

# Module Programmation Objet

## Chapitre 8 - Flots d'entrées/sorties

Pascal André, Gilles Ardourel

Université de Nantes

2010

Projet DVD Miage



## Généralités sur les entrées/sorties

### Entrée/sorties en Java

### Entrée/sorties en C++

### Entrée/sorties en Smalltalk

### Résumé

## Généralités sur les entrées/sorties

### Abstraction

### Flots d'entrée et de sortie

Entrée/sorties en Java

Entrée/sorties en C++

Entrée/sorties en Smalltalk

Résumé

# Entrées et sorties

## Abstraction

- ▶ L'entrée et la sortie de données sont correspondent aux échanges effectués avec les périphériques.
- ▶ Pour éviter la multiplication des protocoles, elles sont traitées de manière homogène en séparant
  1. la nature des informations à échanger (texte, binaire) associés
  2. le mode de transmission, les filtres, les codages...
  3. la nature du périphérique concerné (clavier, souris, écran, imprimante, fichier, port réseau...)
- ▶ Cette abstraction prend la forme de **flux** ou **flots** (**Stream**) auxquels on associe des périphériques.
- ▶ On reprend ainsi un des principes fondateurs d'Unix pour lequel un processus a des flots d'entrée et sortie (standards redirigeables).
- ▶ Concrètement l'opération est déléguée au système d'exploitation.

# Entrées et sorties

## Combinaison

- ▶ Le même flot peut être redirigé sur une source ou une cible différente.
- ▶ Le flot est orienté
  1. `InputStream`
  2. `OutputStream`
  3. plus rarement les deux en même temps `InputStream`  
(souvent hérité des deux)
- ▶ Les flots sont combinables pour établir des opérations
  1. filtrage
  2. encodage
  3. multiplex, ...

## Généralités sur les entrées/sorties

### Abstraction

### Flots d'entrée et de sortie

Entrée/sorties en Java

Entrée/sorties en C++

Entrée/sorties en Smalltalk

Résumé

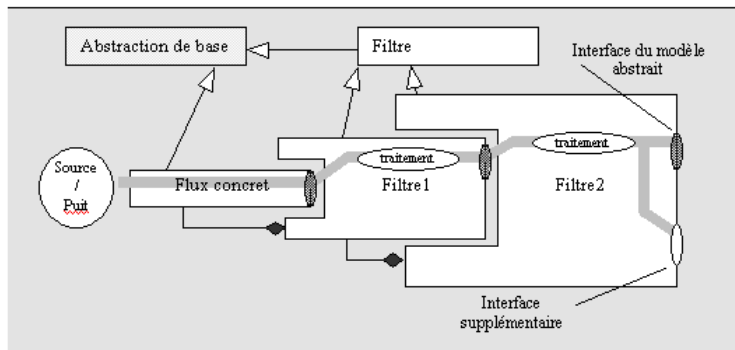
## Flots d'entrée et de sortie

- ▶ L'entrée et la sortie de données sont traitées de manière homogène.
- ▶ Les flots de entrée (clavier, fichier, etc.) et de sortie (écran, fichier, etc.) sont des objets disposant d'une interface compatible.
- ▶ Un programme étant un processus au sens Unix, on lui associe
  - `out` : la sortie standard (l'écran).
  - `in` : l'entrée standard (le clavier).
  - `err` : la sortie standard pour les erreurs (initialement, l'écran).

Le nommage varie d'un langage à l'autre.

# Flots d'entrée et de sortie

## Illustration



*Construction d'un flux d'entrée/sortie*



# Flots d'entrée et de sortie

## Suite et fin

- ▶ En programmation, les entrées sont analysables par les outils de compilation (on les emploie à petite échelle pour analyser les entrées de l'utilisateur).
- ▶ En programmation Web, les E/S sont spécialisées pour
  1. XML
  2. les objets transportables et les composants (ex: Beans)
- ▶ En développement d'application, la persistance se traite avec des entrées/sorties vers les bases de données (via SQL ou OQL par exemple).

⇒ Dans la suite, nous illustrons le propos sur Java  
... et de manière plus anecdotique sur C++ et Smalltalk

## Généralités sur les entrées/sorties

### Entrée/sorties en Java

Flux de données

Sérialisation

### Entrée/sorties en C++

### Entrée/sorties en Smalltalk

### Résumé

# Entrées et sorties en Java

## Organisation

Pour les entrées-sorties, on utilise

- des **flux de données** séquentiels pour les valeurs des types primitifs, souvent sous forme d'octets (*bytes*) ou de caractères;

	Input Stream	Output Stream
<b>Character</b>	<code>Reader</code>	<code>Writer</code>
<b>Byte</b>	<code>InputStream</code>	<code>OutputStream</code>

- la **sérialisation** pour les objets : c'est un rangement, en général binaire, efficace des objets.

