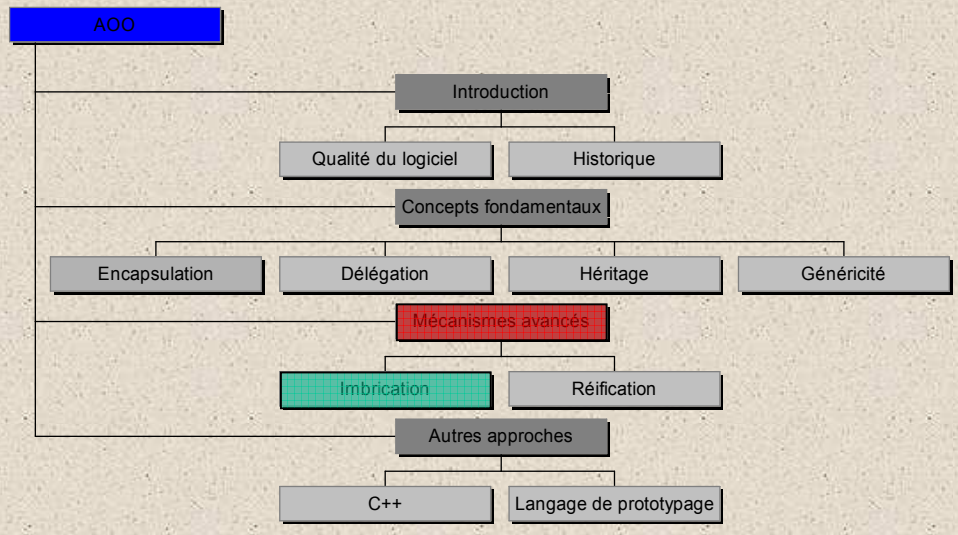
Imbrication

Inner class

Sommaire



Sommaire

La mutabilité

La portée : rappel

Classe imbriquée

- Classe anonyme
- Classe imbriquée dans un bloc classe
- Classe imbriquée dans un bloc fonctionnel
- Classe imbriquée statique

Les différentes utilisations

- L'exemple de la Tortue
- Un trieur
- Vue
- Exemple de Composition

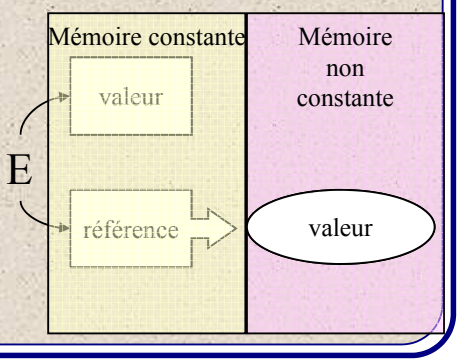
La mutabilité

Détermine si une caractéristique peut être modifiée ou non, cela concerne :

- Les variables
  - Attributs, paramètres
- Les méthodes
- Les classes

Soit E une déclaration de type T, E est dite non modifiable si elle est précédée du qualifieur "final".

- Type primitif : cela implique que la valeur est constante
- Type non primitif (objet) : cela indique que la référence est constante (pas la valeur)



Mutabilité d'attribut

Un attribut non modifiable doit être initialisée

- Statiquement : compile-binding
  - {static} final int exemple1 = 3;
- Dynamiquement :
  - à la création de l'objet (partie implicite du new)
    - {static} final double exemple2 = Math.random()*10;
  - à l'initialisation de l'objet (partie explicite du new)
    - final String exemple3;
    - Les codes des constructeurs doivent contenir une affectation de exemple3
- Exemple : l'image de la tortue
  - static final TurtleImage image = new TurtleImage("<nom du fichier image>");
  - static final TurtleImage image = new TurtleImage(<expression paramétrée>);
  - final TurtleImage image = new TurtleImage(<expression paramétrée>);
  - final TurtleImage image;
    - » public Turtle(TurtleArea feuille, String imageFile) {image = new TurtleImage(imageFile); ...}
  - final TurtleImage image = new TurtleImage("<nom du fichier image>");
  - Attention, si TurtleImage offre des méthodes de modification

Chaque tortue a une image qui ne peut être changée à l'exécution

Toutes les tortues ont la même image qui ne peut être changée que par recompilation

Mutabilité de variable

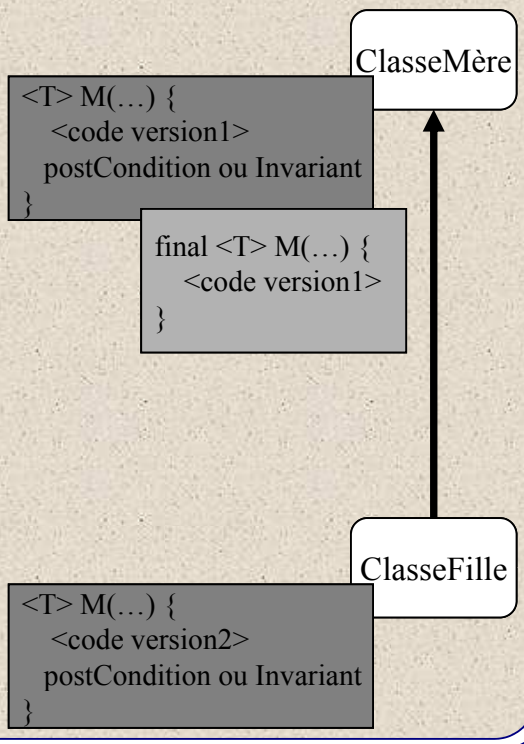
Une variable non modifiable ne peut être en partie gauche(lhs) d'une affectation

