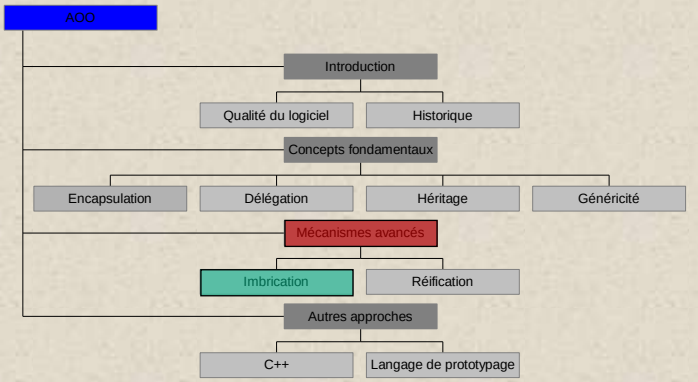
Persistance

« Externalisation »

Sommaire



Introduction

Intérêts

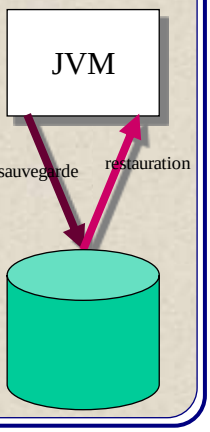
- Conserver des objets au-delà de la session qui les a créés
  - Persistance plus ou moins durable et plus ou moins sophistiquée
- Assurer un passerelle vers d'autres mondes ou d'autres espaces de "nommage"
  - Représentation pivot et/ou changement d'espace d'adressage

Critères

- Durée
  - Temporaire (ne sert que le temps d'un transfert ou d'une passivation)
  - Persistent (constitue une mémoire rémanente)
  - Durable (constitue une mémoire supportant les évolutions)
- Format
  - Propriétaire (format spécifique non exploitable directement)
  - Ouvert (format accessible, voire modifiable extérieurement)
- Mode
  - Structurel (c'est l'état qui est sauvegardé)
  - Opérationnel(c'est le programme construisant l'état qui est sauvegardé)
- Type
  - Invasif (nécessite une modification du code) ou non

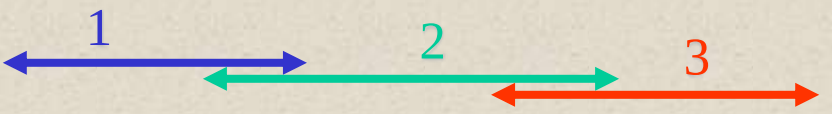
Modèles

- Sérialisation Java – Long Bean Persistence – JDO



Persistance : modes de représentation

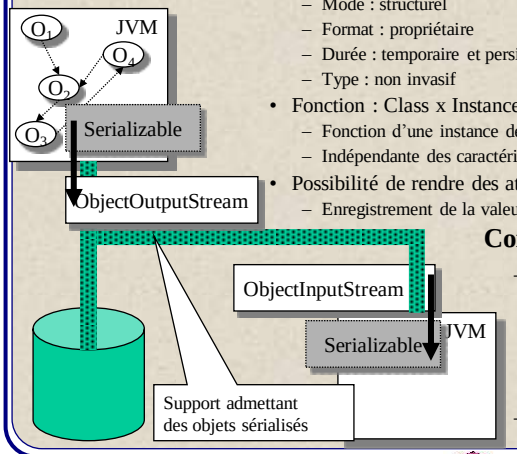
1. On sauvegarde l'état (mémoire) de l'objet
   - Avec @s translatables pour les références
   - Cette solution est restrictive :
     - Toute implantation de JVM doit respecter le même modèle d'implantation
       - » Taille (JVM 32bits vs JVM 64bits), offset des champs, …
2. Améliorations : donner de la souplesse au système de manière à :
   - Rester indépendant du niveau d'implantation (hors spécification)
   - Admettre des variations
     - Version de classe : Ascendante et Descendante
3. On mémorise le programme qui construit cet état
   - Sous la forme d'un texte source
   - Sous une forme plus synthétique



Sérialisation Java

Mettre sous une forme externe indépendante une composante connexe d'objets, soit pour la stocker, soit pour la transférer.