Sources de cette partie : [Cla03, Ses05, Cha03, SM03]

<http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/IO/>

<http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap020.htm>

# Flux de données en Java

## 4 classes

	Input Stream	Output Stream
<b>Character</b>	<a href="#">Reader</a>	<a href="#">Writer</a>
	BufferedReader FilterReader InputStreamReader FileReader PipeReader  StringReader	BufferedWriter FilterWriter OutputStreamWriter FileWriter PipeWriter PrintWriter StringWriter
<b>Byte</b>	<a href="#">InputStream</a>	<a href="#">OutputStream</a>
	ByteArrayInputStream FileInputStream FilterInputStream InputStream  ObjectInputStream PipelnputStream	ByteArrayOutputStream FileOutputStream FilterOutputStream OutputStream PrintStream ObjectOutputStream PipeOutputStream
	RandomAccessFile	

source : [\[Ses05\]](#)

# Flux de données en Java

## PrintStream et InputStream

- ▶ En java, les flots de entrée sont habituellement des instances de la classe `InputStream` et les flots de sortie des instances de la classe `PrintStream`.
- ▶ Certains flots sont accessibles à partir des attributs de classe de la classe `System` :

`System.out` : la sortie standard (l'écran).

`System.in` : l'entrée standard (le clavier).

`System.err` : la sortie standard pour les erreurs  
(initialement, l'écran).

Noter que `System.in` est une instance de la classe `InputStream`, alors que les deux autres flux sont des instances de la classe `PrintStream`.

# Flux de données en Java

## PrintStream

- ▶ La classe `PrintStream` surcharge les méthodes `print()` et `println()` pour les types de base, `char[]`, `Object` (i.e. pour toute instance, indépendamment de sa classe) et `String`.

---

```
Person john = new Person();
```

```
System.out.print("Hello "); // print(String)  
System.out.println(john);   // print(Object)
```

---

# Les sorties en Java

## La méthode toString()

- ▶ La méthode `print(Object)` affiche, par défaut, le nom de la classe et l'adresse de l'instance passée en paramètre.
- ▶ Il est possible de modifier ce comportement en ajoutant une méthode appelée `toString()` à la classe que l'on souhaite afficher :

---

```
class Person {
    public String toString() {
        return name;
    }
}
// ...
Person sarah = new Person();
System.out.println(sarah); // affichage de l'attribut name
```

---

# Paquetage java.io

	Flux d'entrées	Flux de sorties
JDK 1.0 Flux d'octets (8 bits)	InputStream +- FileInputStream +- DataInputStream +- BufferedInputStream +- . . .	OutputStream +- FileOutputStream +- DataOutputStream +- BufferedOutputStream +- . . .
JDK 1.1 Flux de caractères (16 bits)	Reader +- FileReader +- StringReader +- . . .	Writer +- BufferedReader +- FileWriter +- BufferedWriter +- StringWriter +- . . .

Les classes d'E/S octets sont en partie dépréciée depuis la version originale du JDK pour cause de mauvaise portabilité.



# java.io

## abstract classes

`java.io.InputStream` **implements** `java.io.Closeable`

`java.io.Reader` **implements** `java.lang.Readable`, `java.io.Closeable`

`java.io.OutputStream` **implements** `java.io.Closeable`, `java.io.Flushable`

`java.io.Writer` **implements** `java.lang.Appendable`, `java.io.Closeable`,  
`java.io.Flushable`

InputStream	Reader	OutputStream	Writer
read	read	write	write
skip	skip		append
available	ready	flush	flush
close	close	close	close
reset	reset		
markSupported	markSupported		

Il existe des systèmes de conversions entre flots (cf exemple ci-après).