- Variable locale
  - L'intérêt est réduit, voire contradictoire avec la notion de variable
- Paramètre
  - D'un point de vue conceptuel, ceci permet d'indiquer le caractère "immutable" d'un paramètre en entrée.
    - Attention vœu pieux
      - » Public <T> <M>(final <T1> p1 …) {
      - » p1 = … // la référence p1 ne peut être modifiée
      - » p1.modifier(…); // opération qui peut entraîner un changement dans l'objet accessible
      - » }
  - D'un point de vue opératoire
    - Application dans le cas des classes imbriquées présentées dans la suite de ce chapitre.

Mutabilité de méthode

Une méthode non modifiable ne peut être redéfinie

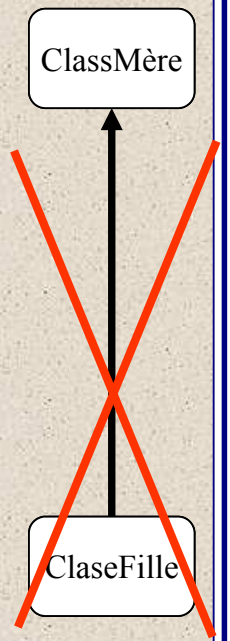
- La valeur d'une méthode, c'est son code
- Le changement de sa valeur, c'est sa redéfinition
- Utilité
  - D'un point de vue conceptuel, ceci permet de garantir des propriétés comportementales pour l'ensemble de l'arbre d'héritage
    - Réduction de propriété par du code
  - D'un point de vue opératoire, ceci permet au compilateur de faire des optimisations
    - Elimination de la liaison dynamique
    - Toute méthode "private" est de facto "final" car non redéfinissable par les classes dérivées



Mutabilité de classe

Une classe non modifiable ne peut avoir de sous-classe

- Interdit l'héritage
- Indique qu'une classe est une feuille de l'arbre d'héritage
- Utilités
  - D'un point de vue conceptuel, ceci permet de garantir des propriétés comportementales pour l'ensemble de l'arbre d'héritage
    - Réduction de propriété par du code
  - D'un point de vue opératoire, ceci permet au compilateur de faire des optimisations
    - Elimination de la liaison dynamique



La portée : rappel

La portée d'une caractéristique :

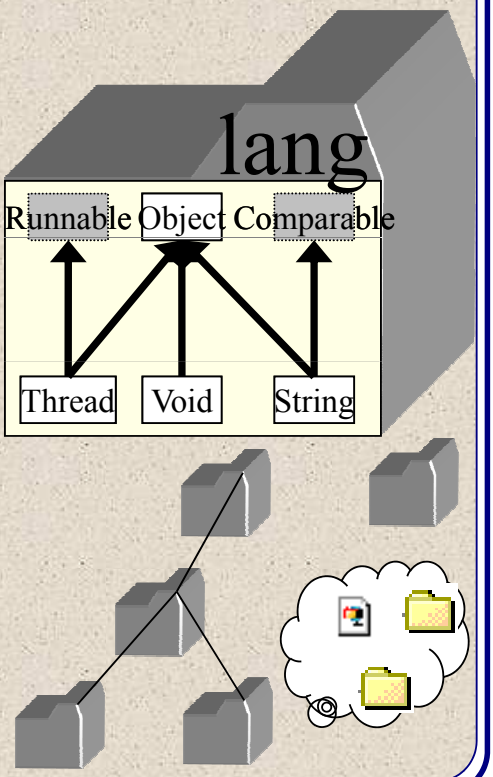
- Est définie par son qualifieur de portée
  - public, package, protected, private
- Elle indique l'espace dans lequel cette caractéristique est accessible
- N'a pas de sens pour les variables locales et les paramètres pour lesquels la portée est implicite

La portée d'une classe est définie par :

- son qualifieur de portée
  - public,
  - package (en l'absence de qualifieur)
  - private (dans le cas des classes imbriquées)
- le package auquel elle appartient

Limitations

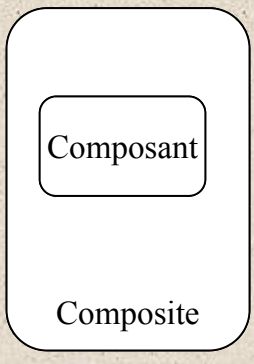
- Espace de protection non contrôlable (modulo le jar)
- Nommer toutes les classes peut devenir rédhibitoire



Classe imbriquée : principe

Une «inner» Classe est une classe dont la définition apparaît dans une autre classe

- Principe de l'imbrication de déclarations
  - Localisation de l'imbrication
    - Classe
    - Un block de la classe
  - Anonymat de la déclaration
    - La classe ne possède pas de nom explicite
  - Récursivité du principe
    - Développement répété du principe de l'imbrication
- Relation avec les autres mécanismes
  - Délégation
  - Héritage
  - Généricité



