

Héritage (2)

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



Université
Lille1
Sciences et Technologies

IEEA - FIL
Informatique

Piège ?..

Question piège.

```

public class C { public void f() { System.out.println("c"); } }
public class A extends C { public void f() { System.out.println("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

```

```

+-----+
| c
| appel avec type A
| appel avec type B
|-----+
| c
| a
| c

```

Late-binding : sur les références invoquant une méthode, pas sur les paramètres des méthodes !

Créer une classe d'exception

- ▶ dans la mesure du possible utiliser les exceptions existantes, sinon
- ❶ définir une classe en lui donnant un nom explicite de la forme
QuelqueChoseException
- ❷ 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

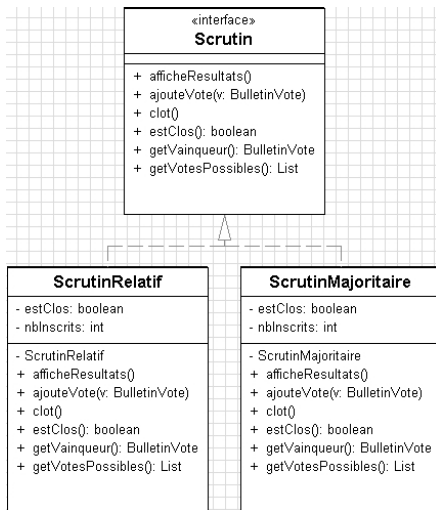
Lors de la surcharge d'une méthode d'une super-classe, la signature doit être rigoureusement la même, **jusqu'aux** exceptions.

