

Les I/O

contenu de la section

LES I/O 1

CONTENU DE LA SECTION..... 2

LES STREAMS : GÉNÉRALITÉS (1)..... 3

les streams..... 3

la hiérarchie des streams de byte..... 3

LES STREAMS : GÉNÉRALITÉS (2)..... 4

exemples..... 4

quelques API..... 4

LES STREAMS ET LES PERFORMANCES..... 5

les performances des différentes classes..... 5

LA SÉRIALISATION (1)..... 6

l'objectif et les principes..... 6

les classes `ObjectInputStream` et `ObjectOutputStream`..... 6

LA SÉRIALISATION (2)..... 7

l'attribut `transient` 7

LA SÉRIALISATION (3)..... 8

le `serialVersionUID`..... 8

LA SÉRIALISATION (4)..... 9

les références partagées..... 9

LA SÉRIALISATION (5)..... 10

les références partagées (suite)..... 10

LA SÉRIALISATION (6)..... 11

les champs `static`..... 11

LA SÉRIALISATION (7)..... 12

les champs `static` (suite)..... 12

COMPLÉMENTS SUR LA SÉRIALISATION..... 13

la compatibilité entre versions..... 13

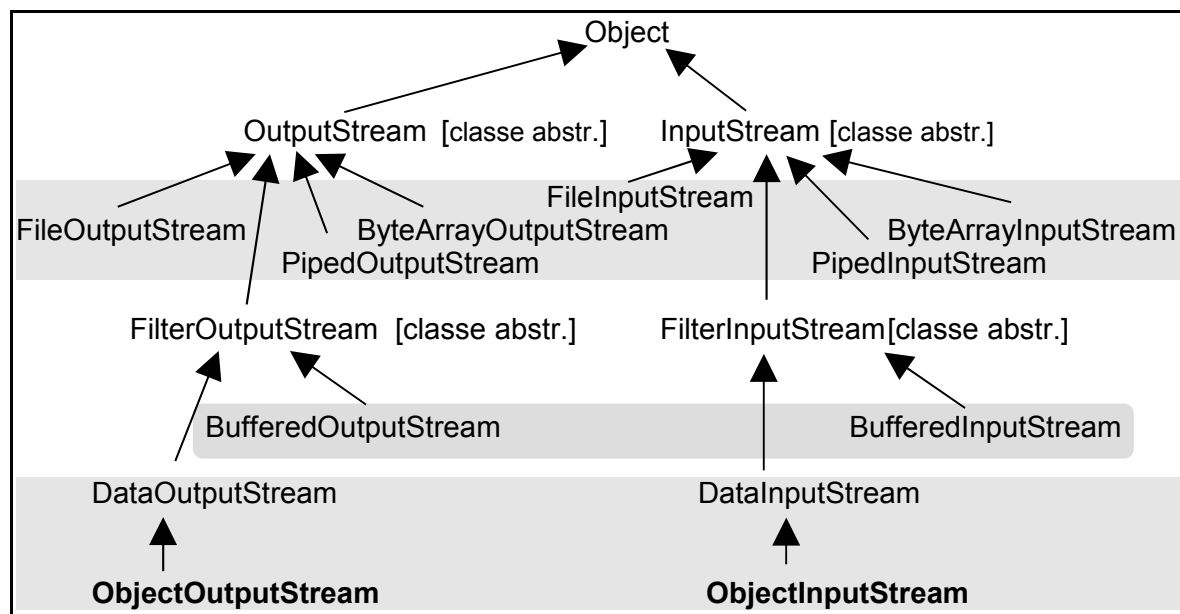
les streams : généralités (1)

les streams

- stream = canal de transfert entre la mémoire et un support d'information externe (fichier, périphérique)
- streams de byte vs streams de char

la hiérarchie des streams de byte

- la nature du support externe d'information (terminal, fichier, mémoire, tube)
- le mode de transfert (mode bufferisé, non bufferisé)
- les types directement manipulables



les streams : généralités (2)

exemples

- //flot d'I/O non bufferisé pour lire des flots d'octets dans un fichier
FileInputStream fln=new FileInputStream("fichier");
- //flot d'I/O bufferisé pour lire des flots d'octets dans un fichier
BufferedInputStream bln=new BufferedInputStream(new FileInputStream("fichier"));
- //flot d'I/O non bufferisé pour lire des flots de types de base dans un fichier
DataInputStream dln= new DataInputStream(new FileInputStream("fichier"));
- //flot d'I/O bufferisé pour lire des flots de types de base dans un fichier
DataInputStream dln=new DataInputStream(new BufferedInputStream(new FileInputStream("fichier")));