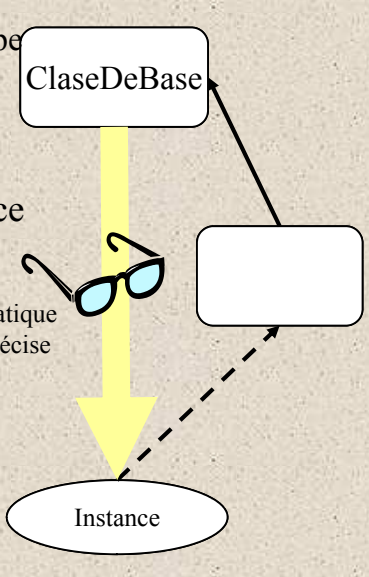
Classe anonyme

Une classe anonyme ne possède pas de nom

- On ne peut pas y faire référence
  - On ne peut pas typer statiquement une référence par le type correspondant.
  - Elle doit être génératrice car elle ne peut être un modèle
- Les seules méthodes applicables statiquement à Instance sont celles de ClasseDeBase

```
<ClasseDeBase> Instance = new <ClasseDeBase> (…) {
        … <méthodes redéfinies>
        … <méthodes spécifiques>
}
```

Vision statique la plus précise



- Utile quand on veut construire un singleton
  - Classe ayant au plus une instance
    - Exemple : while, for , …

Remarque : la classe de base peut être une interface

Objet représentant une méthode

Dans les modèles classiques (non tout objet) une méthode (un code exécutable) n'est pas considéré comme un objet et de ce fait n'est pas l'instance d'une classe

```
class Entier {
    …
    public int get(Entier i) {
        return …;
    }
}
```

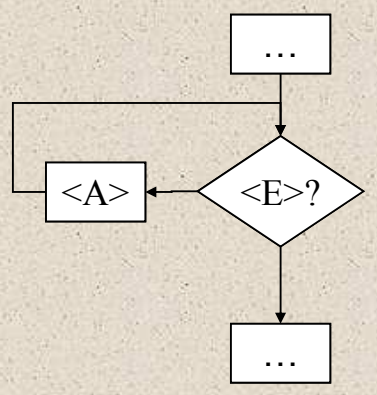
On peut toujours représenter une méthode par un objet dont la seule capacité est d'exécuter le code correspondant

```
class Entier {
    …
    public int get(Entier i) {
        return new Code().do(i);
    }
    …
}
```

```
class Code {
    public int do(int Valeur) {
        return …;
    }
}
```

La classe While

…;
while (<E>) <A> ;
… ;




```
public abstract class While {
        protected abstract boolean continuation();
        protected abstract void action();
        public void execute() {
                if(continuation()) {action(); execute();}
        }
}
```

Sémantique du while :
<E>==true : {while (<E>) <A> ;} <=> {<A>; while (<E>) <A> ;}
<E>==false : {while (<E>) <A> ;} <=> {}

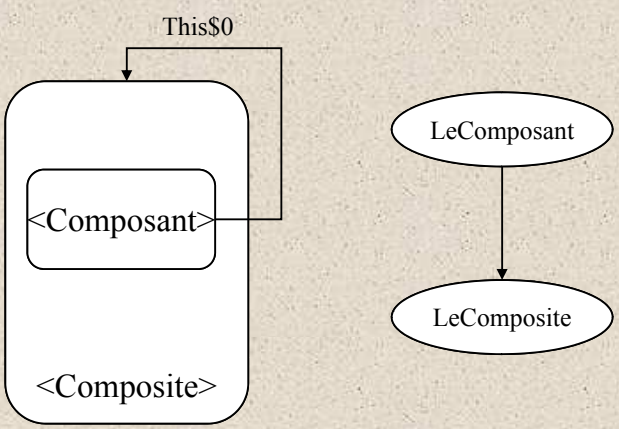
Objet While

```
class <X> {
  <T> <M>(…) {
        …;
        while (<E1>) <A1> ;
        … ;
        while (<E2>) <A2> ;
  …
  }
}
```

```
Class <X> {
   <T> <M>(…) {
      …
      new While() {
         protected boolean continuation() { return <E1>;}
         protected void action(){ <A1>}
         }.execute();
      …
      static final While w1 = new While() {
         protected boolean continuation() { return <E2>;}
         protected void action(){ <A2>}
      };
      w1.execute();
      …
   }
}
```


Classe imbriquée dans un bloc classe

```
class <Composite> {
   <attributs>
   class <Composant> {
      …
   }
   <méthodes>
}
```



Tout objet instance d'un Composant est **instancié** au sein d'un objet de la classe Composite
La forme générale est donc :
… leComposant = this.new <Composant>(…);
ou
<Composite> leComposite = new <Composite>(…);
… leComposant = LeComposite.new <Composant>();
La classe du composant peut être anonyme avec pour classe de base <Classe>

Classe imbriquée dans un bloc classe

```
class <Composiste>$<Composant> {
/* ACC_SUPER bit NOT set */
   private final <Composite> this$0;
   … <variables locales>
   <Composite>.<Composant>(<Composite>, …);
   … <méthodes redéfinies>
   … <méthodes spécifiques>
}
```

La classe a pour nom

De ce fait l'objet LeComposite constitue une partie de la fermeture de l'objet LeComposant.
Ceci constitue une agrégation où le composant n'existe qu'en présence du composite

```
class <Composite>$<#> extends <Classe> {
   private final <Composite> this$0;
   … <variables locales>
   <Composite>$<#> (<Composite >, …);
   …<méthodes redéfinies >
}
```

