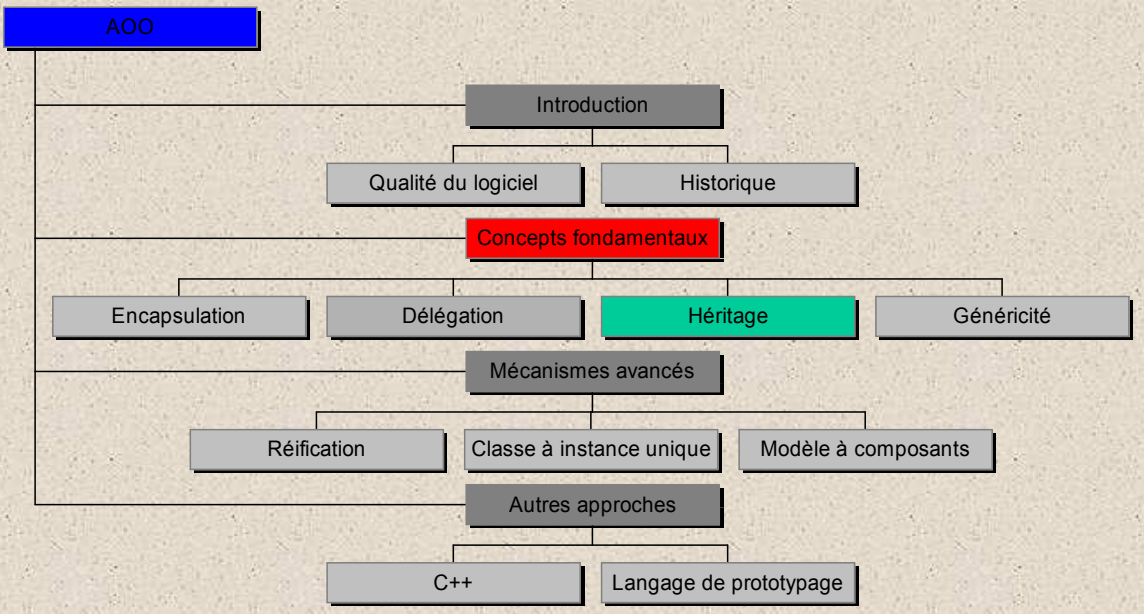
L'HERITAGE

HERITAGE & ABSTRACTION

Sommaire



Sommaire

Abstraction

L'abstraction est un paradigme qui permet de ne retenir d'une chose que l'essentiel, en faisant "abstraction" des éléments qui ne sont pas pertinents pour ce que l'on veut en faire.

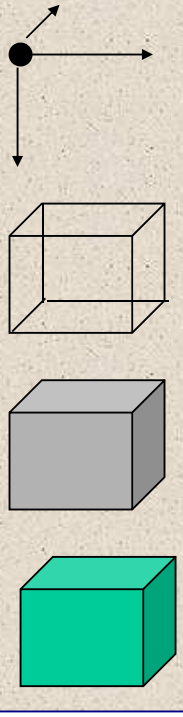
- Un utilisateur s'abstrait de la manière dont on fait une opération, pour ne retenir que le service offert.

L'abstraction est relative, il n 'y a pas une abstraction dans l'absolu

L'abstraction définit des niveaux relativement aux éléments dont on s'abstrait.

- La spécification est l'abstraction de la réalisation.

CUBE

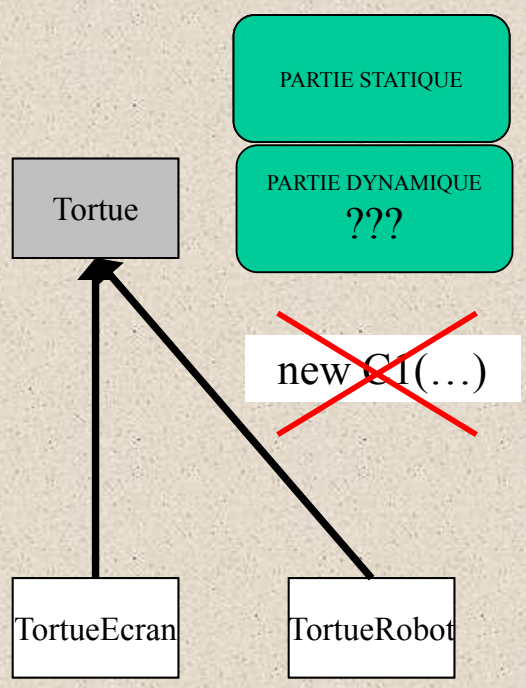


Classe abstraite

– Une classe abstraite est une classe dont le comportement n'est pas complètement défini, elle matérialise une spécification.
  - Certaines de ses méthodes sont abstraites (on s'abstrait de leurs réalisations).