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



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;
    }
}

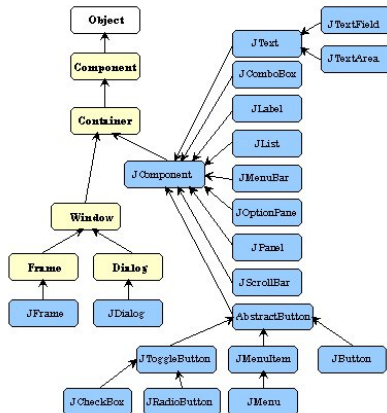
public class CompteEpargne extends Compte { ... }

```

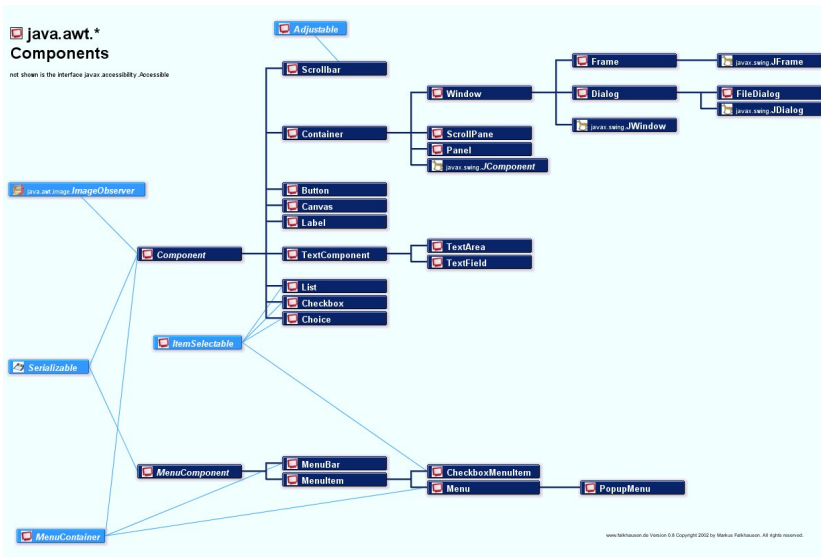
Hiérarchies de composants graphiques

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”



Exemples de hiérarchies de classes

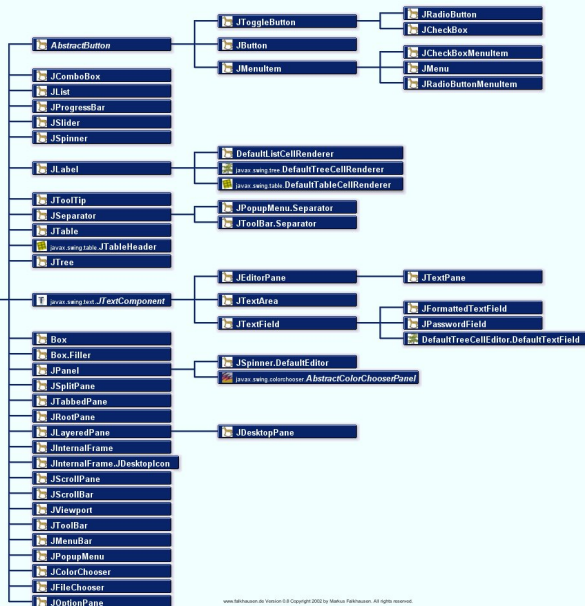


Exemples de hiérarchies de classes

javax.swing.*
JComponents

ImageObserver
MenuContainer
Serializable

Component
Container
JComponent



Héritage et javadoc

javax.swing

Class JComponent



All Implemented Interfaces:

[ImageObserver](#), [MenuContainer](#), [Serializable](#)

Direct Known Subclasses:

[AbstractButton](#), [BaseInternalFrame](#), [TitlePane](#), [Box](#), [Box.Filler](#), [JFileChooser](#), [JComboBox](#), [JFileChooser](#), [InternalFrame](#), [InternalFrame.Desktop](#), [JLabel](#), [JLayeredPane](#), [JList](#), [JMenuBar](#), [JOptionPane](#), [JPanel](#), [JProgressMonitor](#), [JProgressBar](#), [JRootPane](#), [JScrollPane](#), [JSeparator](#), [JSlider](#), [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](#), [addContainerListener](#), [addImpl](#), [applyComponentOrientation](#), [areFocusTraversalKeysSet](#), [countComponents](#), [deliverEvent](#), [doLayout](#), [findComponentAt](#), [getComponent](#), [getComponentAt](#), [getComponentAt](#), [getComponentCount](#), [getComponents](#), [getContainerListeners](#), [getFocusTraversalKeys](#), [getFocusTraversalPolicy](#), [getLayout](#), [insert](#), [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](#), [enableInputMethod](#), [getBackground](#), [getBounds](#), [getColorModel](#), [getComponentListeners](#), [getComponentOrientation](#), [getCursor](#), [getFont](#), [getFont](#), [setFocusCycleRootAncestor](#), [setFocusListeners](#)

Buttons

SwingConstants

- CENTER, TOP, LEFT, BOTTOM, RIGHT, NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST, HORIZONTAL, VERTICAL, LEADING, TRAILING, NEXT, PREVIOUS

[illegible]

```

Component getComponent()
Event[] getSubEvents()
Abstract
void processKeyEvent(KeyEvent event, MenuBarEvent path,
    MenuSelectionManager manager)
void processMouseEvent(MouseEvent event,
    MenuBarEvent path, MenuSelectionManager manager)
Public Methods
void menuSelectionChanged(boolean isInvoked)

```

```

JButton (Action a)
JButton (String text, Icon icon)

boolean isDefaultButton ()
boolean isSetDefaultCapable ()

```

```
JToggleButton (Icon icon)
JToggleButton (String text)
JToggleButton (Action a)
JToggleButton (Icon icon, boolean selected)
JToggleButton (String text, boolean selected)
JToggleButton (String text, Icon icon)
JToggleButton (String text, Icon icon, boolean selected);
```

```

JblennaHess (Icon icon)
JblennaHess (String test)
JblennaHess (Action a)
JblennaHess (String test, Icon icon)
JblennaHess (String test, int mnemonic)

Accessors
KeyStroke get / setAccess
boolean is / setAccess
void setJblennaHess

```