La forme de construction pour une classe anonyme est :
LeComposant = this.new <Classe>() { <définition>} ou
<Composite> leComposite = new <Composite>(…);
… leComposant = LeComposite.new <Classe>() { <définition>};

Exemple : Test

```
class Test  {
      private int a1;
      public Test(int v)  { a1 = v;}
      public class Test1  { public String toString()  { return "Test1:"+a1;} }

      public void run()  {
         Test.Test1 o1 = new Test.Test1();                    System.out.println(o1);
         Test t = new Test(6);        Test.Test1 o2 = t.new Test1();     System.out.println(o2);
      }

      static Test t1;
      static Test.Test1 o3;
      public static void main(String[] args)  {
         new Test(3).run();
         t1 = new Test(5);            o3 =t1.new Test1();                System.out.println(o3);
      }
}
```


Décompilation de Test

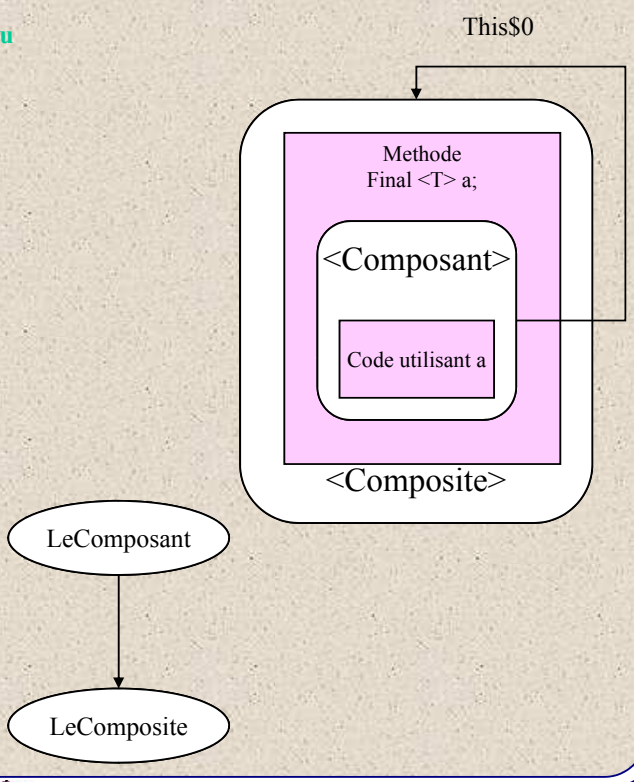
```
// DeCompiled : Test.class
import java.io.PrintStream;
class Test{
   public class Test1{
      public String toString(){ return "Test1:" + a1;}
      public Test1(){}
   }
   private int a1; static Test t1; static Test1 o3;
   public Test(int v){a1 = v;}
   public String toString(){ return "Test";}
   public void run(){
      Test1 o1 = new Test1(); System.out.println(o1);
      Test t = new Test(6);
      Test1 o2 = t. new Test1();System.out.println(o2);
   }
   public static void main(String args[]){
      (new Test(3)).run();
      t1 = new Test(5);
      o3 = t1. new Test1(); System.out.println(o3);
   }
}
```

```
class Test extends java.lang.Object {
   private int a1;
   static Test t1;
   static Test.Test1 o3;
   public Test(int);
   public java.lang.String toString();
   public void run();
   public static void main(java.lang.String[]);
   static int access$000(Test);

   public class Test.Test1 extends java.lang.Object
   /* ACC_SUPER bit NOT set */
   {
      private final Test this$0;
      public Test.Test1(Test);
      public java.lang.String toString();
   }
}
```


Classe imbriquée dans un bloc fonctionnel

Classe privée utilisant une variable ou un paramètre du contexte du bloc fonctionnel.

```
class <Composite> {
  <attributs>
  <T1><M1>(…) {
    final <T> a;
    class <Composant> {
      <T2><M2>(…) {
    }
  }
  …
}
```



L'objet LeComposite et la variable « a » constituent une partie de la fermeture de l'objet LeComposant. « a » doit être **final** pour garantir la cohérence de la fermeture avec le contexte.

Classe imbriquée dans un bloc fonctionnel

```
class <Composite>$1$<Composant> {
   private final <Composite> this$0;
   private final <T> val$<CompositeVariable>;
   … <variables locales>
   <Composite>$1$<Composant>(<Composite>,<T>);
   …<méthodes redéfinies> // utilisant <CompositeVariable>
   …<méthodes spécifiques> // utilisant <CompositeVariable>
}

class <Composite>$2 extends <Classes> {
   private final <Composite> this$0;
   private final <T> val $<CompositeVariable>;
   … <variables locales>
   <Composite>$2 (<Composite>,<T>);
   …<méthodes redéfinies> // utilisant <CompositeVariable>

}
```


Décompilation de Test1

```
// DeCompiled : Test1.class
import java.io.PrintStream;
class Test1{
   private int a1; static Test1 t1;
   public Test1(int v){a1 = v;}
   public String toString(){
      class Test11{
         public String toString(){return "Test11:" + a1;}
         Test11(){}
      }
      return (new Test11()).toString();
   }
   public void run(){
      Test1 t = new Test1(6);
   }
   public static void main(String args[]){
      (new Test1(3)).run();
      t1 = new Test1(5);
      System.out.println(t1);
   }
}
```