# Flux de données en Java

Exemple : in, out, err

Dans certains cas, capter les exceptions est nécessaire

---

```
import java.io.IOException;
```

```
public class IOTest {
    public static void main(String args[]) {
        try {
            int c, nbc = 0;
            System.out.print("rentrez une suite de caractères terminée par 'RC' ");
            while ((c = System.in.read()) != 10) {
                System.out.print(" / lu: " + c);
                nbc++;
            }
        } catch (IOException exc) {
            exc.printStackTrace(System.err);
        }
    }
}
```

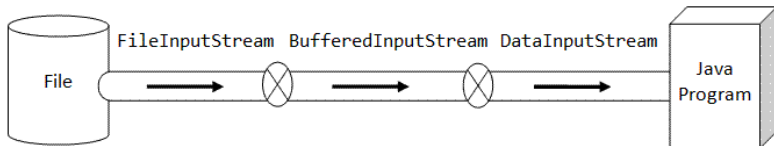
# Flux de données en Java

## Imbrication

L'exécution de l'exemple précédent montre que la lecture d'octet n'est guère adaptée aux chaînes de caractères, on utilisera la classe `BufferedReader` avec éventuellement une analyse lexicale.

rentrer une suite de caractères terminée par 'RC'123  
/ lu: 49 / lu: 50 / lu: 51 / lu: 132e exemple

Pour cela, on enveloppe les flux.



[http://www3.ntu.edu.sg/home/ehchua/programming/java/J5b\\_IO.html](http://www3.ntu.edu.sg/home/ehchua/programming/java/J5b_IO.html)

# Flux de données en Java

## Imbrication InputStream/BufferedReader

L'exemple suivant montre comment générer simplement un objet `BufferedReader` à partir de `System.in`.

```
Reader entree = new InputStreamReader(System.in);  
BufferedReader clavier = new BufferedReader(entree);
```

```
System.out.print("Entrez une ligne de texte : ");  
String ligne = clavier.readLine();  
System.out.println("Vous avez saisi : " + ligne);
```

```
Entrez une ligne de texte : abec hhh jcsq  
Vous avez saisi : abec hhh jcsq
```

# Flux de données en Java

## Imbrication (lecture)

---

```
File f = new File("fichier.mp3");  
FileInputStream fis = new FileInputStream(f);  
BufferedInputStream bis = new BufferedInputStream(fis);  
DataInputStream dis = new DataInputStream(bis);
```

```
int a = dis.readInt();  
short s = dis.readShort();  
boolean b = dis.readBoolean();
```

---

source <http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/IO/>

# Flux de données en Java

## Imbrication (écriture)

---

```
File f = new File("fichier.mp3");  
FileOutputStream fos = new FileOutputStream(f);  
BufferedOutputStream bos = new BufferedOutputStream(fos);  
DataOutputStream dos = new DataOutputStream(bos);
```

```
int a = 10; dos.writeInt(a);  
short s = 3; dos.writeShort(s);  
boolean b = true; dos.writeBoolean(b);
```

---

source <http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/IO/>

## Exemple 1/2

```
public static void main(String args []) {
    char carac;
    int nb1, nb2;
    String chaine, message;
```

```
    BufferedReader stdin = new BufferedReader(new InputStreamReader(
        System.in));    // déclarer le tampon de lecture
    try {
        System.out.print("Entrer le 1er nombre : ");
        chaine = stdin.readLine(); // lire les caractères dans une chaine
        nb1 = Integer.parseInt(chaine); // convertir la valeur lue en nombre
        System.out.print("Entrer le 2ème nombre : ");
        chaine = stdin.readLine(); // lire les caractères dans une chaine
        nb2 = Integer.parseInt(chaine); // convertir la valeur lue en nombre
        System.out.print("Entrer un caractère : ");
        chaine = stdin.readLine(); // lire le caractère dans une chaine
```

## Exemple 2/2

```

carac = chaine.charAt(0); // récupérer le caractère à la
// 1ère position de la chaine
System.out.print ("Entrer un message : ");
message = stdin.readLine ();
System.out.println ();
System.out.println ("le nombre lu est : " + nb1);
System.out.println ("le nombre lu est : " + nb2);
System.out.println ("le caractère lu est : " + carac);
System.out.println ("la chaine lue est : " + message);
} catch (IOException e) {
    System.err.println ("IOException thrown " + e.toString());
    // return false;
}
}

```

source <http://www.iro.umontreal.ca/~dift1870/A04/Pgms/Lecture.java>



# Flux de données en Java

## Exemple : copie de fichiers 1/2

```
> java Copy sourceFile.txt destFile.txt
```

---

```
import java.io.*;
public class Copy {
    public static void main (String [] argv) {
        // Test sur le nombre de paramètres passés
        if (argv.length != 2) {
            System.out.println ("Usage> java Copy sourceFile destinationFile");
            System.exit (0);
        }
        try { // Préparation du flux d'entrée
            File sourceFile = new File(argv [0]);
            FileInputStream fis = new FileInputStream(sourceFile);
            BufferedInputStream bis = new BufferedInputStream(fis);
            long l = sourceFile.length (); // Préparation du flux de sortie
```