– Elle ne peut pas être génératrice d'instance
  - Elle ne possède pas de constructeur directement applicable par new
  - Elle n'a d'intérêt que si elle est la racine d'un arbre d'héritage

– Utilisation des classes abstraites
  - Support d'abstraction
  - Support de généralisation → réutilisation
  - Support de propriété



TORTUE

```
/**
* modélise les fonctions d'une tortue Logo
* @author   P.Morat
* @version  1.0
* date : 1/9/99
* @invariant
* @motcle tortue, Logo, tracé, dessin
*/
public abstract class Tortue  {
      /**
       * Fait avancer la tortue de D pas
       * @param d la distance à parcourir
       * @require argumentValide : d>=0
       */
      public abstract  void avancer(int d);
      /**
       * Fait reculer la tortue de d pas
       * @param d la distance à parcourir
       * @require argumentValide : d>=0
       */
}
```

Une méthode, bien qu'abstraite, est spécifiable

```
public void reculer(int d)  {
    droite(180); avancer(d); gauche(180);
}
…
/**
* indique la situation de la plume
* @return <b>true</b>la plume est levée
*/
public abstract boolean estLevee();
/**
* Lève la plume
* @ensure PlumeLevee : estLevee()
*/
public abstract void lever() ;
/**
* Baisse la plume
* @ Ensure PlumeBaissee : ! estLevee()
*/
public abstract void baisser() ;
```

Une méthode, bien qu'abstraite, est utilisable

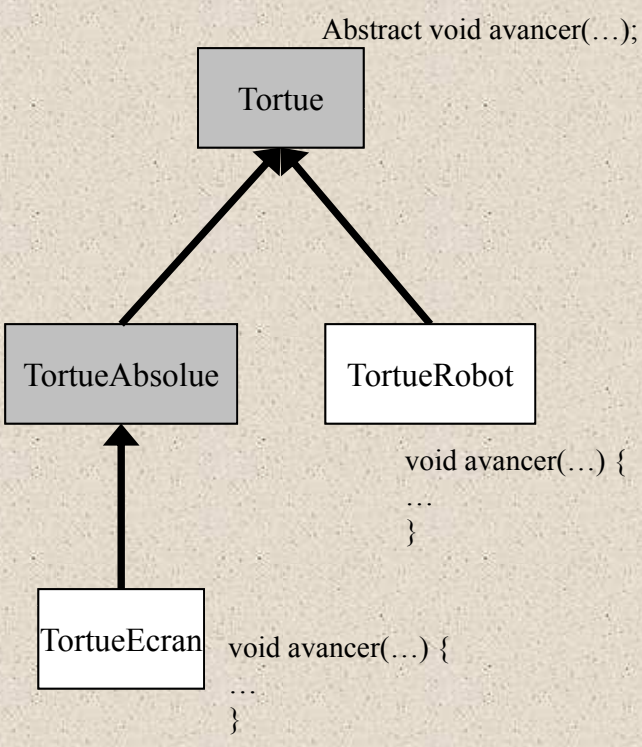
Classe abstraite & Héritage

Une classe abstraite doit avoir des descendants

– La classe abstraite défère (transfère la responsabilité de) la réalisation des abstractions à ces descendants

- La classe TortueRobot prend la responsabilité de la réalisation de avancer (abstraite dans Tortue).
- Une classe abstraite peut hériter d'une autre classe abstraite. Elle étend celle-ci et/ou la concrétise partiellement.

Une classe dont toutes les caractéristiques sont définies est une classe concrète



TortueEcran

```
/**
* modélise les fonctions d'une tortue Logo
* @author    P.Morat
* @version   1.0
* date : 1/9/99
* @invariant
* @motcle tortue, Logo, tracé, dessin
*/
public class TortueEcran extends TortueAbsolue
{
    /**
    * la position haute ou basse de la plume
    */
    private boolean estLeve;
    /**
    * Fait avancer la tortue de D pas
    * @param d la distance à parcourir
    * @require argumentValide : d>=0
    */
    public void avancer(int d) {
        …
    }
}
```

```
/**
* indique la situation de la plume
* @return <b>true</b> si la plume est levée
*/
public boolean estLevee() {
    return estLeve;}
/**
* Lève la plume
* @ensure PlumeLevee : estLevee()
*/
public void lever() {
    estLeve=true;}
/**
* Baisse la plume
* @ensure PlumeBaissee : ! estLevee()
*/
public void baisser() {
    estLeve=false;}
```

Interface

Une classe peut être partiellement ou totalement abstraite. Dans ce dernier cas, toutes les caractéristiques de la classe sont abstraites. Ceci constitue un cas particulier.