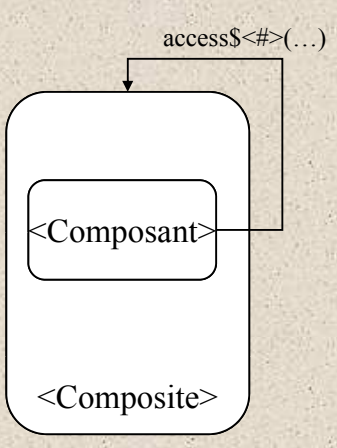
```
// Compiled from Test1.java
class Test1 extends java.lang.Object {
   private int a1;
   static Test1 t1;
   public Test1(int);
   public java.lang.String toString();
   public void run();
   public static void main(java.lang.String[]);
   static int access$000(Test1);

class Test1$1$Test11 extends java.lang.Object {
   private final Test1 this$0;
   Test1$1$Test11(Test1);
   public java.lang.String toString();
}
```


Classe imbriquée statique

Classe statique utilisant des variables statiques ou non du contexte.

Dans ce cas le <Composant> n'est plus associé à un <Composite>. Ce lien devient fonctionnel.

```
static class <Composite>. <Composant> {
   <Composite>.<Composant>();
}
```

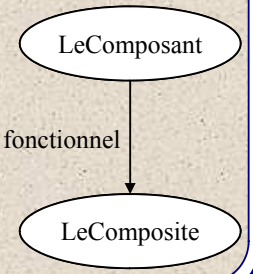


L'accès à une variable statique du composite est réalisé par une méthode spécifique de la forme :

static <T> access$<#>();

L'accès à une variable non statique du composite est réalisé par une méthode spécifique de la forme :

static <T> access$<#>(<Composite>);



Accès à la classe Composite

- Accès aux caractéristiques de la classe composite
  - En faisant référence à
    - <CompositeCaractéristique>
      - » a1 dans Test.Test1
    - <Composite>.this.<CompositeCaracteristique>
      - » Test.this.a1 dans Test.Test1
- Accès à la super-classe de la classe Composite
    - <Composite>.super.<SuperCompositeCaracteristique>
      - » Test.super.toString();
- Accès au super-constructeur d'une classe imbriquée depuis une classe fille extérieure
    - <LeComposite>.super(…);

L'exemple de la Tortue : Ordre

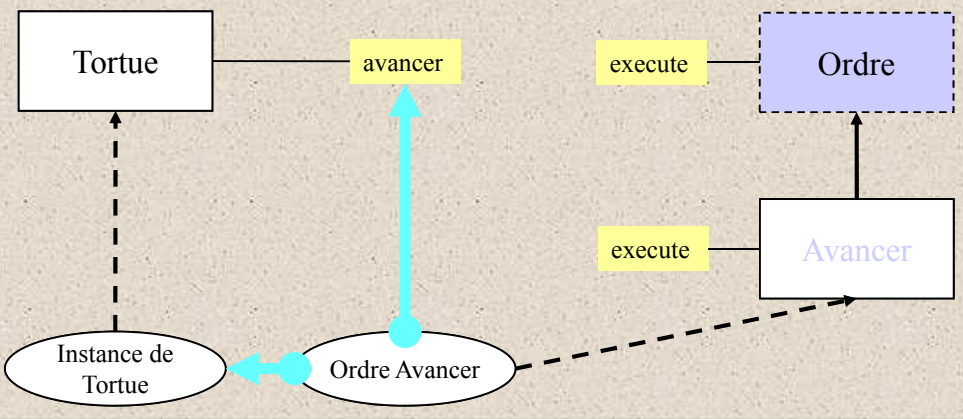
Afin de modéliser fidèlement la notion de tortue, nous introduisons le concept d'Ordre qui correspond à la description d'une opération réalisable par la tortue. Initialement, pour faire fonctionner la tortue, il fallait enficher un ordre dans un lecteur approprié.

Aglaé et les tribulations d'une tortue programmable

L'exemple de la tortue

Pour rendre programmable la tortue, il faut pouvoir mémoriser les ordres qu'elle doit exécuter.

- Réifier les méthodes de la tortue
  - Faire un objet qui représente l'action à entreprendre, il doit contenir
    - L'accès à l'objet destinataire
    - La méthode à appliquer à ce destinataire



La Tortue :

```
public class Turtle  {
   public Turtle(TurtleArea feuille, String imageFile) {…; image = new TurtleImage(imageFile); }
   public void avancer(int d) { … }
   public void reculer(int d) { …}
   public void droite(double a) { …}
   public void gauche(double a) { …}
   public void lever() {…}
   public void baisser() {…}
   public void allerA(Point p) {…}
   public final void tournerVers(double a) {…}
   …
   public TurtleImage image() {return image;}

   private TurtleArea feuille;
   private boolean estLeve;// état de la plume
   private Point position; // la position courante de la tortue
   private Vecteur cap;  // direction courante de la tortue
   private final TurtleImage image; // l'image de la tortue
   protected final Vecteur buter(double d, Vecteur v) {…}
}
public interface Ordre  { public  void execute();}

abstract class OrdreTortue implements Ordre  {}
```