# Flux de données en Java

Exemple : copie de fichiers 2/2

```

FileOutputStream fos = new FileOutputStream(argv[1]);
BufferedOutputStream bos = new BufferedOutputStream(fos);
// Copie des octets du flux d'entrée vers le flux de sortie
for(long i=0;i<l;i++) { bos.write(bis.read()); }
// Fermeture des flux de données
bos.flush(); bos.close(); bis.close();
} catch (Exception e) {
    System.err.println("File access error !");
    e.printStackTrace();
}
System.out.println("Copie terminée");
}
}

```

source <http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/IO/>

## Généralités sur les entrées/sorties

### Entrée/sorties en Java

Flux de données

Sérialisation

### Entrée/sorties en C++

### Entrée/sorties en Smalltalk

### Résumé

# Sérialisation

## Principes

- ▶ Entrées et sorties d'objets (binaires) = Solution simple de persistance
- ▶ L'interface `Serializable` ne définit aucune méthode mais permet par sous-typage de marquer une classe comme pouvant être sérialisée.

---

```
public final class String
extends Object
implements Serializable, Comparable, CharSequence
```

---

- ▶ `ObjectOutputStream` contient plusieurs méthodes de sérialisation des types primitifs : `writeInt`, `writeDouble`, `writeFloat` ...
- ▶ `ObjectInputStream` permet de lire des données de type primitifs : `readInt()`, `readDouble()`, `readFloat` ...
- ▶ Lors de la désérialisation, le constructeur de l'objet n'est jamais utilisé.

# Sérialisation

## Exceptions

La lecture ou l'écriture peut conduire à une erreur de flux sous-jacent ou d'objet sérialisable. Dans ces cas, Java lève une **exception**. Par exemple :

- ▶ `InvalidClassException` : fonctionnement incorrect d'une classe sérialisée.
- ▶ `NotSerializableException` : l'objet n'implante pas l'interface adéquate.
- ▶ `IOException` : erreur du flux sous-jacent.
- ▶ `OptionalDataException` : une valeur primitive est rencontrée au lieu d'un objet.
- ▶ `StreamCorruptedException` : les informations de contrôle du flux sont incohérentes.
- ▶ `ClassNotFoundException` : la classe d'un objet sérialisé n'est pas trouvée.

# Sérialisation

## Exemple 1/6

Exemple de sérialisation de voitures [SM03].

```
import java.io. Serializable ;
```

```
public class Moteur implements Serializable {
```

```
    String valeur ;
```

```
    Moteur (String s) {
```

```
        valeur = s;
```

```
    }
```

```
    String getValeur() {
```

```
        return valeur ;
```

```
    }
```

```
}
```

# Sérialisation

## Exemple 2/6

---

```
import java.io . Serializable ;

public class Carrosserie implements Serializable {
    String valeur ;
    Carrosserie (String s) {
        valeur = s;
    }
    String getValeur() {
        return valeur ;
    }
}
```

---

Un moteur et une carrosserie sont sérialisables (pas de méthodes spécifiques, car `String` est sérialisable). Une voiture est simplement définie comme un "sérialisable", sachant que ses variables d'instance le sont.

# Sérialisation

## Exemple 3/6

```
import java.io. Serializable ;

public class Voiture implements
    Serializable {
    Moteur moteur;
    Carrosserie carrosserie ;
    transient int essence;

    Voiture (String m, String c) {
        moteur = new Moteur(m);
        carrosserie = new Carrosserie(c)
    }

    String getMoteur() {
        return moteur.getValeur();
    }
    String getCarrosserie () {
        return carrosserie .getValeur ();
    }
    void setCarburant(int e) {
        essence += e;
    }
    int getCarburant() {
        return essence; }
}
```

Le modificateur de visibilité **transient** signifie que cet attribut doit être ignoré dans la sérialisation, il est "temporaire".



