Fruit
```

Changement de l'interface

```
public class Peel {
   private int numberOfPeels;
      public Peel(int numberOfPeels) {
         this.numberOfPeels = numberOfPeels;
      }
   public int getNumberOfPeels() { return numberOfPeels; }
}
public class Fruit {                          // change interface Fruit
   public Peel peel() {                       // change type de retour de peel()
      System.out.println("peel done");
      return new Peel(1);
   }
}
public class Apple extends Fruit { }          // rien n'est à changer

      // corruption de Example1Fruit
public class Example1CorrompuFruit {
   public static void main(String[] args) {
      Apple apple = new Apple();
      int pieces = apple.peel();              // ERREUR : CODE CORROMPU
   }
} // Example1CorrompuFruit
```

Utilisation de la composition

au lieu d'hériter d'une classe C, on peut :

1. détenir une référence vers une instance de C
2. encapsuler les méthodes à récupérer de C dans de nouvelles méthodes

- il n'y a pas de récupération automatique de toute l'interface publique de C, elle doit être explicite $\Rightarrow$ **délégation**
- C devient une classe "back-end"
- encapsulation *plus forte* : on peut ne pas changer l'**interface** de la classe englobante lorsque l'on change celle de C

Rappel : *interface = signatures des messages acceptés par instances*

Exemple revisité

```
public class Fruit {
   public int peel() {
      System.out.println("peel done");
      return 1;
   }
}
public class Apple {
   private Fruit fruit = new Fruit();  // composition

   public int peel() {
      return this.fruit.peel();
   }
}
public class Example2Fruit {
   public static void main(String[] args) {
      Apple apple = new Apple();
      int pieces = apple.peel();
   }
} // Example2Fruit
```

Changement de l'interface

```
public class Peel {
   private int numberOfPeels;
   public Peel(int numberOfPeels) { this.numberOfPeels = numberOfPeels; }
   public int getNumberOfPeels() { return numberOfPeels; }
}
public class Fruit {                          // change interface Fruit
   public Peel peel() {                       // change du type de retour de peel()
      System.out.println("peel done");
      return new Peel(1);
   }
}
public class Apple {
   private Fruit fruit = new Fruit();
   public int peel() {                        // il faut changer le code de peel()
      return (this.fruit.peel()).getNumberOfPeels();
   }
}
public class Example2Fruit {                  // pas de corruption de Example2Fruit
   public static void main(String[] args) {
      Apple apple = new Apple();
      int pieces = apple.peel();
}} // Example2CorrompuFruit
```

Composition > Héritage

- C plus facile de changer l'interface d'une classe "back-end" que d'une super-classe.
  ↪ permet de conserver l'**interface** de la classe "front-end" (même si chgmt code)
- C plus facile de changer l'interface d'une classe "front-end" que d'une sous-classe.
  ↪ il n'est pas toujours possible de changer l'interface d'une méthode héritée (ex : type de retour); avec l'agrégation, on a le choix.
- C avec agrégation, on peut retarder la création des objets agrégés avec l'héritage, sous-couche créée dès la création de l'objet

- = le changement de l'implémentation (pas de l'interface) est aussi facile dans les 2 cas

Héritage > composition

- **H** plus facile d'ajouter des sous-classes par héritage que d'ajouter des classes "front-end" par composition
  ↪ grâce au polymorphisme, on peut facilement passer d'une (instance de) super-classe à sa sous-classe sans changer de code
  pas si simple avec composition...
- **H** la délégation d'invocation de méthode de la composition a un coût, surtout si elle est systématique
- **H** lien par héritage plus facile à appréhender que lien par composition

- **=** le changement de l'implémentation (pas de l'interface) est aussi facile dans les 2 cas

Utiliser les interfaces

- composition plus de flexibilité, risque de corruption de code limité
- héritage plus facile d'hériter des sous-classes (grâce à l'utilisation de l'upcast)

Penser à combiner composition et interfaces

Exemple revisité (2)

```
public interface Peelable {
   public Peel peel();
}
public class Fruit {
   public int peel() {
      System.out.println("peel done");
      return 1;
   }
}
public class Apple implements Peelable {
   private Fruit fruit = new Fruit();
   public Peel peel() {
      return new Peel(this.fruit.peel());
   }
}
public class PeelProcessor() {
   public void peelSomething(Peelable something) {
      something.peel();
   }
   public static void main(String[] args) {
      PeelProcessor pp = new PeelProcessor();
      pp.peelSomething(new Apple());
   }
} // PeelProcessor
```

- le problème de la modification de la classe “back-end” est pris en charge par la composition
- le problème de l’ajout de sous-classe est prise en charge par l’interface `Peelable`

Héritage=tentation

- héritage **SSI** relation *is-a* entre les classes
  exemples :
  - `Apple` *is-a* `Fruit` semble naturel
  - `StudentList` *is-a* `ArrayList<Student>` (me ?) paraît plus discutable
- la relation *is-a* est elle durable/fiable ?
  pour la durée l'application mais aussi la durée du code en général (API)
  on peut avoir une `Person` *is-a* `Employee` mais quid si la `Person` devient sans emploi... ou devient un `Boss`...
- ne pas utiliser l'héritage juste pour "récupérer" du code, préférer la composition

The Liskov Substitution Principle (LSP)

Les sous-classes doivent pouvoir remplacer leur classe de base
Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser "inconsciemment" des objets dérivés de cette classe

On doit pouvoir Upcaster sémantiquement

Le dilemme cercle/ellipse

un *Cercle* est une *Ellipse*

Ellipse : 2 foyers

Circle : 1 seul centre

problème avec `setFoyers(Point p1, Point p2)`

ne pas utiliser l'héritage juste pour factoriser du code

héritage $\Longrightarrow$ extension/spécialisation

(ce n'est pas le cas *Cercle* pour *Ellipse*, dans ce cas utiliser la composition)