– En Java une classe totalement abstraite peut être modélisée par le concept d'interface qui est «incident» à celui de classe.


```
public interface Tortue  {
    /**
    * la valeur d'une unité de déplacement
    */
    public static final pas = 10;
    /**
    * Fait avancer la tortue de D pas
    * @param d la distance à parcourir
    * @require argumentValide : d>=0
    */
    public void avancer(int d);
    ...
                                                    }
```

Une interface ne possède pas de constructeur

Tout attribut est public et constant

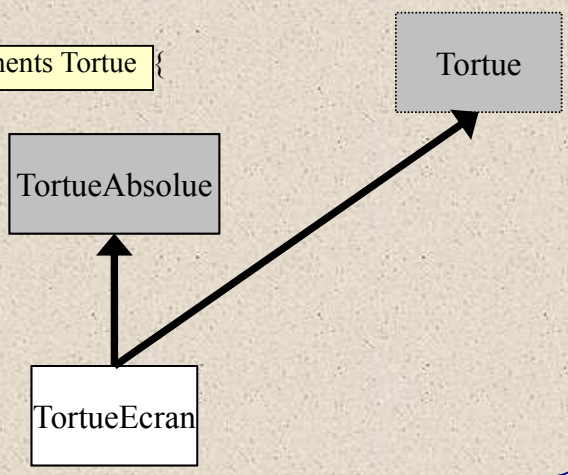
```
...
/**
* Lève la plume
* @ensure PlumeLevee : estLevee()
*/
public void lever() ;
/**
* Baisse la plume
* @ensure PlumeBaissee : ! estLevee()
*/
public void baisser() ;
```

Toute méthode est publique et abstraite

Classe concrète, abstraite et interface

– Une classe peut étendre une autre classe (qui peut être abstraite)
  • relation «extends» **Class x Class**
– Une classe «implante» une interface
  • relation «implements» **Class x Interface**
– Ces 2 relations sont des relations d'héritage
  • elles ont les mêmes propriétés.

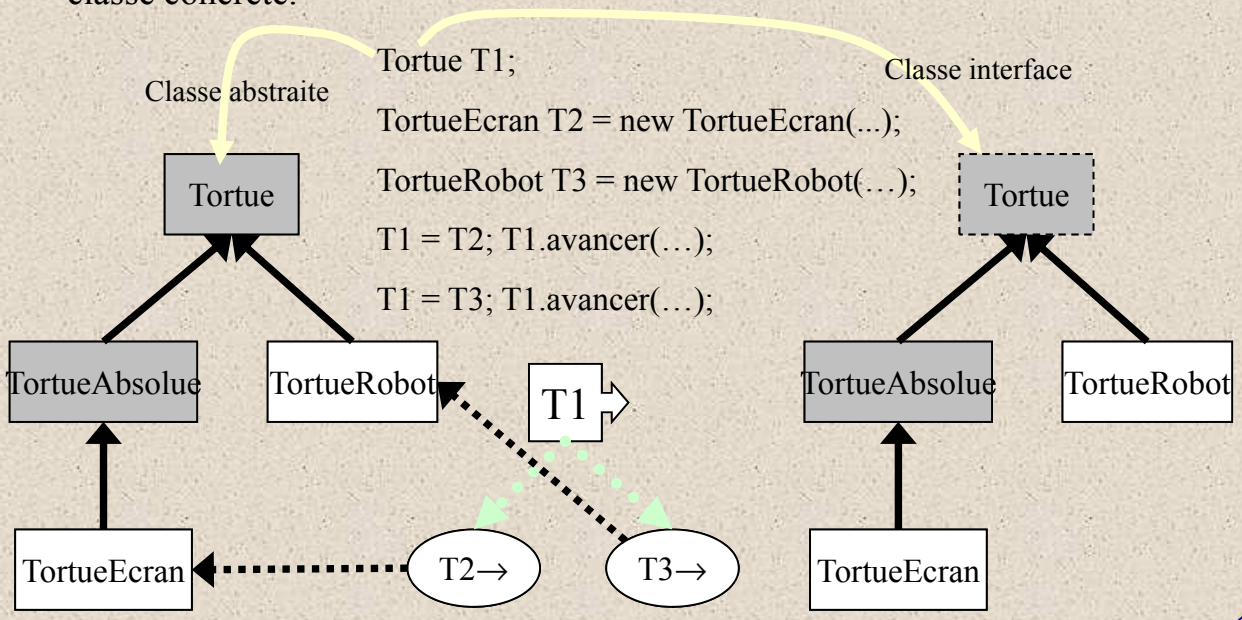
```
public class TortueEcran extends TortueAbsolue implements Tortue {
    /**
    * Fait avancer la tortue de d pas
    * @param d la distance à parcourir
    * @require argumentValide : d>=0
    */
    public void avancer(int d) {
    …
    }
    ...
```



Classe abstraite & polymorphisme

Une classe abstraite est un type possible pour une référence.