```
void FireAlarmDraghouseDragged (AlarmDraghouseEvent event)
void FireAlarmDraghouseEntered (AlarmDraghouseEvent event)
void FireAlarmDraghouseExited (AlarmDraghouseEvent event)
void FireAlarmDraghouseReleased (AlarmDraghouseEvent event)
void FireAlarmKeyPressed (AlarmKeyEvent event)
void FireAlarmKeyReleased (AlarmKeyEvent event)
void FireAlarmKeyUpTyped (AlarmKeyUpEvent event)
AlarmDraghouseListeners()
AlarmKeyListeners()
processAlarmDraghouseEvent (AlarmDraghouseEvent e)
processAlarmKeyUpEvent (AlarmKeyUpEvent e)
```

<code>JRadioButtonMenuItem</code>	(String text)
<code>JRadioButtonMenuItem</code>	(Action a)
<code>JRadioButtonMenuItem</code>	(String text, Icon icon)
<code>JRadioButtonMenuItem</code>	(String text, boolean selected)
<code>JRadioButtonMenuItem</code>	(Icon icon, boolean selected)
<code>JRadioButtonMenuItem</code>	(String text, Icon icon, boolean selected)

```

JCheck Boolean return (Icon icon)
JCheck Boolean return (String text)
JCheck Boolean return (Action a)
JCheck Boolean return (String text, Icon icon)
JCheck Boolean return (String text, boolean b)
JCheck Boolean return (String text, Icon icon, boolean b)
boolean out (outState f)

```

- **Rule** (String s)
- **Rule** (Action a)
- **Rule** (String s, boolean b)

```

    int getDelay ()
    JMeterBean getBean (int pos)
    int getBeanCount ()
    Component getBeanComponent (int n)
    int getBeanComponentCount ()
    Component[] getBeanComponents ()
    JPopUpMenu getPopUpMenu ()
    Point getPopUpMenuOrigin ()
    boolean isBeanComponent (Component)
    boolean isSetPopUpMenuVisible ()
    boolean isToolOff ()
    boolean isTrackMouse ()

```

```

AddItem() add (AddItem menu item)
AddItem() add (String s)
AddItem() add (Action a)
void addSeparator()
void insert (String s, int pos)
AddItem() insert (AddItem mi, int pos)
AddItem() insert (Action a, int pos)
void insertSeparator (int index)

```