– Propriétés :
- Opération implicite pouvant être adaptée par redéfinition
  – Mode : structurel
  – Format : propriétaire
  – Durée : temporaire et persistent
  – Type : non invasif
- Fonction : Class x Instance -> Externe
  – Fonction d'une instance de n'importe quel type fournissant sa forme externe
  – Indépendante des caractéristiques de portées
- Possibilité de rendre des attributs non sérialisables (transient)
  – Enregistrement de la valeur par défaut du type correspondant (primitif ou non)



Conventions :

– Type primitif : représentation implicite
  – int, long, short, byte, double, float : 0
  – boolean : false
  – character : <blanc>
  – …

– Type non primitif : référence : null

Sérialisation : Serializable

public abstract interface **Serializable**

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable in this case. The error will be detected at runtime.

During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

When traversing a graph, an object may be encountered that does not support the Serializable interface. In this case the NotSerializableException will be thrown and will identify the class of the non-serializable object.

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

```
private void writeObject(java.io.ObjectOutputStream out),throws IOException
private void readObject(java.io.ObjectInputStream in)  throws IOException, ClassNotFoundException;
```

Sérialisation : ObjectOutputStream

public class **ObjectOutputStream**

extends OutputStream

implements ObjectOutput, ObjectStreamConstants

An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream. The objects can be read (reconstituted) using an ObjectInputStream. Persistent storage of objects can be accomplished by using a file for the stream. If the stream is a network socket stream, the objects can be reconsituted on another host or in another process.

Only objects that support the java.io.Serializable interface can be written to streams. The class of each serializable object is encoded including the class name and signature of the class, the values of the object's fields and arrays, and the closure of any other objects referenced from the initial objects.

The method **writeObject** is used to write an object to the stream. Any object, including Strings and arrays, is written with writeObject. Multiple objects or primitives can be written to the stream. The objects must be read back from the corresponding ObjectInputstream with the same types and in the same order as they were written.

For example to write an object that can be read by the example in ObjectInputStream:

```
FileOutputStream ostream = new FileOutputStream("t.tmp");
ObjectOutputStream p = new ObjectOutputStream(ostream);

p.writeInt(12345);
p.writeObject("Today");
p.writeObject(new Date());

p.flush();
ostream.close();
```


Sérialisation : ObjectInputStream

public class **ObjectInputStream**

extends InputStream

implements ObjectInput, ObjectStreamConstants

An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream. ObjectOutputStream and ObjectInputStream can provide an application with persistent storage for graphs of objects when used with a FileOutputStream and FileInputStream respectively. ObjectInputStream is used to recover those objects previously serialized. Other uses include passing objects between hosts using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system.

ObjectInputStream ensures that the types of all objects in the graph created from the stream match the classes present in the Java Virtual Machine. Classes are loaded as required using the standard mechanisms.

Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read from streams. The method **readObject** is used to read an object from the stream. Java's safe casting should be used to get the desired type. In Java, strings and arrays are objects and are treated as objects during serialization. When read they need to be cast to the expected type.

Primitive data types can be read from the stream using the appropriate method on DataInput.

Reading an object is analogous to running the constructors of a new object. Memory is allocated for the object and initialized to zero (NULL). No-arg constructors are invoked for the non-serializable classes and then the fields of the serializable classes are restored from the stream starting with the serializable class closest to java.lang.object and finishing with the object's most specifiec class.

For example to read from a stream as written by the example in ObjectOutputStream:

```
FileInputStream istream = new FileInputStream("t.tmp");
ObjectInputStream p = new ObjectInputStream(istream);

int i = p.readInt();
String today = (String)p.readObject();
Date date = (Date)p.readObject();

istream.close();
```


Serialisation : la classe Rectangle

```
import java.io.*;

public class Rectangle implements Serializable {

   protected int x, y, height,  width;
   public Rectangle()  {this(0,0,0,0);}
   public Rectangle(int height, int width)  {this(0,0,height,width);}
   public Rectangle(int x, int y, int height, int width)  {
      this.height=height; this.width=width; this.x=x; this.y=y;
   }
   public int x()  { return x;}
   public int y()  { return y;}
   public int height()  { return height;}
   public int width()  { return width;}
   public void x(int x)  { this.x = x;}
   public void y(int y)  { this.y = y;}
   public void height(int height)  { this.height = height;}
   public void width(int width)  { this.width = width;}
   public double surface()  {return  height*width;}
   public double perimetre()  {return  (height+width)*2;}
   public String toString()  {return  getClass().getName()+"{"+x+","+y+"}["+height+","+width+"]";}
}
```