– Le polymorphisme s'applique donc dans les mêmes conditions que pour une classe concrète.



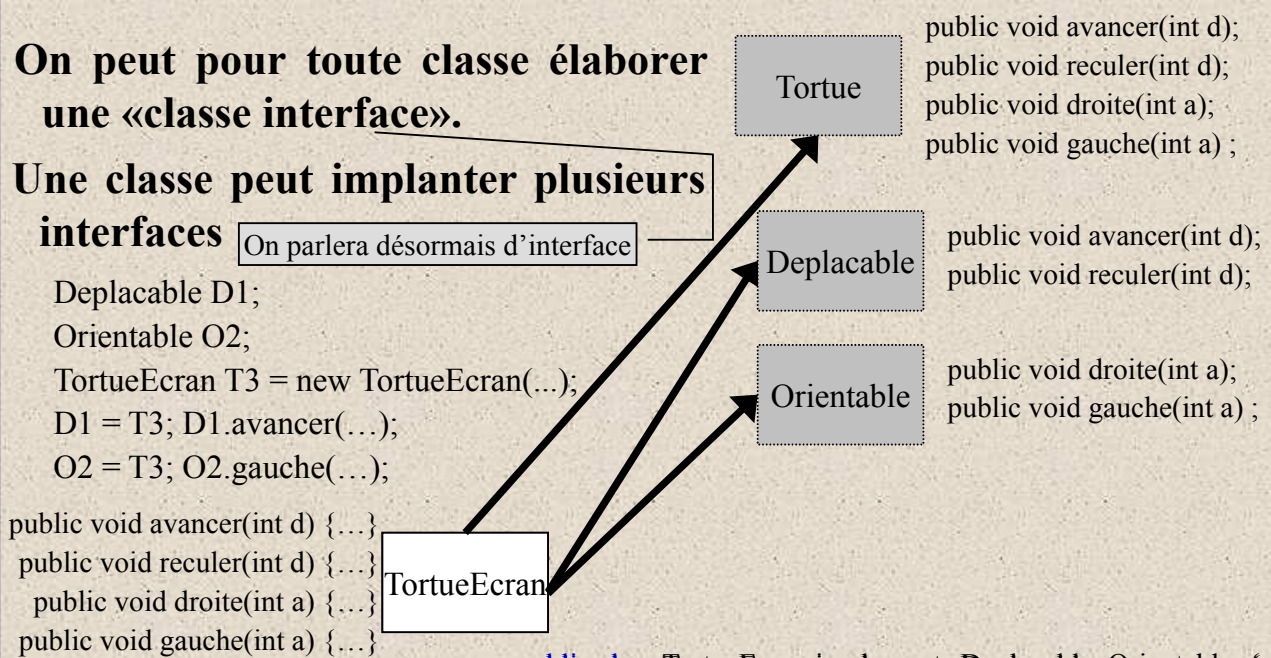
Interface de classe & classe Interface

Ne pas confondre la notion d'interface d'une classe avec le concept de «classe Interface»

On peut pour toute classe élaborer une «classe interface».