quelques API

- Les classes abstraites java.io.InputStream et java.io.OutputStream

abstract int read() throws IOException int read(byte[] buffer) throws IOException int read(byte[] buffer,int offset,int length) throws IOException long skip(long count) throws IOException int available() throws IOException void close() throws IOException	abstract int write(int octet) throws IOException void write(byte[] buffer) throws IOException void write(byte[] buffer,int offset,int length) throws IOException void flush() throws IOException void close() throws IOException
---	---

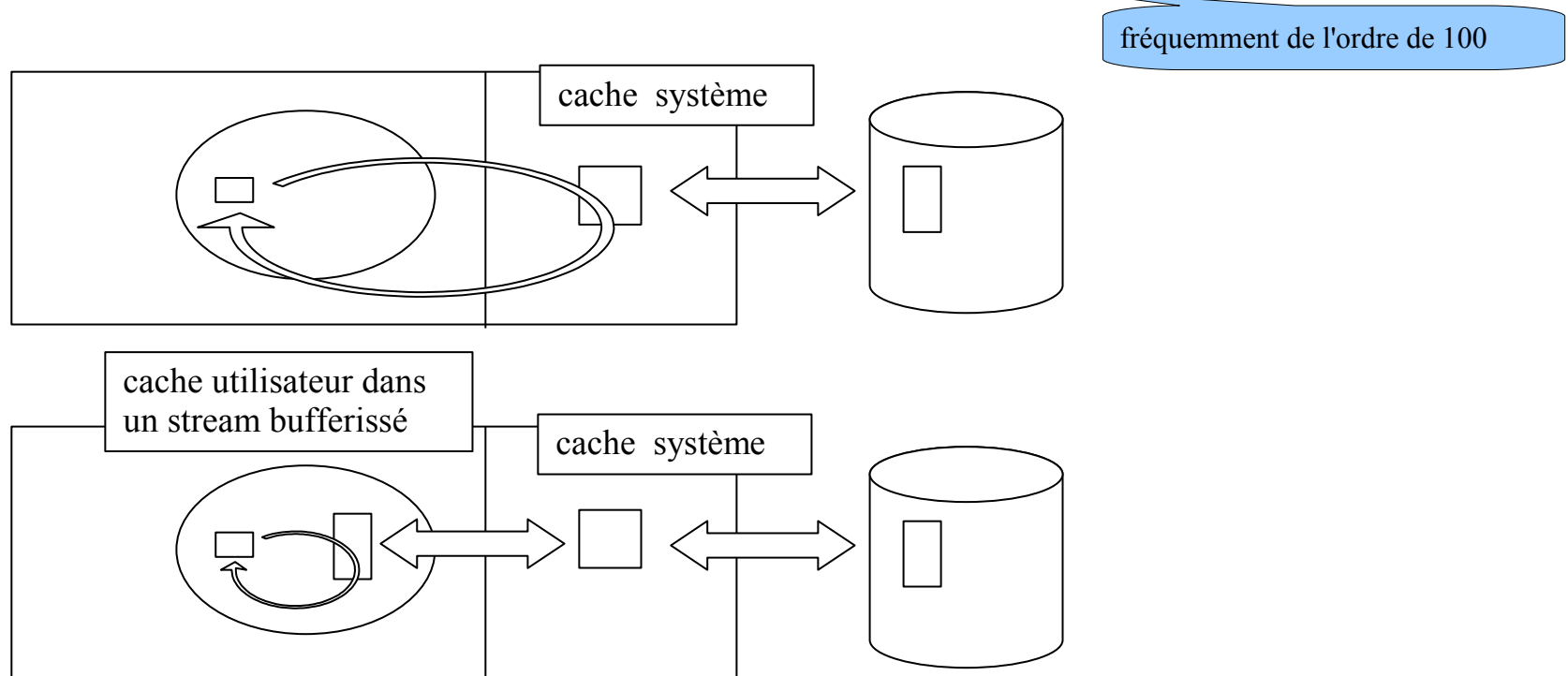
- les classes java.io.DataInputStream et java.io.DataOutputStream

boolean readBoolean() throws IOException; byte readByte() throws IOException; short readShort() throws IOException;	void writeBoolean(boolean) throws ... void writeByte(Byte) throws
---	---