## Exemple 4/6

La sérialisation (du même paquetage) s'écrit ainsi :

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class Serialisation {
    public static void main (String [] args) throws IOException {
        Voiture voiture = new Voiture("V6", "Cabriolet");
        voiture.setCarburant(50);
        FileOutputStream f = new FileOutputStream("garage");
        ObjectOutputStream o = new ObjectOutputStream(f);
        o.writeObject( voiture );
        o.close ();
    }
}
```

Le fichier 'garage' est peu lisible et non imprimable.

## Exemple 5/6

La désérialisation (du même paquetage) s'écrit ainsi :

---

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class Deserialisation {
    public static void main (String [] args)
        throws IOException,
        ClassNotFoundException {
        FileInputStream f = new FileInputStream("garage");
        ObjectInputStream o = new ObjectInputStream(f);

        Voiture voiture = (Voiture)o.readObject();
        o.close ();
```

## Exemple 6/6

```
System.out.println ("Carrosserie : " + voiture.getCarrosserie());  
System.out.println ("Moteur : " + voiture.getMoteur());  
System.out.println ("Carburant : " + voiture.getCarburant());  
}  
}
```

Ce code produit le résultat suivant :

```
Carrosserie : Cabriolet  
Moteur : V6  
Carburant : 0
```

Un autre exemple sur les personnes est donné dans

<http://perso.wanadoo.fr/jm.doudoux/java/tutorial/chap020.htm>

# Exemple

## Exceptions

Si la classe a changé entre le moment où une instance a été sérialisée et le moment où l'instance est désérialisée, une exception est levée. Par exemple si on modifie et recompile la classe XX, on obtient :

```
java.io.InvalidClassException: XX; Local class not compatible: stream class
```

Une exception de type `StreamCorruptedException` peut être levée si le fichier a été corrompu par exemple en le modifiant avec un éditeur.

```
java.io.StreamCorruptedException: InputStream does not contain a serialized obj
```

Une exception de type `ClassNotFoundException` peut être levée si l'objet est transtypé vers une classe qui n'existe plus ou pas au moment de l'exécution.

```
java.lang.ClassNotFoundException: XX  
at java.io.ObjectInputStream.inputObject(ObjectInputStream.java:981)
```

## Exemple amusant 1/3

Voici un exemple amusant de sauvegarde de fenêtres Swing dû à

<http://www.infini-fr.com/Sciences/Informatique/Langages/Imperatifs/Java/IO/serialisation.html>

```
package serialisation ;

import java.io.*;
import java.awt.*;
import javax.swing.*;

public class Serialisation {
    private final static Reader reader = new InputStreamReader(System.in);
    private final static BufferedReader keyboard = new BufferedReader(reader);

    // Permet de créer une fenêtre et de la sérialiser dans un fichier.
    public void saveWindow() throws IOException {
        JFrame window = new JFrame("Ma fenêtre");
```

## Exemple 2/4

```

JPanel pane = (JPanel)window.getContentPane();
pane.add(new JLabel("Barre de status"), BorderLayout.SOUTH);
pane.add(new JTree(), BorderLayout.WEST);
JTextArea textArea = new JTextArea("Ceci est le contenu !!!");
textArea.setBackground(Color.GRAY);
pane.add(textArea, BorderLayout.CENTER);
JPanel toolbar = new JPanel(new FlowLayout());
toolbar.add(new JButton("Open"));
toolbar.add(new JButton("Save"));
toolbar.add(new JButton("Cut"));
toolbar.add(new JButton("Copy"));
toolbar.add(new JButton("Paste"));
pane.add(toolbar, BorderLayout.NORTH);
window.setSize(400,300);

```

## Exemple 3/4

```

    FileOutputStream fos = new FileOutputStream("window.ser");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(window);
    oos.flush ();
    oos.close ();
}

// Permet de reconstruire la fenêtre à partir des données du fichier.
public void loadWindow() throws Exception {
    FileInputStream fis = new FileInputStream("window.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    JFrame window = (JFrame)ois.readObject();
    ois.close ();
    window.setVisible (true);
}

```

## Exemple 4/4

```
// Permet de saisir différentes commandes. Testez plusieurs load
// consécutifs : plusieurs fenêtres doivent apparaître
public static void main(String[] args) throws Exception {
    Serialisation object = new Serialisation ();
    while(true) {
        System.out.print("Saisir le mode d'exécution
                           (load ou save) : ");
        String mode = keyboard.readLine();
        if (mode.equalsIgnoreCase("exit")) break;
        if (mode.equalsIgnoreCase("save")) object.saveWindow();
        if (mode.equalsIgnoreCase("load")) object.loadWindow();
    }
    System.exit(0);
}
```