Serialisation : la classe Rectangle1

```
import java.io.*;

/** @author morat
* On prend en charge les fonctions de sauvegarde et restauration en incluant
* la date de sauvegarde.
*/

public class Rectangle1 extends Rectangle {

   public Rectangle1() {super();}
   public Rectangle1(int height, int width) {super(height, width);}
   public Rectangle1(int x, int y, int height, int width) {super(x, y, height, width);}

   private void writeObject(java.io.ObjectOutputStream stream) throws IOException{
      stream.writeObject(new java.util.Date());
      stream.defaultWriteObject();
   }
   private void readObject(ObjectInputStream stream)throws IOException, ClassNotFoundException{
      System.out.println(" save at "+(java.util.Date)stream.readObject());
      stream.defaultReadObject();
   }
}
```


Serialisation : la classe Rectangle2

```
/**@author morat
*  mise en place de champs non sauvegardés.
*/
public class Rectangle2 extends Rectangle {

   /**  comptabilise le nombre de modifications*/
   protected transient int chgt =0;
   private transient int unsavable =0;

   public Rectangle2() {this(0,0,0,0);}
   public Rectangle2(int height, int width) {this(0, 0, height, width);}
   public Rectangle2(int x, int y, int height, int width) {super(x, y, height, width);}

   public void x(int x)  { chgt++; unsavable--; super.x(x);}
   public void y(int y)  { chgt++; unsavable--; super.y(y);}
   public void height(int height)  { chgt++; unsavable--; super.height(height);}
   public void width(int width)  { chgt++; unsavable--; super.width(width);}
   public String toString()  {return  super.toString()+"<"+chgt+","+unsavable+">";
}
```


Serialisation : la classe Rectangle22

```
import java.io.IOException;
/**@author morat
*  Le champ "chgt" qui est "transient" dans Rectangle2 est sauvegardé et restauré, on lève
*  par programmation le qualifieur "transient", ceci est impossible sur l'attribut
*  "unsavable" du fait qu'il est "private".
*/

public class Rectangle22 extends Rectangle2 {
   public Rectangle22() {super();}
   public Rectangle22(int height, int width) {super(height, width);}
   public Rectangle22(int x, int y, int height, int width) {super(x, y, height, width);}

   private void writeObject(java.io.ObjectOutputStream stream) throws IOException{
      stream.writeObject(""+chgt);
      stream.defaultWriteObject();
   }
   private void readObject(.ObjectInputStream stream)throws IOException, ClassNotFoundException
      chgt = Integer.parseInt((String)stream.readObject());
      stream.defaultReadObject();
   }
}
```


Serialisation : la classe Rectangle3

```
import java.awt.Color;
import java.io.*;
/**@author morat
*On définit les champs sauvegardés en extension, ne concerne que les champs déclarés
*/
public class Rectangle3 extends Rectangle {
   private static final ObjectStreamField[] serialPersistentFields= {
      new ObjectStreamField("lineColor", Color.class),
   };
   protected Color lineColor, fillColor;
   public Rectangle3()  {this(0,0,0,0);}
   public Rectangle3(int height, int width)  {this(0,0,height,width);}
   public Rectangle3(int x, int y, int height, int width)  {
      super(x,y,height,width);
      lineColor = Color.BLUE;
      fillColor = Color.RED;
   }
   public String toString()  {return  super.toString()+"<"+lineColor+","+fillColor+">";}
}
```