Les streams et les performances

les performances des différentes classes

- streams bufferisées **beaucoup plus performants** que les streams non bufferisés (surtout `BufferedInputStream`)



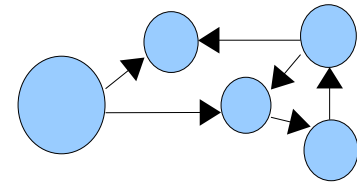
- effet du buffer fortement dépendant de la taille des données échangées
- bufferisation explicite via `read(byte[] buf)` et `write(byte[] buf)`
- streams de byte plus performants que les streams de char (conversion UTF)

fréquemment de l'ordre de 2

La sérialisation (1)

l'objectif et les principes

- sauvegarde de l'état d'objets (mécanisme de persistance)
- transmission des objets par valeur entre JVM
- transfert des objets **référéncés** → les objets référencés doivent être sérialisables
- format d'encodage des données
- service offert par les classes `ObjectInputStream` et `ObjectOutputStream`
les objets serialisables implémentent l'interface `java.io.Serializable` sinon génération de `NotSerializableException`



Attention : certaine classes usuelles ne sont pas sérialisables : IO classes, JDBC classes,

les classes `ObjectInputStream` et `ObjectOutputStream`

```
class ObjectInputStream extends DataInputStream {  
    Object readObject( ) throws IOException  
    .....  
}
```

conversion explicite de l'objet lu

```
.....  
FileOutputStream fs = new FileOutputStream("t.tmp");  
ObjectOutputStream ostream = new ObjectOutputStream(fs);  
ostream.writeInt(12345);  
ostream.writeObject("Today");  
ostream.writeObject(new Date());  
ostream.close();  
.....
```

```
class ObjectOutputStream extends DataOutputStream {  
    void writeObject(Object) throws IOException  
    .....  
}
```

```
.....  
FileInputStream fs = new FileInputStream("t.tmp");  
ObjectInputStream istream = new ObjectInputStream(fs);  
int i = istream.readInt();  
String today = (String) istream.readObject();  
Date date = (Date) istream.readObject();  
istream.close();  
.....
```

La sérialisation (2)

l'attribut transient

- la valeur exacte des champs déclarés « transient » non transférée ... (passage de la valeur par défaut)

```
import java.util.*;
import java.io.*;

class Logon implements Serializable {
    private Date date = new Date( );
    private String userName;
    private transient String passwd;

    Logon(String name,String pwd) {
        userName=name;
        passwd=pwd;
    }

    public void afficher( ) {
        System.out.println("logon info");
        System.out.println("nom : "+userName);
        System.out.println("date : "+date.toString());
        if (passwd==null) System.out.println("passwd : inconnu");
        else System.out.println("passwd : "+passwd);
    }
}
```

```
class TestLogon {
    public static void main(String[] args) {
        Logon a=new Logon("AAAA","aa");
        a.afficher( );
        try {
            FileOutputStream fO = new FileOutputStream("lg.out");
            ObjectOutputStream oS= new ObjectOutputStream(fO);
            oS.writeObject(a);
            oS.close( );
            FileInputStream fl = new FileInputStream("lg.out");
            ObjectInputStream iS=new ObjectInputStream(fl);
            Logon a1=(Logon) iS.readObject( );
            System.out.println("=====");
            a1.afficher( );
        } catch(Exception e) { .... }
    }
}
```

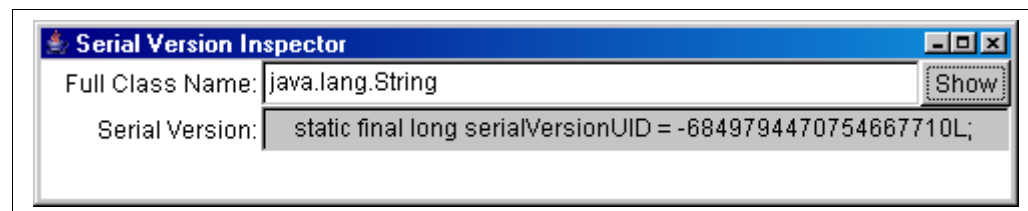
résultat de l'exécution

```
logon info
nom : AAAA
date : Sun Mar .....
password : aa
=====
logon info
nom : AAAA
date : Sun Mar .....
password : inconnu
```