```
void add (IItemViewMenuListener pMenuListener)
// Property Change Listener
createActionChangeListener (IActionChangeListener p)
// Win Listener
createWinListener (IAppPopupMenu p)
void fireMenuCanceled ()
void fireMenuDeselected ()
void fireMenuSelected ()
// Menu Listener
```

www.fishbase.org, de Vries et al. 2003, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025, 2026, 2027, 2028, 2029, 2030, 2031, 2032, 2033, 2034, 2035, 2036, 2037, 2038, 2039, 2040, 2041, 2042, 2043, 2044, 2045, 2046, 2047, 2048, 2049, 2050, 2051, 2052, 2053, 2054, 2055, 2056, 2057, 2058, 2059, 2060, 2061, 2062, 2063, 2064, 2065, 2066, 2067, 2068, 2069, 2070, 2071, 2072, 2073, 2074, 2075, 2076, 2077, 2078, 2079, 2080, 2081, 2082, 2083, 2084, 2085, 2086, 2087, 2088, 2089, 2090, 2091, 2092, 2093, 2094, 2095, 2096, 2097, 2098, 2099, 2100, 2101, 2102, 2103, 2104, 2105, 2106, 2107, 2108, 2109, 2110, 2111, 2112, 2113, 2114, 2115, 2116, 2117, 2118, 2119, 2120, 2121, 2122, 2123, 2124, 2125, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2133, 2134, 2135, 2136, 2137, 2138, 2139, 2140, 2141, 2142, 2143, 2144, 2145, 2146, 2147, 2148, 2149, 2150, 2151, 2152, 2153, 2154, 2155, 2156, 2157, 2158, 2159, 2160, 2161, 2162, 2163, 2164, 2165, 2166, 2167, 2168, 2169, 2170, 2171, 2172, 2173, 2174, 2175, 2176, 2177, 2178, 2179, 2180, 2181, 2182, 2183, 2184, 2185, 2186, 2187, 2188, 2189, 2190, 2191, 2192, 2193, 2194, 2195, 2196, 2197, 2198, 2199, 2200, 2201, 2202, 2203, 2204, 2205, 2206, 2207, 2208, 2209, 2210, 2211, 2212, 2213, 2214, 2215, 2216, 2217, 2218, 2219, 2220, 2221, 2222, 2223, 2224, 2225, 2226, 2227, 2228, 2229, 2230, 2231, 2232, 2233, 2234, 2235, 2236, 2237, 2238, 2239, 2240, 2241, 2242, 2243, 2244, 2245, 2246, 2247, 2248, 2249, 2250, 2251, 2252, 2253, 2254, 2255, 2256, 2257, 2258, 2259, 2260, 2261, 2262, 2263, 2264, 2265, 2266, 2267, 2268, 2269, 2270, 2271, 2272, 2273, 2274, 2275, 2276, 2277, 2278, 2279, 2280, 2281, 2282, 2283, 2284, 2285, 2286, 2287, 2288, 2289, 2290, 2291, 2292, 2293, 2294, 2295, 2296, 2297, 2298, 2299, 2300, 2301, 2302, 2303, 2304, 2305, 2306, 2307, 2308, 2309, 2310, 2311, 2312, 2313, 2314, 2315, 2316, 2317, 2318, 2319, 2320, 2321, 2322, 2323, 2324, 2325, 2326, 2327, 2328, 2329, 2330, 2331, 2332, 2333, 2334, 2335, 2336, 2337, 2338, 2339, 2340, 2341, 2342, 2343, 2344, 2345, 2346, 2347, 2348, 2349, 2350, 2351, 2352, 2353, 2354, 2355, 2356, 2357, 2358, 2359, 2360, 2361, 2362, 2363, 2364, 2365, 2366, 2367, 2368, 2369, 2370, 2371, 2372, 2373, 2374, 2375, 2376, 2377, 2378, 2379, 2380, 2381, 2382, 2383, 2384, 2385, 2386, 2387, 2388, 2389, 2390, 2391, 2392, 2393, 2394, 2395, 2396, 2397, 2398, 2399, 2400, 2401, 2402, 2403, 2404, 2405, 2406, 2407, 2408, 2409, 2410, 2411, 2412, 2413, 2414, 2415, 2416, 2417, 2418, 2419, 2420, 2421, 2422, 2423, 2424, 2425, 2426, 2427, 2428, 2429, 2430, 2431, 2432, 2433, 2434, 2435, 2436, 2437, 2438, 2439, 2440, 2441, 2442, 2443, 2444, 2445, 2446, 2447, 2448, 2449, 2450, 2451, 2452, 2453, 2454, 2455, 2456, 2457, 2458, 2459, 2460, 2461, 2462, 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2470, 2471, 2472, 2473, 2474, 2475, 2476, 2477, 2478, 2479, 2480, 2481, 2482, 2483, 2484, 2485, 2486, 2487, 2488, 2489, 2490, 2491, 2492, 2493, 2494, 2495, 2496, 2497, 2498, 2499, 2500, 2501, 2502, 2503, 2504, 2505, 2506, 2507, 2508, 2509, 2510, 2511, 2512, 2513, 2514, 2515, 2516, 2517, 2518, 2519, 2520, 2521, 2522, 2523, 2524, 2525, 2526, 2527, 2528, 2529, 2530, 2531, 2532, 2533, 2534, 2535, 2536, 2537, 2538, 2539, 2540, 2541, 2542, 2543, 2544, 2545, 2546, 2547, 2548, 2549, 2550, 2551, 2552, 2553, 2554, 2555, 2556, 2557, 2558, 2559, 2560, 2561, 2562, 2563, 2564, 2565, 2566, 2567, 2568, 2569, 2570, 2571, 2572, 2573, 2574, 2575, 2576, 2577, 2578, 2579, 2580, 2581, 2582, 2583, 2584, 2585, 2586, 2587, 2588, 2589, 2590, 2591, 2592, 2593, 2594, 2595, 2596, 2597, 2598, 2599, 2600, 2601, 2602, 2603, 2604, 2605, 2606, 2607, 2608, 2609, 2610, 2611, 2612, 2613, 2614, 2615, 2616, 2617, 2618, 2619, 2620, 2621, 2622, 2623, 2624, 2625, 2626, 2627, 2628, 2629, 2630, 2631, 2632, 2633, 2634, 2635, 2636, 2637, 2638, 2639, 2640, 2641, 2642, 2643, 2644, 2645, 2646, 2647, 2648, 2649, 2650, 2651, 2652, 2653, 2654, 2655, 2656, 2657, 2658, 2659, 2660, 2661, 2662, 2663, 2664, 2665, 2666, 2667, 2668, 2669, 2670, 2671, 2672, 2673, 2674, 2675, 2676, 2677, 2678, 2679, 2680, 2681, 2682, 2683, 2

Exemple : Dessin de Formes

- ▶ On souhaite réaliser une application de dessin permettant, entre autres, la manipulation de différentes formes graphiques : cercles, triangles, rectangles, polygones, 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

↪ on doit pouvoir **facilement** ajouter des formes à l'application.

(Open Close Principle)

Exemple : des Formes