Une classe peut implanter plusieurs interfaces

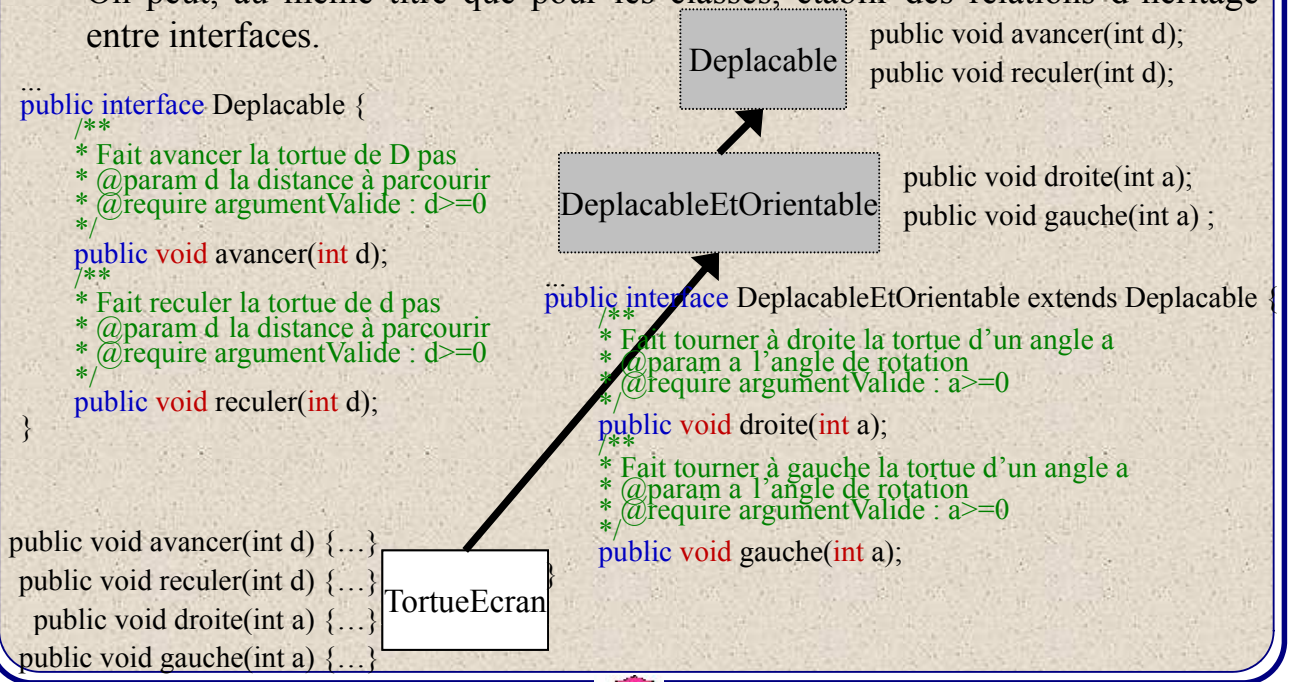
On parlera désormais d'interface



Interface & héritage

Les interfaces sont conceptuellement des classes abstraites

– On peut, au même titre que pour les classes, établir des relations d'héritage entre interfaces.




```
...
public interface Deplacable {
    /**
     * Fait avancer la tortue de D pas
     * @param d la distance à parcourir
     * @require argumentValide : d>=0
     */
    public void avancer(int d);
    /**
     * Fait reculer la tortue de d pas
     * @param d la distance à parcourir
     * @require argumentValide : d>=0
     */
    public void reculer(int d);
}
```

```
...
public interface DeplacableEtOrientable extends Deplacable {
    /**
     * Fait tourner à droite la tortue d'un angle a
     * @param a l'angle de rotation
     * @require argumentValide : a>=0
     */
    public void droite(int a);
    /**
     * Fait tourner à gauche la tortue d'un angle a
     * @param a l'angle de rotation
     * @require argumentValide : a>=0
     */
    public void gauche(int a);
```

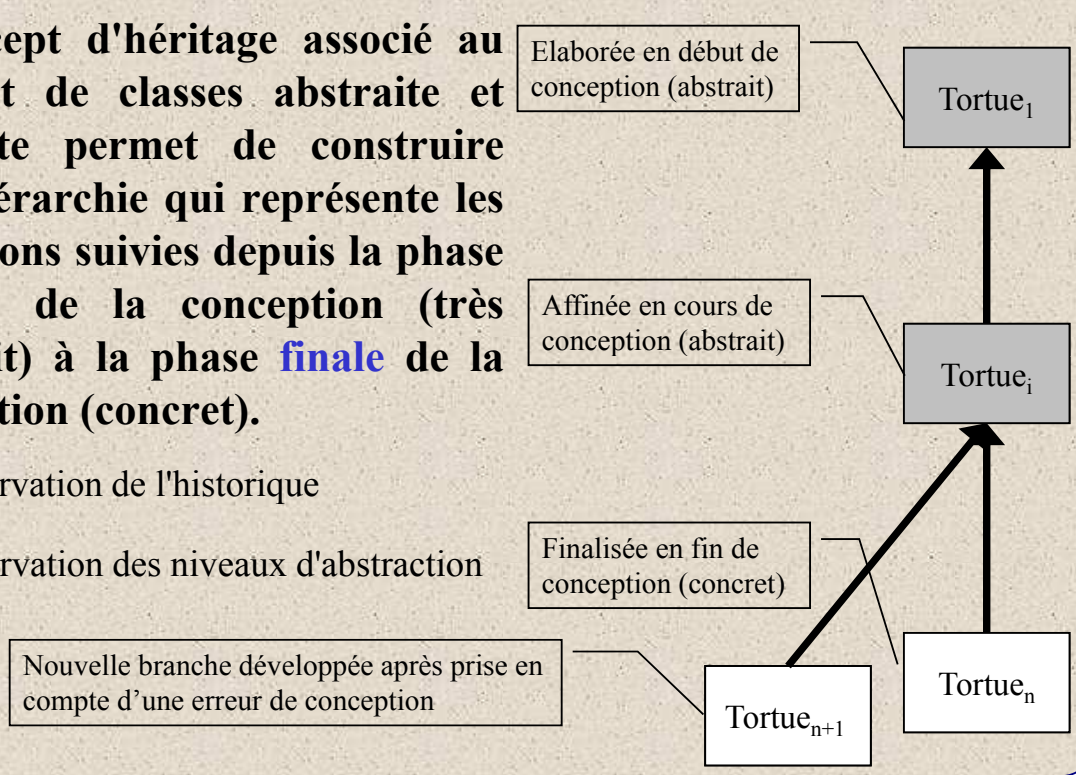
```
public void avancer(int d) {…}
public void reculer(int d) {…}
public void droite(int a) {…}
public void gauche(int a) {…}
```


Classe abstraite & Conception

Le concept d'héritage associé au concept de classes abstraite et concrète permet de construire une hiérarchie qui représente les évolutions suivies depuis la phase initiale de la conception (très abstrait) à la phase finale de la réalisation (concret).