La Tortue programmable

```
public class TurtleProgrammable extends Turtle  {
  private enum Mode {IM, DF};
  protected Mode mode = Mode.IM;
  protected List<OrdreTortue> programme = new LinkedList<OrdreTortue>();
  public TurtleProgrammable(TurtleArea feuille, String i){super(feuille,i);}
  public void modeDifere() {          mode =Mode.DF; programme.clear();}
  public void modeImmediat() {        mode = Mode.IM;}
  public void run() {
    for(OrdreTortue order : programme)  {order.execute();}
  }
  public void avancer(int d) {
    if (mode==Mode.IM) super.avancer(d);else programme.add(this.new Avancer(d));
  }
  public class Avancer extends OrdreTortue  {
    protected int d;
    public Avancer(int d)  { this.d=d;}
    public void execute()  {avancer(d);}
  }
```


La Tortue programmable

```
  public void droite(final double a) {
    class Droite extends OrdreTortue  {
      public void execute()  {droite(a);}
    }
    if (mode==Mode.IM) super.droite(a); else programme.add(new Droite() );
  }


  public void reculer(int d) {
    class Reculer extends OrdreTortue  {
      protected int d;
      public Reculer(int d)  {this.d=d;}
      public void execute() {TurtleProgrammable.super.reculer(d);}
    }
    if (mode==Mode.IM) super.reculer(d); else programme.add(new Reculer(d) );
  }
```


La Tortue programmable

```
public void lever() {
   if (mode==Mode.IM) super.lever();
   else programme.add(new OrdreTortue() {public void execute() { lever();}} );
}
public void gauche(final double a) {
   if (mode==Mode.IM) super.gauche(a);
   else programme.add(new OrdreTortue() {
                        public void execute() {TurtleProgrammable.super.gauche(a);} } );
}
private class Baisser extends OrdreTortue  {public void execute()  {baisser();}}
public void baisser() {
   if (mode==Mode.IM) super.baisser();
   else programme.add(TurtleProgrammable.this.new Baisser() );
}
public void allerA(final Point p) {
   if (mode==Mode.IM) super.allerA(p);
   else programme.add(new OrdreTortue()  {public void execute()  {allerA(p);} } );
}
}
```


La Tortue sérielle

Permettre de définir une relation d'ordre dans le domaine des tortues.

– Consiste à faire de la comparaison un opérateur manipulable intervenant dans des abstractions, par exemple le concept de tri.

Le principe de tri

– Le tri s'applique à une collection d'éléments de même nature pour laquelle il existe un relation d'ordre.

- Le fait que ce soit une collection permet d'énumérer tout les éléments de la collection
- Le fait qu'il existe une relation d'ordre permet de comparer deux éléments de la collection.

Réalisation d'un trieur de tableau (un cas de collection)

```
public class ArraySort  {
        public static void sort(Object[] a, Comparator c) ;
}
```


Spécification d'un comparateur

Interface Comparator

public interface **Comparator<T>**

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.In the foregoing description, the notation sgn(*expression*) designates the mathematical *signum* function, which is defined to return one of -1, 0, or 1 according to whether the value of *expression* is negative, zero or positive.

The implementor must ensure that sgn(compare(x, y)) == -sgn(compare(y, x)) for all x and y. (This implies that compare(x, y) must throw an exception if and only if compare(y, x) throws an exception.)

The implementor must also ensure that the relation is transitive: ((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

Finally, the implementor must ensure that compare(x, y)==0 implies that sgn(compare(x, z))==sgn(compare(y, z)) for all z.

It is generally the case, but *not* strictly required that (compare(x, y)==0) == (x.equals(y)). Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."

Method Summary

int **compare**(T o1, T o2)
Compares its two arguments for order.

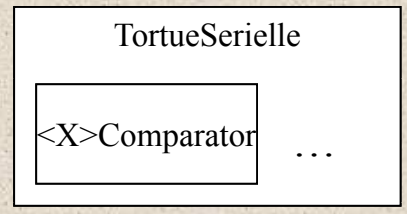
boolean **equals**(Object obj)
Indicates whether some other object is "equal to" this Comparator.

Un trieur

```
public class ArraySort {
   public static <T> void sort(T[] a, Comparator<T> c) {
      T[] aux = (T[])a.clone(); mergeSort(aux, a, 0, a.length, c);
   }
   private static void mergeSort(T[] src, T[] dest, int low, int high, Comparator c) {
    int length = high – low, i, j, mid;
    if (length < 7) {
       for (i=low; i<high; i++) for (j=i; j>low && c.compare(dest[j-1], dest[j])>0; j--) swap(dest, j, j-1);
       return;
    }
    mid = (low + high)/2; mergeSort(dest, src, low, mid, c); mergeSort(dest, src, mid, high, c);
    if (c.compare(src[mid-1], src[mid]) <= 0) { System.arraycopy(src, low, dest, low, length); return;}
    for(i = low, p = low, q = mid; i < high; i++) {
          if (q>=high || p<mid && c.compare(src[p], src[q]) <= 0) dest[i] = src[p++];
          else dest[i] = src[q++];
    }
   }
   private static <T> void swap(T[] x, int a, int b) { T t = x[a]; x[a] = x[b]; x[b] = t;}
}
```


La Tortue sérielle

```
public class TurtleSerielle extends TurtleProgrammable  {
 private int numero;
 public String toString()  { return ""+numero;}
 public TurtleSerielle(TurtleArea feuille, String imageFile,int numero) {
    super(feuille,imageFile);
    this.numero=numero;
 }
 public static class PositionTurtleComparator implements Comparator<TurtleSerielle>  {
    public int compare(TurtleSerielle t1, TurtleSerielle t2) {
       Point centre = new Point(Point.CARTESIEN,t1.feuille.getLargeur()/2, t1.feuille.getHauteur()/2);
       Vecteur v1 = new Vecteur(t1.position(),centre), v2 = new Vecteur(t2.position(),centre);
       if ( v1.norme()<v2.norme())  { return -1;
       }else if(v1.norme() == v2.norme()) { return 0;
       }else { return 1;
       }
    }
 }
```



