

Clonage

Conception Orient e Objet

Jean-Christophe Routier
Licence mention Informatique
Universit  Lille 1



Rappel : r f rence et affectation

```
public classe UneClasse {
    public int i;
}

UneClasse reference1 = new UneClasse();
reference1.i = 5;
UneClasse reference2 = reference1;
```

reference1 et reference2 d signent le **m me** objet

```
reference2.i = 25;
System.out.println(reference1.i); // -> 25
```

Comment dupliquer un objet ?

Cloner un objet

- int r t : avoir deux versions d'un m me objet susceptibles d' voluer diff remment

dans la classe Object :

- protected Object clone() throws CloneNotSupportedException
- Action de Object.clone() : r servation de l'espace m moire : copie bit   bit.
- Object.clone() est **protected**
- la valeur de retour est Object => downcast

```
Value v = new Value();
Value w = (Value) v.clone(); // illegal clone() non accessible
```

Permettre le clonage

- impl menter l'interface Cloneable

```
public interface Cloneable
```

Sert de d'indicateur. Int r t :

- "typer" les objets clonables (test par instanceof)
- permettre aux d veloppeurs d'avoir des classes d'objets pas clonable (Object.clone() v rifie si Cloneable)
- d clarer public la m thode clone()
- appeler (syst matiquement) super.clone();

Exemple

```
public class SomeValues implements Cloneable {
    String name;
    public SomeValues(String name) {
        this.name = name;
    }
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

G rer son propre clonage

Ajouter des "manipulations" dans clone().

```
public class SomeValues implements Cloneable {
    String name;
    public SomeValues(String name) {
        this.name = name;
    }
    public Object clone() throws CloneNotSupportedException {
        SomeValues monClone = (SomeValues) super.clone();
        monClone.name = "clone de "+this.name;
        return monClone;
    }
}
```

Attention : copie de r f rences

Le clonage par d faut est une copie bit   bit :

⇒ en cas de clonage, c'est la r f rence des attributs qui est copi e.

“shallow” copy

Extrait API doc pour ArrayList :

```
clone :
public Object clone()
Returns a shallow copy of this ArrayList instance.
(The elements themselves are not copied.)
Returns :
a clone of this ArrayList instance.
Overrides :
clone in class Object
```

```
public class SomeValues implements Cloneable {
protected List<Integer> al = new ArrayList<Integer>();
protected String name;
public SomeValues(String name) {
this.name = name;
al.add(Math.random()*100);
al.add(Math.random()*100);
}
public void dump() {
// affiche this.name et les  l ments de this.al
}
public Object clone() throws CloneNotSupportedException {
SomeValues sv = (SomeValues) super.clone();
sv.name = "clone de "+this.name;
return sv;
}
}
```

```
/** clonage "sans prise en compte" de la liste */
public class SomeValues1 extends SomeValues implements Cloneable {
public SomeValues1(String name) { super(name); }
public Object clone() throws CloneNotSupportedException {
return super.clone();
}
}

/** clonage par copie des valeurs de la List */
class SomeValues2 extends SomeValues implements Cloneable {
public SomeValues2(String name) { super(name); }
public Object clone() throws CloneNotSupportedException {
SomeValues2 sv = (SomeValues2) super.clone();
sv.al = new ArrayList<Integer>();
for(Integer i : this.al) { // objets dans al doivent  tre clonables
(sv.al.add((Integer) i.clone());
}
return sv;
}
}
```

```
/** mise en  vidence du probl me de la "shallow copy" dans le clonage */
public class TestClone {
public static void main(String[] args) {
SomeValues1 sv1 = new SomeValues1("sv1");
sv1.dump(); sv2.dump();
SomeValues1 cloneSv1 = null;
try {
cloneSv1 = (SomeValues1) sv1.clone();
cloneSv2 = (SomeValues2) sv2.clone();
catch(CloneNotSupportedException e) {}
cloneSv1.dump();
cloneSv2.dump();
sv1.al.set(0,0); //modification
sv2.al.set(0,0);
sv1.dump(); // affichage de la modification r alis e
sv2.dump();
cloneSv1.dump(); //!!! la liste du clone aussi a  t  modifi e!!!
cloneSv2.dump();
}
}
```



Conception Orient e Objet

Jean-Christophe Routier
Licence mention Informatique
Universit  Lille 1




Entr es-sorties : java.io.*

- bas es sur la notion de **stream**
= canal de communication entre un * crivain* et un *lecteur*.

Classes de streams de base :

- InputStream/OutputStream** classes abstraites d finissant les op rations de base pour la lecture/ criture de s quence d'octets
- Reader/Writer** classe abstraites d finissant les op rations de base pour la lecture/ criture de s quence de donn es de type caract re et supportant l'*Unicode*.

m thodes principales : read(...) et write(...), close()

java.io.IOException

Les principaux streams

- `InputStream[Reader|Writer]` classes pour transformer des caract res en octets et vice-versa.
 - `Data[Input|Output]Stream` streams sp cialis s qui ajoutent la possibilit  de lire/ crire des donn es de types de base.
 - `Object[Input|Output]Stream` streams sp cialis s dans la lecture/ criture d'objets Java s rialis s. Ici lecture implique reconstruction d'objet.
 - `Buffered[Input|Output]Stream/Buffered[Reader|Writer]` ajoutent un syst me de tampon (buffer) pour am liorer les performances.
 - `Piped[Input|Output]Stream/Piped[Reader|Writer]` Streams fonctionnant par paires, utilis s notamment pour la communication entre threads.
 - `File[Input|Output]Stream/File[Reader|Writer]` Streams permettant de lire/ crire dans un fichier.
- utiliser les variantes `Reader/Writer` pour des donn es caract res

"Combinaison" de streams

- Syst mes d'"enveloppes" successives, mises en cascade
 ⇔ design-patter `decorator`
- **bufferisation** envelopper le stream   "bufferiser" dans un `Buffered***Stream` :
`InputStream bufferedIn = new BufferedInputStream(unInputStream);`
- **streams de donn es** pour acc der facilement aux donn es de types primitifs :
`DataInputStream dis = new DataInputStream(System.in);`
`doudle d = dis.readDouble();`
- **transformations en cha ne de caract res** `PrintWriter` m thodes `print()`, `println()`