La sérialisation (3)

le serialVersionUID

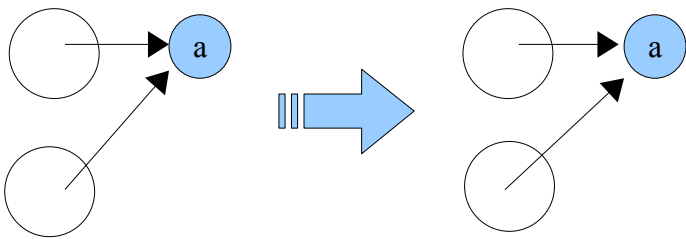
- garantir l'usage de classes compatibles à l'écriture et la lecture
- les 64 premiers bits du condensé SHA-1 calculé à partir :
 - le nom de la classe en UTF
 - les attributs de la classe (sous forme d'un int)
 - le nom de chaque interface triée (par le nom) en UTF
 - pour chaque champ trié par nom (sauf private static et private transient)
 - le nom du champ en UTF
 - les attributs du champ (sous forme d'un int)
 - le descripteur du champ en UTF
 - pour chaque méthode non private (y compris les constructeurs) triée par le nom et la signature
 - le nom de la méthode en UTF
 - les attributs de la méthode (sous forme d'un int)
 - le descripteur de la méthode en UTF
 - calcul du condensé par SHA-1
- serialVersionUID :
 - obtenu à partir du champs : `static final long serialVersionUID`
 - calculé par défaut par l'ObjectStream au moment de la sérialisation
- l'utilitaire : `serialver -show` (travaille à partir des `.class`)



La sérialisation (4)

les références partagées

- la sérialisation évite de dupliquer des objets référencés par plusieurs entités
 - l'**ObjectOutputStream** possède un cache des objets envoyés
 - pas de mise à jour des objets du cache !!!!



les tableaux sont des objets

```
.....
A refA1 = new A();
B refB1 = new B(refA1);
B refB2 = new B(refA1);

oS.writeObject(refB1);
refA1.setValue(10);
oS.writeObject(refB2);
refB1.setValue(1);
oS.writeObject(refB1);

affiche("B1", (B) iS.readObject());
affiche("B2", (B) iS.readObject());
affiche("B1", (B) iS.readObject());
.....

private void affiche(B iref) {
    System.out.println(nm + " "+ iref + " val de B " +
        iref.getValue() + " A1 " + iref.getA() + " val de A " +
        iref.getA().getValue());
}
.....
```

modification de A1

modification de B1

```
class A implements Serializable {
    private int value;

    public int getValue() { return value; }
    public void setValue(int v) { value = v; }
}

class B implements Serializable {
    private A a;
    private int value;

    B(A _a) { a = _a; }
    public A getA() { return a; }
    public void setA(A _a) { a = _a; }
    public int getValue() { return value; }
    public void setValue(int v) { value = v; }
}
```

```
B1 rmi.B@e7b241 val de B1 0 A1 rmi.A@167d940 val de A 0
B2 rmi.B@e83912 val de B2 0 A1 rmi.A@167d940 val de A 0
B1 rmi.B@e7b241 val de B1 0 A1 rmi.A@167d940 val de A 0
```

La sérialisation (5)

les références partagées (suite)

- l'invocation de reset() sur l'ObjectOutputStream remet à zéro le cache ...

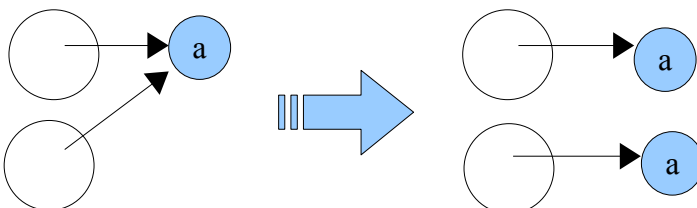
```
.....
A refA1 = new A();
B refB1 = new B(refA1);

bS = new ByteArrayOutputStream();
oS = new ObjectOutputStream(bS);

oS.writeObject(refB1);
oS.reset();
refA1.setValue(1);
refB1.setValue(1);
oS.writeObject(refB1);

affiche( "B1", (B) iS.readObject());
affiche( "B2", (B) iS.readObject());
.....

private void affiche(B iref) {
    System.out.println(nm + " " + iref + " val de B " +
        iref.getValue() + " A1 " + iref.getA() + " val de A " +
        iref.getA().getValue());
}
```



```
B1 rmi.B@e7b241 value de B 0 A1 rmi.A@167d940 value de A 0
B2 rmi.B@e83912 value de B 1 A1 rmi.A@1fae3c6 value de A 1
```