Serialisation : la classe Rectangle4

```
import java.io.ObjectStreamException; import java.io.Serializable;
/**@author morat  : On sérialise un Rectangle4 par un objet instance de Coord si les entiers ont
 *  des valeurs absolues inférieures à (2^16)-1*/
public class Rectangle4 extends Rectangle {
   private static final int SIZ = 16,  int MAX = (2<<SIZE)-1, SUP = 0x0000FFFF;
   private static final int abs(int v){ return Math.abs(v);}
   public Rectangle4() {this(0,0,0,0);}
   public Rectangle4(int height, int width) {this(0,0,height, width);}
   public Rectangle4(int x, int y, int height, int width) {super(x, y, height, width);}
   private static class Coord implements Serializable {
      private int c1, c2 ;
      private Coord(int x, int y, int height, int width){
         c1=(x << SIZ) | (y & SUP); c2=(height << SIZ) | (width &SUP);
      }
      private Object readResolve() throws ObjectStreamException{
         return new Rectangle4((c1>>SIZ),(c1<<SIZ>>SIZ),(int)(c2>>SIZ),(int)(c2<<SIZ>>SIZ));
      }
   }
   private Object writeReplace() throws ObjectStreamException{
      if(abs(x)<MAX && abs(y)<MAX && abs(height)<MAX && abs(width)<MAX)
         return new Coord(x,y,height,width); else return this;}
}
```

Serialisation : la classe Test

```
import java.io.*;
/**@author morat*/
public class Test {
   static ObjectOutputStream oos;
   static ObjectInputStream ois;
   public static Object restore() throws Exception {
      ois = new ObjectInputStream(new FileInputStream("t.tmp"));
      Object r = ois.readObject();
      ois.close();
      return r;
   }
   public static void save(Object obj) throws Exception {
      oos = new ObjectOutputStream(new FileOutputStream("t.tmp"));
      oos.writeObject(obj);
      oos.close();
   }
   public static void saveAndrestore(Rectangle r) throws Exception{
      System.out.print(r); save(r);
      System.out.print("\tsize(t.tmp)="+new FileInputStream("t.tmp").available());
      System.out.println("\t"+restore());
   }
```


Serialisation : la classe Test

```
public static void main(String[] args) throws Exception{
   saveAndrestore(new Rectangle(10,20,30,40));
   saveAndrestore(new Rectangle1(10,20,30,40));
   Rectangle2 r = new Rectangle2(10,20,30,40); r.width(50); System.out.print(r); save(r);
      System.out.print("\tsize(t.tmp)="+new FileInputStream("t.tmp").available()+"\t"+restore());
   Rectangle22 r = new Rectangle22(10,20,30,40); r.width(50); System.out.print(r); save(r);
      System.out.print("\tsize(t.tmp)="+new FileInputStream("t.tmp").available()+"\t"+restore());
   saveAndrestore(new Rectangle3(10,20,30,40));
   saveAndrestore(new Rectangle4(0,-2085,3000,-40));
   saveAndrestore(new Rectangle4(0,-2085,300000,-40));
}
```

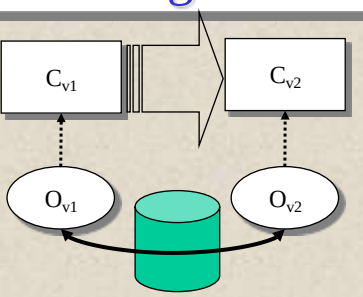
```
Rectangle{10,20}[30,40]                size(t.tmp)=71    Rectangle{10,20}[30,40]
Rectangle1{10,20}[30,40]               size(t.tmp)=139   save at Sun Feb 01 23:18:13 CET 2004Rectangle1{10,20}[30,40]
Rectangle2{10,20}[30,40]<1,-1>         size(t.tmp)=96    Rectangle2{10,20}[30,40]<0,0>
Rectangle22{10,20}[30,40]<1,-1>        size(t.tmp)=127   Rectangle22{10,20}[30,40]<1,0>
Rectangle3{10,20}[30,40]<java.awt.Color[r=0,g=0,b=255],java.awt.Color[r=255,g=0,b=0]>
                                       size(t.tmp)=252
                                                         Rectangle3{10,20}[30,40]<java.awt.Color[r=0,g=0,b=255],null>
Rectangle4{0,-2085}[3000,-40]          size(t.tmp)=55    Rectangle4{0,-2085}[3000,-40]
Rectangle4{0,-2085}[300000,-40]        size(t.tmp)=96    Rectangle4{0,-2085}[300000,-40]
```


Sérialisation : Versioning

– Evolutions non admises
  - Changer le type d'un champ
  - Rendre non sérialisable qqc
  - *Supprimer un champ ?*
  - …

– Evolutions admises
  - Ajouter un champ
  - Changer la portée d'un champ
  - …

– Principes
  - Le stockage par défaut des champs est associatif (nom-valeur), sinon l'ordre doit être respecté
  - Le stream d'une classe comporte les "required data" correspondant au champs sérialisées et les "optional data" qui sont des informations complémentaires