```
import java.io.*;
public class UtilInput {
    ...
    public static String readString() throws java.io.IOException {
        Reader isReader = new InputStreamReader(System.in);
        BufferedReader bReader = new BufferedReader(isReader);
        return bReader.readLine();
    }
}
OU
(new BufferedReader(new InputStreamReader(System.in))).readLine();
OU
new java.util.Scanner(System.in).nextLine();
```

Fichiers

la classe `File` : informations sur fichiers + manipulations

acc s s quentiel : `RandomAccessFile (seek())`

Streams de fichiers

```
FileInputStream in = new FileInputStream("/tmp/exemple");
DataInputStream dataIn = new DataInputStream(in);
BufferedInputStream bufDataIn = new BufferedInputStream(dataIn);

FileOutputStream out = new FileOutputStream(unObjetFile);
FileWriter fwOut = new FileWriter(out);
```

```
import java.io.*;
...
public void copieFichierTexte(String nomSource, String nomCible) {
    File source = new File(nomSource);
    File cible = new File(nomCible);
    try{
        BufferedReader in = new BufferedReader(new FileReader(source));
        PrintWriter out = new PrintWriter(new FileWriter(cible));
        String ligne;
        while ((ligne = in.readLine()) != null) {
            out.println(ligne);
        }
        out.close();
        in.close();
    }
    catch(IOException e) { e.printStackTrace(); }
}
```

S rialisation

Conception Orient e Objet

Jean-Christophe Routier
 Licence mention Informatique
 Universit  Lille 1



La Sériálisatió

Sériálisatió

La sérialisatió c'est transformer un objet (en mémoire) en une suite d'octets le représentant.

objet $\xrightarrow{\text{sériálisatió}}$ suite de bytes $\xrightarrow{\text{désériálisatió}}$ objet

objet o ds JVM₁ \rightarrow suite de bytes \rightarrow $\left(\begin{array}{c} \text{fichier} \\ \text{réseau} \end{array} \rightarrow \right)$ objet o ds JVM₂

- y compris entre JVM d'OS différents... (application : RMI)
- Permet la persistance entre sessions

Mise en œuvre, sérialisatió par défaut

La classe de l'objet doit implémenter `java.io.Serializable`

- **Sériálisatió** `ObjectOutputStream/ writeObject()`
la sérialisatió réalise la cloture transitive des dépendances sur l'objet sérialisé (graphe d'objets)
si o , à sérialiser, possède la référence sur des objets o_1, o_2 , ils seront eux aussi sérialisés (et donc y compris les objets qu'ils référencent : clôture transitive du graphe de dépendances des objets)
- **Désériálisatió** `ObjectInputStream/ readObject()`
(évidemment le ".class" doit être accessible à la JVM d'accueil)

writeObject()

`java.io.ObjectOutputStream`

`public final void writeObject(Object obj) throws IOException`

Write the specified object to the `ObjectOutputStream`. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are written. Default serialization for a class can be overridden using the `writeObject` and the `readObject` methods. Objects referenced by this object are written transitively so that a complete equivalent graph of objects can be reconstructed by an `ObjectInputStream`.(...)

Throws :

- `InvalidClassException` - Something is wrong with a class used by serialization.
- `NotSerializableException` - Some object to be serialized does not implement the `java.io.Serializable` interface.
- `IOException` - Any exception thrown by the underlying `OutputStream`.

readObject()

`java.io.ObjectInputStream`

`public final Object readObject()`

`throws OptionalDataException, ClassNotFoundException, IOException`

Read an object from the `ObjectInputStream`. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are read. Default deserializing for a class can be overridden using the `writeObject` and `readObject` methods. Objects referenced by this object are read transitively so that a complete equivalent graph of objects is reconstructed by `readObject`. The root object is completely restored when all of its fields and the objects it references are completely restored. (...)

Throws :

- `ClassNotFoundException` Class of a serialized object cannot be found.
- `InvalidClassException` Something is wrong with a class used by serialization.
- `StreamCorruptedException` Control information in the stream is inconsistent.
- `OptionalDataException` Primitive data was found in the stream instead of objects.
- `IOException` Any of the usual Input/Output related exceptions.