```

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.

Plutôt que de choisir, 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
↪ sous peine d'être elles aussi abstraites

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 >> <i>ClasseAbstraite</i>
...
... + <i>methodeAbstraite</i>

Retour en Forme

```

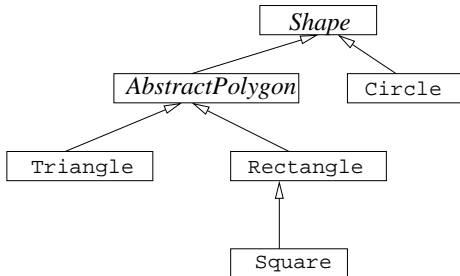
public abstract class Shape {
    protected int lineWidth;
    protected FillStyle fillStyle;
    protected byte depth;
    public void setLineWidth(int width) { this.lineWidth = width }
    public void toFront() { depth = 0; }
    public FillStyle getFillStyle() { return fillStyle; }

    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 ... }
}

```

Classes abstraites



```

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; } },
    dividedBy {
        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
```

Des exemples

► dans java.util

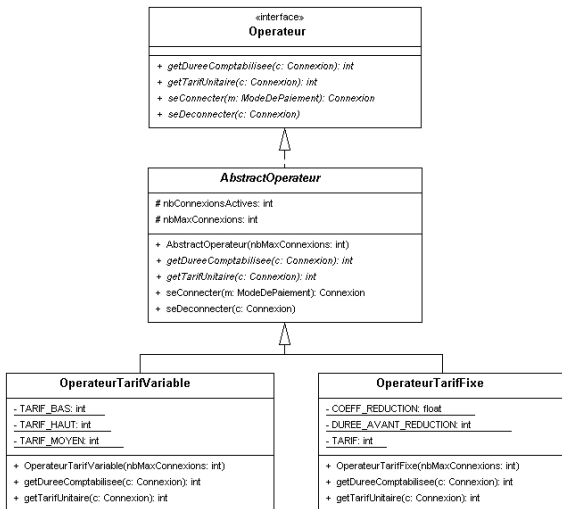
```
public interface Collection<E> { ... }
public abstract class AbstractCollection<E> implements Collection<E> { ... }
public abstract class AbstractSet<E> extends AbstractCollection<E> { ... }
public abstract class AbstractList<E> extends AbstractCollection<E> { ... }
public class ArrayList<E> extends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

Des exemples

Opérateurs



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 UnsupportedOperationException { ... }
}
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 ?
```

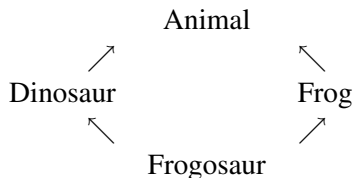
- une solution (?) : imposer le “cast”

```
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)

Multi-héritage ?

```

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.

Gestion du problème : utiliser les 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.

```
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();
FrogInterface 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

Gestion du problème : utiliser les interfaces

```

public interface Aquatique {
    public void nage();
}
public class Cétace 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 addCétace(Cétace cetace) {
        // à la fois un Animal et un Aquatique
        this.addAnimal(cetace);
        this.aquarium.addAquatique(cetace);
    } }

    // ... utilisation ...
    ZooAvecAquarium zoo = new ZooAvecAquarium();
    Cétace cet = new Cétace();
    zoo.addCétace(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 implements Talkative {
    public void talk() {
        System.out.println("Ah que coucou !!!");
    }
    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 :

- ❶ détenir une référence vers une instance de C
- ❷ 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 \implies extension/spécialisation

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