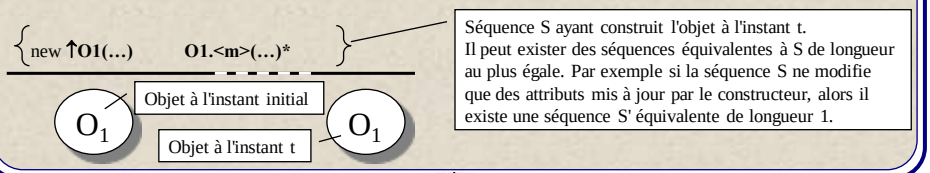
fonction de transfert permettant l'acquisition d'une instance de version ascendante ou descendante

Serialisation : Stream Unique Identifier

- **The class name written using UTF encoding.**
- **The class modifiers written as a 32-bit integer.**
- **The name of each interface sorted by name written using UTF encoding.**
- **For each field of the class sorted by field name (except private static and private transient fields):**
  a. The name of the field in UTF encoding.
  b. The modifiers of the field written as a 32-bit integer.
  c. The descriptor of the field in UTF encoding
- **If a class initializer exists, write out the following:**
  a. The name of the method, <clinit>, in UTF encoding.
  b. The modifier of the method, java.lang.reflect.Modifier.STATIC, written as a 32-bit integer.
  c. The descriptor of the method, ()V, in UTF encoding.
- **For each non-private constructor sorted by method name and signature:**
  a. The name of the method, <init>, in UTF encoding.
  b. The modifiers of the method written as a 32-bit integer.
  c. The descriptor of the method in UTF encoding.
- **For each non-private method sorted by method name and signature:**
  a. The name of the method in UTF encoding.
  b. The modifiers of the method written as a 32-bit integer.
  c. The descriptor of the method in UTF encoding.
- **The SHA-1 algorithm is executed on the stream of bytes produced by DataOutputStream and produces five 32-bit values sha[0..4].**
- **The hash value is assembled from the first and second 32-bit values of the SHA-1 message digest. If the result of the message digest, the five 32-bit words H0 H1 H2 H3 H4, is in an array of five int values named sha, the hash value would be computed as follows:**

Long Bean Persistence

- Utilisation du format XML garantissant une certaine standardisation et lisibilité externe.
- Propriétés :
  - Opération semi-implicite pouvant être adaptée par annotation
    - Mode : opérationnel (sous forme XML)
    - Format : ouvert
    - Durée : durable
    - Type : non invasif
  - Fonction : Class x Object -> Externe
    - Fonction d'une instance de n'importe quel type fournissant sa forme externe
      - » Représentation dépendante des opérations publiques du type.
      - » Représentation indépendante de la représentation interne du type



LBP : équivalence

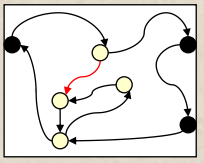
Equivalence de séquences :

Soit S une séquence interne = {S_0, S_1, …, S_n} ayant établi l'objet en l'état t. Il existe une séquence S' équivalente restituant un objet dans ce même état si :

- S' = {S'_0, .., S'_m} telle que :
  - Elle permet d'initialiser l'ensemble des attributs de l'objet sans utiliser de méthode ayant un effet de bord indésirable sur l'environnement de cet objet.
  - Autrement dit, soit l'ensemble des états de l'objets sont initiaux (atteignables par new), soit ils existent des méthodes publiques permettant d'atteindre cet état et celles-ci n'ont pas d'effet de bord.