La Tortue sérielle

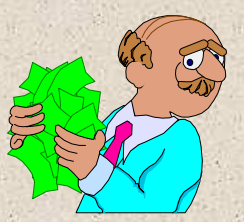
```
  protected static class AngleTurtleComparator implements Comparator<TurtleSerielle>  {
     public int compare(TurtleSerielle t1, TurtleSerielle t2) {
        if (t1.cap().module()<t2.cap().module())  {return -1;
        }else if(t1.cap().module()==t2.cap().module()) {return 0;
        }else {return 1;
        }
     }
  }
  static class NumeroTurtleComparator implements Comparator<TurtleSerielle>  {
     public int compare(TurtleSerielle t1, TurtleSerielle t2) {
        if (t1.numero<t2.numero)  {return -1;
        }else if(t1.numero==t2.numero) {return 0;
        }else {return 1;
        }
     }
  }
}
```


La Tortue sérielle

```
protected void StartTest()  {
    for(int i = nbTurtle; i>0;i--)  {
        turtles[i-1] = new TurtleSerielle(turtleArea,"turtle.gif",i);
        turtleArea.addTurtle(turtles[i-1]);
    }
    turtles1 = (TurtleSerielle[])turtles.clone();
    turtles2 = (TurtleSerielle[])turtles.clone();
    for(int i = (int)(Math.random()*10000); i>0;i--)  {
        if(Math.random()>0.25)  {
            turtles[(int)(Math.random()*nbTurtle)].avancer((int)(Math.random()*200)); nbOperation++;
        }
        if(Math.random()>0.35)  {
            turtles[(int)(Math.random()*nbTurtle)].droite((int)(Math.random()*360)); nbOperation++;
        }
    }
    ArraySort.sort(turtles,new TurtleSerielle.PositionTurtleComparator());
    ArraySort.sort(turtles1,new TurtleSerielle.AngleTurtleComparator());
    ArraySort.sort(turtles2,new TurtleSerielle.NumeroTurtleComparator());
    for(int i = 0;i<nbTurtle;i++)  {
        System.out.println(turtles2[i]+","+turtles[i]+","+turtles1[i]);
    }
    System.out.println(nbTurtle+" tortues traitées par "+nbOperation+" opérations");
}
```

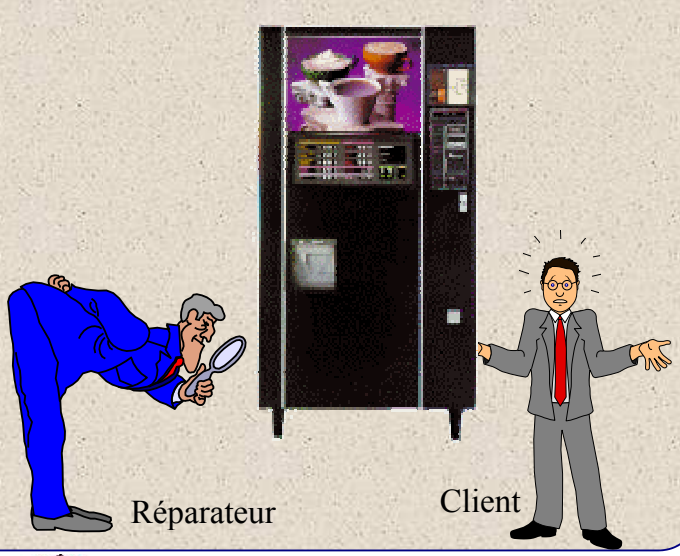

Vue

Une vue est une restriction des compétences d'un objet destinée à un type particulier d'utilisateurs



Exploitant

- Permet de fixer les compétences accessibles en fonction du type de l'utilisateur
- Les vues peuvent avoir des intersections non vides : une compétence peut être accessible à plusieurs types d'utilisateurs.
- La vue n'influe pas sur l'intégrité de l'objet, elle n'offre qu'une interface dédiée.