# Entrées et sorties en Java

## Persistence

- ▶ La sérialisation est un mécanisme de base qui doit être adapté pour la gestion de versions, la sécurité (cryptage), les propriétés de classes, les objets non sérialisables (le mot-clé `transient` pas toujours indiqué).
- ▶ Il est possible de personnaliser la sérialisation d'un objet. Dans ce cas, la classe doit implémenter l'interface `Externalizable` qui hérite de l'interface `Serializable`. Cette interface définit deux méthodes : `readExternal()` et `writeExternal()`.
- ▶ La persistance se gère aussi à plus grande échelle avec les bases de données et l'interface JDBC (*Java DataBase Connectivity*) ou pour le Web par JDO, DAO ou Hibernate, ....

## Généralités sur les entrées/sorties

### Entrée/sorties en Java

### Entrée/sorties en C++

Entrée/sorties standards en C++

Entrée/sorties fichiers en C++

### Entrée/sorties en Smalltalk

### Résumé

# Entrées et sorties en C++

- ▶ En C++, les flots d'entrée et de sortie sont accessibles à partir de quelques flots prédéfinis :
  - `cout` : la sortie standard.
  - `cin` : l'entrée standard.
  - `cerr`, `clog` : les sorties d'erreurs (direct, via tampon).
- ▶ Ces flots sont fournis par la bibliothèque *I/O Stream*.
- ▶ Les opérateurs `<<` et `>>` sont utilisés pour écrire et lire dans un flot, respectivement.

# Les sorties en C++

---

```
#include <iostream.h>
main() {
    int i = 42;
    Person sarah;

    cout << "La valeur de i: " << i << "\n";
    cout << "La personne est: " << sarah << endl;
}
```

---

# Les sorties en C++

## L'opérateur <<

- ▶ Tout comme en Java, il est possible de modifier la façon dont un objet est affiché, grâce à la surcharge de l'opérateur <<.
- ▶ Cependant, cela ne peut pas se faire à l'intérieur d'une classe.

---

```
ostream& operator<<(ostream& out, const Person& p){  
    out << p.getName();  
    return out;  
}  
// ...  
Person sarah;  
cout << sarah;
```

---

# Les sorties en C++

## L'opérateur <<

- ▶ Comme l'opérateur est défini à l'extérieur de la classe, il ne peut pas accéder aux propriétés privées de celle-ci (encapsulation oblige).
- ▶ Cette règle peut être contournée si la classe accorde explicitement à une fonction le droit d'accéder à ses propriétés privées.
- ▶ Les fonction qui ont ce privilège, sont appelées les fonctions *amies*.

# Les fonctions amies

C++

- Pour qu'une fonction devienne l'amie d'une classe, elle doit être explicitement citée dans la description de la classe.

---

```
class Person {  
    friend ostream& operator<<(ostream& out, const Person& p);  
}
```

```
ostream& operator<<(ostream& out, const Person& p){  
    out << p.name; // accès à un attribut privé  
    return out;  
}
```

---

## Généralités sur les entrées/sorties

### Entrée/sorties en Java

### Entrée/sorties en C++

#### Entrée/sorties standards en C++

#### Entrée/sorties fichiers en C++

### Entrée/sorties en Smalltalk

### Résumé



# Entrées et sorties en C++

## Fichiers

- ▶ En C++, les flots d'entrée et de sortie sur fichiers s'utilisent de manière similaire.
  - `ofstream` : écriture seule.
  - `ifstream` : lecture seule.
  - `fstream` : lecture/écriture.
- ▶ Ces flots sont fournis par la bibliothèque `<fstream>`.

# Entrées et sorties en C++

## Fichiers

- Il faut ouvrir et fermer explicitement les fichiers par les méthodes `open (filename, mode)` et `close`. Le mode est une combinaison de

<code>ios::in</code>	lecture.
<code>ios::out</code>	écriture.
<code>ios::binary</code>	mode binaire.
<code>ios::ate</code>	se positionne à la fin.
<code>ios::app</code>	mode "append".
<code>ios::trunc</code>	si le contenu existe, il est substitué.

```
ofstream fic ;
//déclaration
fic.open("example.bin", ios::out | ios::app | ios::binary );
//ouverture
```

# Entrées et sorties en C++

## Fichiers

- On peut simultanément déclarer et ouvrir le flot:

```
ofstream fic ("example.bin", ios::out | ios::app | ios::binary);
```

La fonction renvoie 0 si tout c'est bien passé, un entier non nul si une condition d'erreur existe.