```
package essais.serialisable;
public class SerialisableData implements java.io.Serializable {
    private int i;
    private SerialisableData sd = null;
    public SerialisableData(int i, int j) {
        this(i);
        sd = new SerialisableData(j);
    }
    public SerialisableData(int i) {
        this.i = i;
    }
    public String toString() {
        return ""+i+ " et "+(sd==null? "" : sd.toString());
    }
} // SerialisableData
```

```
import java.io.*;
public class TestSerialisation {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        SerialisableData sd1 = new SerialisableData(1,2);
        SerialisableData sd2 = new SerialisableData(5,8);
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("sdata.dat"));
        out.writeObject(sd1); out.writeObject(sd2);
        out.close();
    } // TestSerialisation
    ===== DANS UN AUTRE main INDEPENDANT =====
    public class TestDeSerialisation {
        public static void main(String[] args)
            throws IOException, ClassNotFoundException {
            ObjectInputStream in = new ObjectInputStream(
                new FileInputStream("sdata.dat"));
            SerialisableData sd1 = (SerialisableData) in.readObject();
            SerialisableData sd2 = (SerialisableData) in.readObject();
            in.close();
            System.out.println("\n sd1 :"+sd1+"\n sd2 :"+sd2+"\n");
        }
    } // TestDeSerialisation
```

transient

- le modificateur d'attribut `transient` sp cifie que l'attribut ne doit pas  tre s rialis  (ni donc restaur )
- valeur `null`   la d s rialisation
- n cessit  de prise en charge par le programme de l'attribut d lais 

Personnalisation de la s rialisation

ajouter   une classe `Serializable` les m thodes :

- `private void writeObject(ObjectOutputStream out) throws IOException`
- `private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException`
- `private !`
- appel es automatiquement par `ObjectOutputStream.writeObject()` et `ObjectInputStream.readObject()`
- `ObjectOutputStream.defaultWriteObject()` et `ObjectInputStream.defaultReadObject()`
-   g re tous les attributs ni `static` ni `transient`

```
import java.io.*;
public class SerializableData2 implements java.io.Serializable {
    private int i;
    private transient String code;
    private SerializableData2 sd;
    public SerializableData2(int i, int j) {
        this(i); sd = new SerializableData2(j);
    }
    public SerializableData2(int i) { this.i = i; code = ""+i; }
    private void writeObject(ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        out.writeObject(encode(code));
    }
    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        code = decode((String)in.readObject());
    }
    private String encode(String code) { return code+"CRYPTE"; }
    private String decode(String code){
        return code.substring(0,code.length()-6);
    }
    public String toString() {
        return ""+i+" code :"+code+(sd==null? "" : " et "+sd);
    }
} // SerializableData2
```

Coherence de la s rialisation ?

Pb :

- Deux objets o_1 et o_2 partagent une r f rence sur l'objet o_3
- On s rialise o_1 et o_2 (qui chacun s rialise o_3) via le m me stream
- Qu'en est il lors de leur d s rialisation vis- -vis de o_3 ?
- Les d pendances d'objets sont conserv es !

```
public class SerializableData3 implements java.io.Serializable {
    private int i;
    public SerializableData3 sd = null;
    public SerializableData3(int i, SerializableData3 sd) {
        this(i); this.sd = sd;
    }
    public SerializableData3(int i) { this.i=i; }
    public boolean equals(SerializableData3 sd3) {
        return this.i == sd3.i &&
            ((this.sd == null && sd3.sd == null) || this.sd.equals(sd3.sd));
    }
    public boolean memoSd(SerializableData3 sd3) {
        return this.sd == sd3.sd;
    }
} // SerializableData3
...
SerializableData3 sd = new SerializableData3(12);
SerializableData3 sd1 = new SerializableData3(1, sd);
SerializableData3 sd2 = new SerializableData3(5, sd);
... S rialisation
... puis ouverture d'un stream de lecture "in"...
SerializableData3 sd1 = (SerializableData3) in.readObject();
SerializableData3 sd2 = (SerializableData3) in.readObject();
System.out.println("sd1 et sd2, memo Sd? "+ sd1.memoSd(sd2));
-----
| sd1 et sd2, memo Sd? true
-----
```