Réparateur

Client

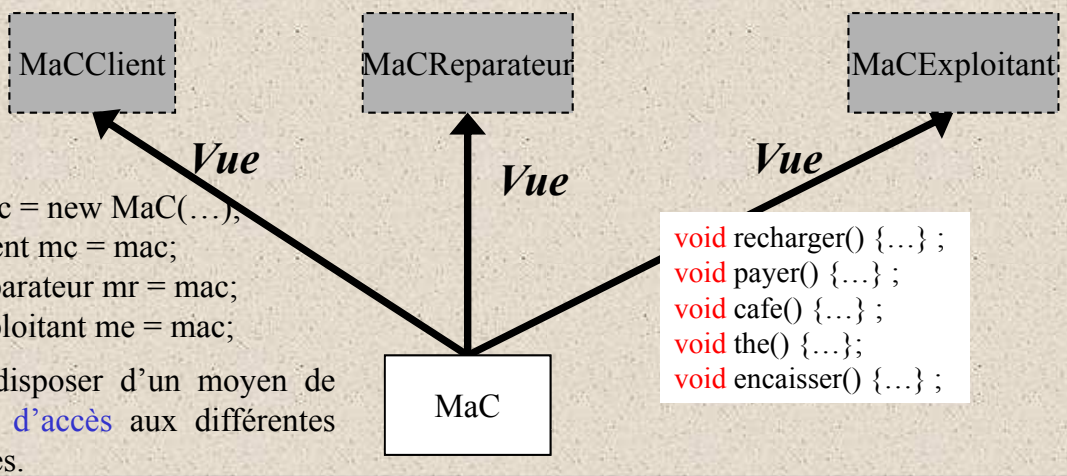
Vue : approche comportement

On définit les vues comme des abstractions de la classe représentant l'objet dans sa totalité.

```
absract void payer() ;
absract void cafe();
absract void the();
```

```
absract void recharger();
absract void payer();
absract void cafe() ;
absract void the();
```

```
absract void encaisser() ;
```




```
MaC mac = new MaC(…);
MaCClient mc = mac;
MaCReparateur mr = mac;
MaCExploitant me = mac;
```

Il faut disposer d'un moyen de contrôle d'accès aux différentes interfaces.

```
void recharger() {…} ;
void payer() {…} ;
void cafe() {…} ;
void the() {…};
void encaisser() {…} ;
```


Vue : réalisation par classes imbriquées

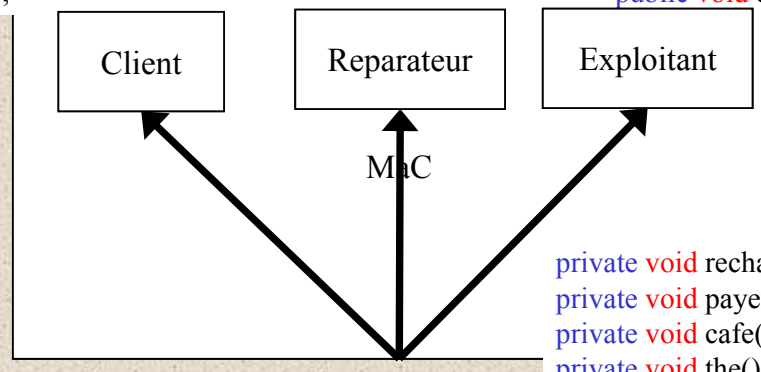
On définit les vues par des objets plénipotentiaires de l'objet réel.

```
public void payer() ;
public void cafe();
public void the();
```

```
public void recharger();
public void payer();
public void cafe() ;
public void the();
```

Il faut disposer d'un moyen de contrôle d'accès aux différentes interfaces.

```
public void encaisser() ;
```




```
private void recharger() {…} ;
private void payer() {…} ;
private void cafe() {…} ;
private void the() {…};
private void encaisser() {…} ;
```

```
MaC mac = new MaC(…);
MaC.Client mc = mac.createClient();
MaC.Reparateur mr = mac.createReparateur(1234);
MaC.Exploitant me = mac.createExploitant(5678);
MaC.Distributeur m = me;
```

La classe MaC

```
public class MaC {
  private int codeExploitant,codeReparateur;    private String nom;
  private String nom()          {return nom;}
  private void payer()          {System.out.println("payer");}
  private void cafe()           {System.out.println("café");}
  private void the()            {System.out.println("thé");}
  private void recharger()      {System.out.println("recharger");}
  private void encaisser()      {System.out.println("encaisser");}

  public MaC(int codeExploitant, int codeReparateur, String nom) {
    this.codeExploitant=codeExploitant;
    this.codeReparateur=codeReparateur; this.nom=nom;
  }
  public final Reparateur createReparateur(int code) throws Exception {
     if (code!=codeReparateur) throw new Exception();
     return new Reparateur(); else return null;
  }
  public final Exploitant createExploitant(int code) {
     if (code===codeExploitant) return new Exploitant(); else return null;
  }
  public final Client createClient() { return new Client();}
```

```
  private class User {
     public String nom()    {return MaC.this.nom();}
  }
  public class Reparateur extends User {
     private Reparateur () {}
     public void recharger()  {MaC.this.recharger();}
     public void payer()      {MaC.this.payer();}
     public void cafe()       {MaC.this.cafe();}
     public void the()        {MaC.this.the();}}
  public class Exploitant extends User {
     public Exploitant (int code) throws Exception {
        if(code!=codeExploitant) throw new Exception();
     }
     public void encaisser()  {MaC.this.encaisser();}}
  public class Client extends User {
     public Client() {}
     public void payer()      {MaC.this.payer();}
     public void cafe()       {MaC.this.cafe();}
     public void the()        {MaC.this.the();}}
}
```


Utilisation de la classe MaC

```
public class Client1 extends MaC.Client {

      public Client1(MaC MaC) {MaC.super();}

      public void payer()    {System.out.print("client1 : "); super.payer();}
      public void cafe()     {System.out.print("client1 : "); super.cafe();}
      public void the()      {System.out.print("client1 : "); super.the();}
}
```

```
MaC mine                    = new MaC(1234, 5678,"LaMienne");

MaC.Reparateur reparateur   = mine.createReparateur(5678);
MaC.Exploitant exploitant   = mine.new Exploitant(1234);
MaC.Client     client       = mine.new Client();
Client1        client 1     = new Client1(mine);
```