---

```
if (! fichier_sortie ) {
    cerr<< "erreur d'ouverture de fic" ;
    exit(-1);
}
```

---

- `endl` est le caractère de de saut de ligne (équivalent à `'\n'`)

# Entrées et sorties en C++

## Fichiers

- Les opérateurs << et >> sont aussi utilisables et ce de préférence aux fonctions

<pre> write(const char* str,int length) put(char) read(const char* str,int length) get(char) getline peek() seek() </pre>	<p>écriture. écriture d'un caractère. lecture d'une chaîne. lecture d'un caractère. lecture d'une ligne. retourne le caractère suivant. accès direct.</p>
---	---

`getline (char * buf,int limit ,char delim='\\n')`

# Entrées et sorties en C++

## Exemple de fichier texte

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
    ofstream myfile ("example.txt");
    if (myfile.is_open())
    {
        myfile << "This is a line.\n";
        myfile << "This is another line.\n";
        myfile.close();
    }
    else cout << "Unable to open file";
    return 0;
}
```

source <http://www.cplusplus.com/doc/tutorial/files/>

# Entrées et sorties en C++

## Méthodes de tests

### ► Valeurs de tests des primitives d'accès

<code>eof()</code>	fin de fichier.
<code>bad()</code>	erreur de flot.
<code>fail()</code>	échec futur.
<code>good()</code>	succès futur.
<code>clear()</code>	réinitialisation.

### Exemple de tests

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```

# Entrées et sorties en C++

## Exemple de tests

...

```
int main () {  
    string line ;  
    ifstream myfile ("example.txt");  
    if ( myfile.is_open() )  
    { while ( ! myfile.eof() ) {  
        getline ( myfile , line );  
        cout << line << endl; }  
        myfile.close ();  
    } else cout << "Unable to open file";  
    return 0;  
}
```

source <http://www.cplusplus.com/doc/tutorial/files/>

# Entrées et sorties en C++

## Exemple binaire

```
ifstream :: pos_type size ;
char * memblock;
int main () {
    ifstream file ("example.bin", ios::in | ios :: binary | ios :: ate);
    if ( file .is_open())
    { size = file . tellg ();
      memblock = new char [size];
      file .seekg (0, ios :: beg);
      file .read (memblock, size); file .close ();
      cout << "the complete file content is in memory";
      delete [] memblock;
    } else cout << "Unable to open file";
    return 0;
}
```

source <http://www.cplusplus.com/doc/tutorial/files/>



# Sérialisation

## Principes

- ▶ La sérialisation est possible par des bibliothèques qui étendent STL.
  - ▶ STL Serialization Library (STL-SL)
  - ▶ s11n.net
  - ▶ POST++
  - ▶ STLplus
  - ▶ Boost C++
  - ▶ ...
- ▶ Les principes sont similaires à ceux de Java.
- ▶ Nous ne donnons pas plus de détails.

## Généralités sur les entrées/sorties

### Entrée/sorties en Java

### Entrée/sorties en C++

### Entrée/sorties en Smalltalk

Flots de données

Fichiers et stockage binaire

### Résumé

# Entrées et sorties en Smalltalk

## Flux de données

La gestion des entrées/sorties se fait via

- ▶ les flux : hiérarchie de **Stream**
- ▶ le modèle MVC
  - ▶ hiérarchie de **View** : affichage, y compris la console
  - ▶ hiérarchie de **Controller** : actions clavier/souris - menus
  - ▶ hiérarchie de **Model** : contenu à afficher
- ▶ les protocoles standard d'affichage **printing** et d'entrée/sortie texte **fileIn/Out** hérités de la classe **Object**.

Nous illustrons le propos avec des exemples déjà abordés dans les chapitres 1 et 2, et d'autres repris du net.

# Entrées et sorties en Smalltalk

## Entrées et sorties standard

- ▶ Il n'y a pas d'entrée/sortie standard a priori : cela dépend toujours du contexte d'exécution (le contrôleur et la fenêtre active) e.g. l'évaluation `printIt` d'une expression affiche dans la fenêtre de l'expression.
- ▶ On peut néanmoins indiquer le flot d'entrée ou de sortie. e.g. la console `Transcript` est une sous-vue de la fenêtre principale.

---

`afficheSomme`

*"Affiche les relevés mensuels"*