– Conservation de l'historique

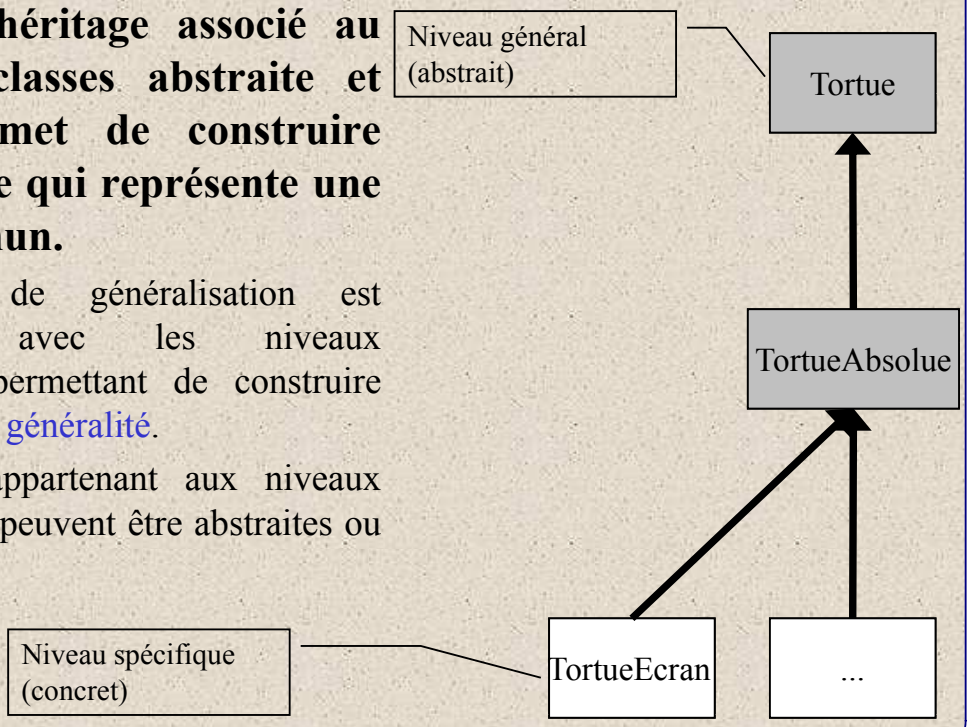
– Conservation des niveaux d'abstraction



Classe abstraite & Généralisation

Le concept d'héritage associé au concept de classes abstraite et concrète permet de construire une hiérarchie qui représente une mise en commun.

- Cet aspect de généralisation est combinable avec les niveaux d'abstraction permettant de construire des niveaux de généralité.
- Les classes appartenant aux niveaux intermédiaires peuvent être abstraites ou concrètes.



ORDRE

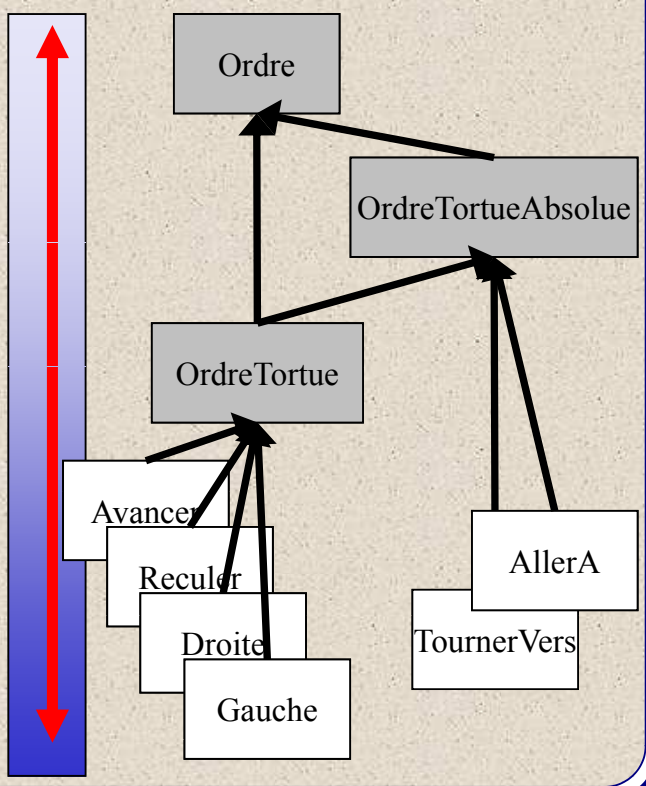
Afin de modéliser fidèlement la notion de tortue, nous introduisons le concept d'Ordre qui correspond à la description d'une opération réalisable par la tortue. Initialement, pour faire fonctionner la tortue, il fallait enficher un ordre dans un lecteur approprié.