Opération utilisable par LBP

- Expression restituant une valeur (Objet)
  - <object> … </object>
- Instruction appliquée à un objet (envoi de message)
  - <void> … </void>



LBP : syntaxe

– Composant du fichier XML
- <object {id="id"} class="c" {method="m"}> <args>*</object>
- <object idref="idref"/>
  - Si on omet le champ method, c'est équivalent à method="new" sinon c'est une méthode statique.
  - Il s'agit d'une expression qui restitue un objet, les types primitifs sont représentés par leurs wrappers.
  - Id permet de nommer l'objet créé.
- <void {id="id"} method ="m"> <args>*</void>
  - Apparaît nécessairement dans le contexte d'un objet (autre clause)
  - Si id est présent, il est associé à la valeur restituée par l'envoi de message

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1-beta" class="java.beans.XMLDecoder">
 <object class="Circuit">
    <void method="set">
       <object class="Generateur">
          <double>220.0</double>
       </object>
       …
```

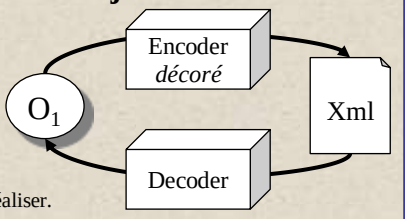

LBP : raccourci

<object class="java.lang.Integer"> <string>123</string> </object>	**123** **new Integer("123")**
<int>123</int>	
<object class="java.lang.Class" method="forName"> <string>java.awt.event.ActionListener</string> <object>	**ActionListener.class**
<class> java.awt.event.ActionListener</class>	
<void class="javax.swing.JTable" method="getField"> <string>AUTO_RESIZE_OFF</string> <void id="Id0" method"get"><null/></void> </void> <object idref="Id0"/>	**JTable.AUTO_RESIZE_OFF**
<object class="javax.swing.JTable" field="AUTO_RESIZE_OFF"/>	

LBP : delegate

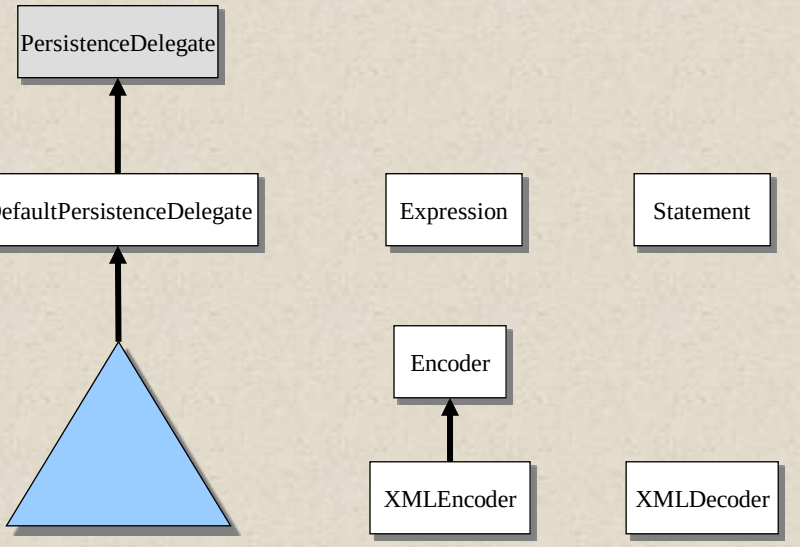
On décrit la séquence équivalente dans un objet d'annotation de la classe correspondante.

- Classe Delegate contenant 2 méthodes
  - instantiate (correspond à la balise object)
    - Qui décrit la fonction d'instanciation (new)
  - initialize (correspond à la balise void)
    - Qui décrit la séquence d'envoi de message à réaliser.