La s rialisation (6)

les champs static

- probl me pour equals() dans le cas des champs static
- utilisation de la m thode readResolve()

```
package syntax;

public class CouleurJ4 implements java.io.Serializable {
    private static final long serialVersionUID = -3396537866397086728L;
    public static final CouleurJ4 BLEU = new CouleurJ4("BLEU", 0);
    public static final CouleurJ4 BLANC = new CouleurJ4("BLANC", 1);
    .....

    private String name;
    private int ordinal;

    private CouleurJ4(String n, int o) { name = n; ordinal = o; }
    public String toString() { return name; }
    public int getOrdinal() { return ordinal; }

    /*public Object readResolve() {
        switch(getOrdinal()) {
            case 0 : return BLEU;
            case 1 : return BLANC;
            .....
            default : return null;
        }
    }*/
}
```

```
package syntax;
import java.io.Serializable;

public class PeintureJ4 implements Serializable {
    private static final long serialVersionUID = -6472424259061400096L;

    private CouleurJ4 couleur;

    public PeintureJ4(CouleurJ4 c) { couleur = c; }
    public CouleurJ4 getCouleur() { return couleur; }
}
```

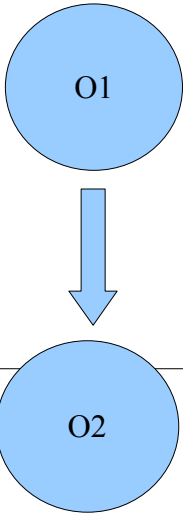
La s rialisation (7)

les champs static (suite)

•

```
PeintureJ4 p1 = new PeintureJ4(CouleurJ4.BLEU);
PeintureJ4 p2 = new PeintureJ4(CouleurJ4.BLEU);
.....
try {
    Socket socket = new Socket("localhost", 6668);
    ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
    out.writeObject(p1);
    out.writeObject(p2);
    .....
```

```
.....
PeintureJ4 p1 = (PeintureJ4) in.readObject();
PeintureJ4 p2 = (PeintureJ4) in.readObject();
if (p1.getCouleur() == CouleurJ4.BLEU)
    System.out.println("la couleur de p1 vaut Couleur.BLEU : " + p1.getCouleur());
else System.out.println("la couleur de p1 ne vaut pas Couleur.BLEU : " + p1.getCouleur());
if (p1.getCouleur() == p2.getCouleur())
    System.out.println("les couleurs de p1 et p2 sont identiques et valent : " + p1.getCouleur());
```



la couleur de p1 ne vaut pas Couleur.BLEU : BLEU
les couleurs de peint1 et peint2 sont identiques et valent : BLEU

r sultat de de l'ex cution de O2 sans d finition du readResolve dans CouleurJ4

la couleur de p1 vaut Couleur.BLEU : BLEU
les couleurs de peint1 et peint2 sont identiques et valent : BLEU

r sultat de de l'ex cution de O2 avec d finition du readResolve dans CouleurJ4

Compléments sur la sérialisation

la compatibilité entre versions

- Règles de base :
 - l'ORDRE de lecture (ou des sauts) doit être identique à celui des écritures
 - la sérialisation par défaut ne réalise aucune conversion de types
 - les informations superflues sont ignorées
 - les informations manquantes sont initialisées à la valeur par défaut
 - chaque classe référencée est identifiée par : son nom et ceux de sa super-classe, des types et des noms des champs non static et non transient.
 - L'écriture et la lecture des champs sont ordonnées : les types primitifs d'abord (triés par ordre alphabétique), les types références ensuite (triés par ordre alphabétique)
- modifications incompatibles :
 - champ non static non transient : suppression, changement du type, transformation en static ou transient
 - déplacement de la hiérarchie des classes
- modifications compatibles :
 - ajout d'un champ (non static, non transient, static ou transient)
 - transformation d'un champ static ou transient en non static, ou/et non transient
 - modification du droit d'accès d'un champ
 - ajout/suppression de classe et d'interfaces