Abstraction/Généralisation

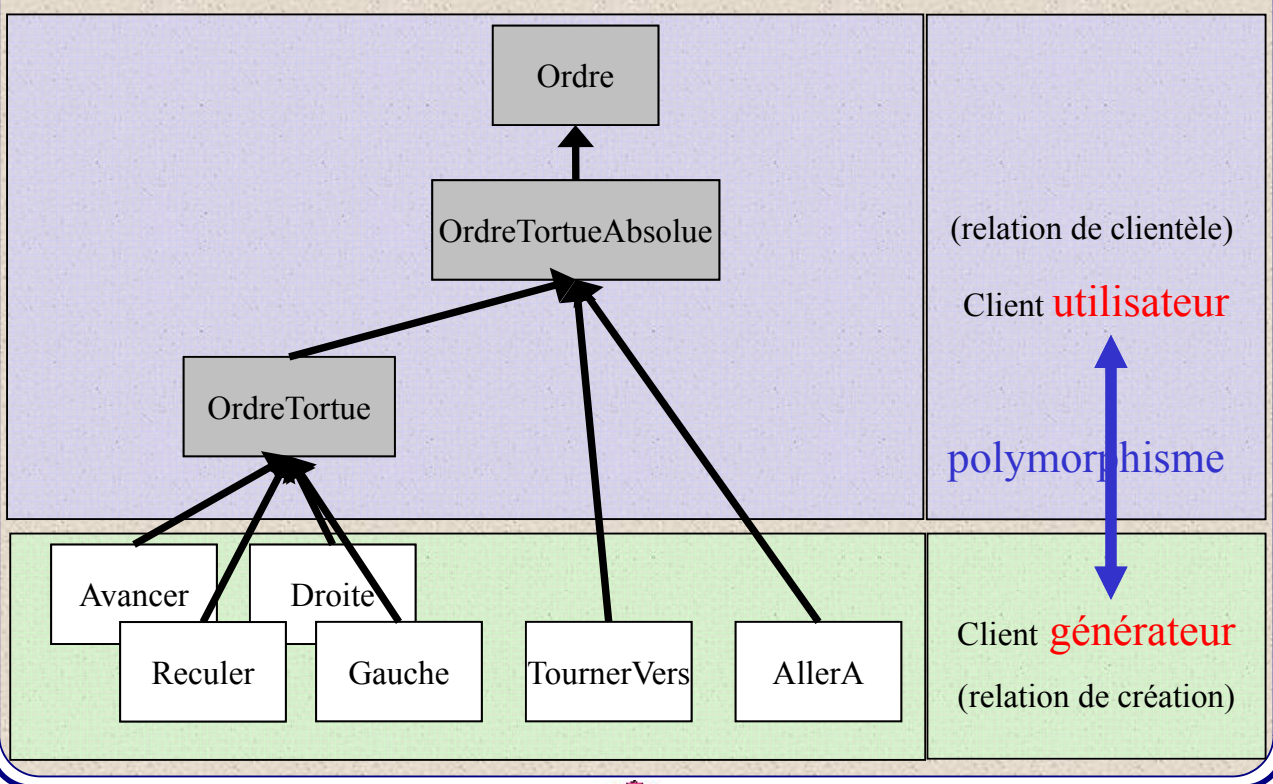
L'approche objet est performante de ce point de vue car elle n'oppose pas d'obstacle à la modélisation.

- Elle permet une modélisation descendante
- Elle permet une modélisation à posteriori (ascendante)

Permet de prendre en compte l'expérience du concepteur ou de l'entreprise



Utilisation d'une hiérarchie d'héritage



Définition de types énumérés

- Un type énuméré est un domaine dont les valeurs sont symboliquement énumérées et pour lesquelles il existe une relation d'ordre.
  - Le type Jour énumère tous les jours de la semaine
    - {LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE}
- Du fait de la structure d'ordre on peut obtenir le successeur d'une valeur si ce n'est pas la plus grande ou obtenir le prédécesseur si ce n'est pas la plus petite, comparer 2 valeurs d'un type énuméré, etc.
  - Successeur
    - MARDI succède à LUNDI
    - DIMANCHE n'a pas de successeur ou **lui-même** ou **LUNDI**
  - Prédécesseur
    - JEUDI précède VENDREDI
    - LUNDI n'a pas de prédécesseur ou **lui-même** ou **DIMANCHE**
  - Comparaison
    - LUNDI est plus petit que MARDI et que tous ceux plus grands que MARDI
- On peut aussi avoir des opérations de conversion entre types scalaires.