```
import java.beans.*;
public class InterrupteurPersistenceDelegate extends DefaultPersistenceDelegate  {
  protected void initialize(Class type, Object oldInstance, Object newInstance, Encoder out) {
    Interrupteur oI = (Interrupteur)oldInstance;
    if(oI.ouvert()) out.writeStatement(new Statement(oI,"ouvrir",new Object[] {}));
    else out.writeStatement(new Statement(oI,"fermer",new Object[] {}));
  }
  protected Expression instantiate(Object oldInstance,Encoder out)  {
    Interrupteur oI = (Interrupteur)oldInstance;
    return new Expression(oI,Interrupteur.class,"new",new Object[] {oI.label()});
  }}
```


LBP : Les classes de la librairie



LBP : PersitenteDelegate

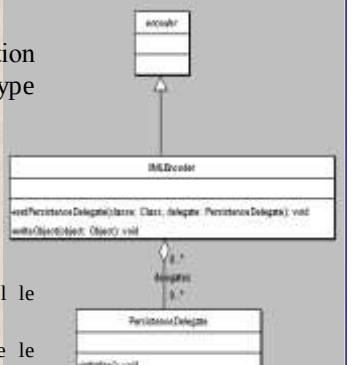
Method Summary

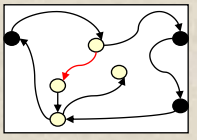
protected void	initialize(Class type, Object oldInstance, Object newInstance, Encoder out) Produce a series of statements with side effects on newInstance so that the new instance becomes *equivalent* to oldInstance.
protected abstract Expression	instantiate(Object oldInstance, Encoder out) Returns an expression whose value is oldInstance.
protected boolean	mutatesTo(Object oldInstance, Object newInstance) Returns true if an *equivalent* copy of oldInstance may be created by applying a series of statements to newInstance.
void	writeObject(Object oldInstance, Encoder out) The writeObject is a single entry point to the persistence and is used by a Encoder in the traditional mode of delegation.

LBP : Limitations

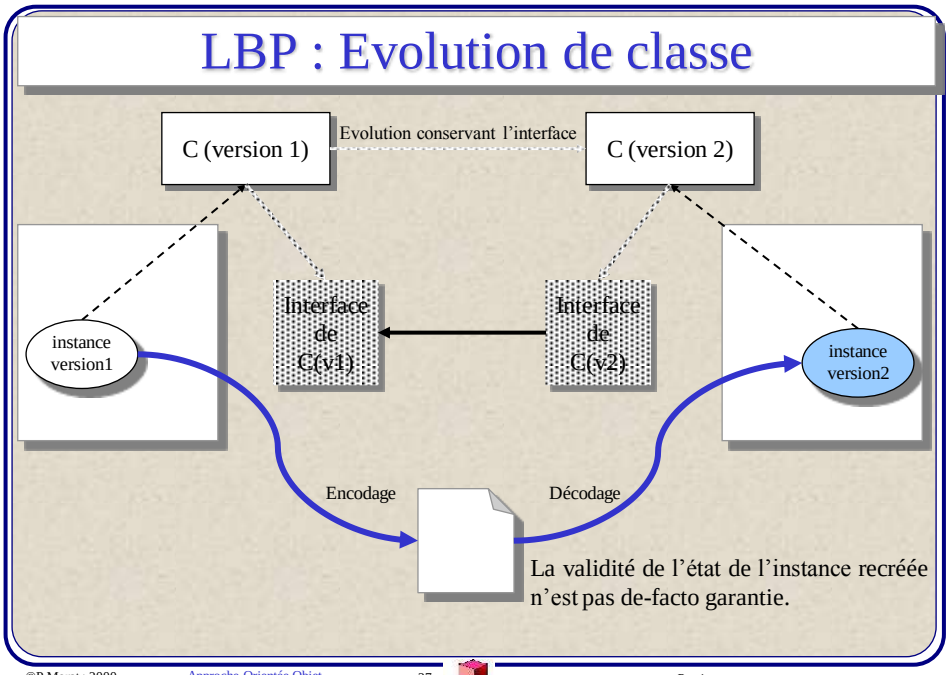
Principe opérationnel du codeur

- Le XMLEncoder possède une table d'association liant un type d'objet à une instance d'un sous-type de PersistenceDelegate.
  - Si l'association existe, on utilise le délégué
  - Sinon on applique une procédure par défaut
- Problèmes
  - Le renseignement de la table est fait par un tier
    - Rend l'ajout de nouveaux délégués difficile (viol le principe de open-close)
    - Il faut externaliser cette information pour rendre le modèle open-close.
      - » Utiliser un format XML !
  - La classe doit permettre d'atteindre tous les états
    - Toute classe n'est pas immédiatement LBPisable.





LBP : Evolution de classe



LBP : Exemple

```
public class Complexe implements Serializable {
   /** version 1 : utilisation du type double */
   private double reelle, imaginaire;
   public Complexe(double reelle, double imaginaire)  {
      this.reelle=reelle; this.imaginaire=imaginaire;
   }
   ...
}
```

```
public class Reel  implements Serializable {
   public Reel(double reel) { …}
   …
}
public class Complexe implements Serializable {
   /** version 2 : utilisation du type Reel */
   private Reel reelle, imaginaire;
   public Complexe(double reelle, double imaginaire)  {
      this.reelle=new Reel(reelle);
      this.imaginaire= new Reel( imaginaire);
   }
   ...
}
```