Les solutions

- Utiliser le type primitif "int" pour coder les valeurs
  - Absence de typage réel qui limite les contrôles de validité
- Construire une classe représentant le type énuméré
  - Le type int (attribut valeur) sera le support de représentation de la valeur énuméré
    - Fournit la relation d'ordre
  - L'égalité est assurée par l'équivalence (même valeur)
    - Deux instances du type ayant même valeur seront égales
  - Chaque type construit devra définir les opérateurs souhaités pour un type énuméré
    - fromInt : int -> <T> fromEnum : <T> -> int
    - Succ, pred : <Ti> -> <Tj>
  - Dénomination des valeurs
    - Une instance nommée par valeur du type énuméré
  - Gestion des noms symboliques.
    - Tabuler l'application liant un nom symbolique et une valeur (int <-> String)
      - » Soit au niveau de la classe
      - » Soit au niveau de l'instance, ce qui implique un seul exemplaire de la valeur et une égalité réduite à l'identité.
- Construire une hiérarchie de types avec un type abstrait racine

La spécification de Enum (Java 1.5)

```
public class Enum<E extends Enum<E> > extends Object implements Comparable<E>, Serializable {
        int          compareTo(E o)
                     Compares this enum with the specified object for order.
        boolean      equals(Object other)
                     Returns true if the specified object is equal to this enum constant.
        Class<E>     getDeclaringClass()
                     Returns the Class object corresponding to this enum constant's enum type.
        int          hashCode()
                     Returns a hash code for this enum constant.
        String       name()
                     Returns the name of this enum, exactly as declared in its enum declaration.
        int          ordinal()
                     Returns the ordinal of this enumeration constant (its position in its enum
                     declaration, where the initial constant is assigned an ordinal of zero).
        String       toString()
                      Returns the name of this enum constant, as contained in the declaration.
        static
        <T extends Enum<T>>
        T            valueOf(Class<T> enumType, String s)
                     Returns the enum constant of the specified enum type with the specified name.
}
```


L'énumération Fruit

```
public enum Fruit {FRAISE,ORANGE,POMME}
```

Version approchée

```
public final class Fruit extends Enum{
  public static final Fruit[] values(){return (Fruit[]) $VALUES.clone();}
  public static Fruit valueOf(String s){
    Fruit afruit[] = $VALUES; int i = afruit.length;
    for(Fruit fruit : $VALUES){if(fruit.name().equals(s)) return fruit;}
    throw new IllegalArgumentException(s);
  }
  private Fruit(String s, int i){super(s, i);}
  public static final Fruit FRAISE;
  public static final Fruit ORANGE;
  public static final Fruit POMME;
  private static final Fruit[] ENUM$VALUES;
  static {
    FRAISE = new Fruit("FRAISE", 0);
    ORANGE = new Fruit("ORANGE", 1);
    POMME = new Fruit("POMME", 2);
    ENUM$VALUES = (new Fruit[] {FRAISE, ORANGE, POMME});
  }
}
```


Exemple d'énumération

```
public class Test {
  public enum Carte {TREFFLE,CARREAU,COEUR,PIQUE};
  public enum Couleur {ORANGE,BLEU,VERT,JAUNE,ROUGE};

  public static void test () {
    Carte c = Carte.COEUR;
    // impression de la valeur du type et de son rang dans le type
    System.out.println("c="+c+", ordinal="+c.ordinal());
    // conversion explicite par le rang d'un type dans un autre
    Couleur cc = Couleur.values()[c.ordinal()];
    System.out.println("cc="+cc+", ordinal="+cc.ordinal());
    // conversion explicite par le nom d'un type dans un autre
    cc = Couleur.valueOf(Fruit.ORANGE.toString());
    System.out.println("cc="+cc+", ordinal="+cc.ordinal());
    // énumeration des valeur du type
    for(Fruit e : Fruit.values()){System.out.println(e);}
  }
}
```


Exemple d'énumération

```
public enum Couleur {

  ORANGE(251,184,45),
  BLEU(0,0,255),
  VERT(0,255,0),
  JAUNE(255,255,0),
  ROUGE(255,0,0);

  private byte r, v, b;
  private Couleur(int r, int v, int b){
    this.r=(byte)r; this.v=(byte)v; this.b=(byte)b;
  }
  public String toHtml(){
    return new Formatter().format("#%02X%02X%02X",new Object[]{r,v,b}).toString();
  }
  public int toInt(){
    return (((256+r)&255)<<16)+(((256+v)&255)<<8)+((256+b)&255);
  }
}
```