```
| somme |
somme ←self exploiterRelevés .
(1 to: 12) do: [:i | Transcript cr;
  show: 'Somme du mois de ', i printString;
  tab; show: (somme at: i) printString ]
```

# Entrées et sorties en Smalltalk

## Quelques méthodes d'affichage

Méthode de <code>Stream</code>	Description
<code>show:</code>	affiche une chaîne de caractères.
<code>cr</code>	passse à la ligne.
<code>tab</code>	place une tabulation.
<code>space</code>	place une tabulation.
Méthode de <code>Object</code>	Description
<code>printString</code>	construit une flot sur une chaîne de caractères.
<code>printOn: aStream</code>	affiche l'objet dans un flot.
<code>storeOn: aStream</code>	stocke l'objet sous forme d'une chaîne de caractères.

Revenons sur les flots...

# Flots en Smalltalk

## Principales classes de flots

```

Object ()
Stream ()
  ExternalDatabaseAnswerStream ('session' 'nextRow')
  PeekableStream ()
    EncodedStream ('binary' 'stream' 'encoder' 'policy'
                  'lineEndConvention' 'lineEndCharacter' 'pos')
  PositionableStream ('collection' 'position' 'readLimit'
                    'writeLimit' 'policy')
  ExternalStream ()
  InternalStream ()
  ReadStream ()
  WriteStream ()
    ReadWriteStream ()
      ByteCodeReadWriteStream ('noPeekPosition')
      TextStream ('lengths' 'emphases' 'currentEmphasis'
                'runStartPosition')
  Random ('seed' 'increment' 'modulus' 'fmodulus' 'multiplier')

```

# Flots en Smalltalk

## Principales classes de flots externes

```

ExternalStream ()
  BufferedExternalStream ('lineEndCharacter' 'binary'
    lineEndConvention' 'bufferType' 'ioBuffer' 'ioConnection')
  ExternalReadStream ()
    CodeReaderStream ('swap' 'isBigEndianPlatform' 'scratchBuffer')
    ExternalReadAppendStream ('writeStream')
    ExternalReadWriteStream ()
  ExternalWriteStream ()
    CodeWriterStream ('scratchBuffer')

```

### Méthodes

```

('accessing' #contents #flush #next #next: #next:put: #nextPut: #nextPutAll: ..
('testing' #atEnd #isReadable #isWritable #needsFileLineEndConversion)
('enumerating' #do:)
('character writing' #cr #crtab #crtab: #emphasis #emphasis: #space #tab #tab:)
('status' #close)
('fileOut' #nextChunkPut: #timeStamp)
('printing' #print: #store:)

```

# Entrées et sorties en Smalltalk

## Quelques méthodes de la classe Stream

Méthode	Description
<code>next</code>	lit le prochain élément.
<code>nextPut:</code>	place un objet dans le flot.
<code>nextPutAll:</code>	place une collection d'objets dans le flot.
<code>contents</code>	récupère le contenu.
<code>flush</code>	vide le tampon sur le périphérique (si tampon).
<code>close</code>	ferme le flot.



# Entrées et sorties en Smalltalk

## Exemple

---

```
"write a simple file "  
file ← 'myTestFile.txt' asFilename.  
stream ← file writeStream.  
stream nextPutAll: 'Line 1'.  
stream nextPut: Character cr.  
stream nextPutAll: 'Line 2'.  
stream nextPut: Character cr.  
stream nextPutAll: 'Line 3'.  
stream nextPut: Character cr.  
stream close.
```

---

source <http://www.cincomsmalltalk.com/files/jarober/simple-file-io.ws>

## Généralités sur les entrées/sorties

### Entrée/sorties en Java

### Entrée/sorties en C++

### Entrée/sorties en Smalltalk

#### Flots de données

#### Fichiers et stockage binaire

### Résumé

# Entrées et sorties en Smalltalk

## Fichiers

- ▶ Les entrées/sorties fichiers se font sur des flots externes `ExternalStream`
- ▶ Il "suffit" de connecter le périphérique au flot...
  - ▶ en associant le flot à un nom de fichier `nomFic : Filename`
  - ▶ qu'on aura créé à partir d'un chemin exprimé par une chaîne `String` `asFilename` et une connexion `FileConnection`
  - ▶ On remplit le flot par `nextPutAll`.
  - ▶ On ferme le flot par `close`.

---

```
| nomFic flot |
nomFic ← 'ficTexte.txt' asFilename.
(nomFic exists)
  ifTrue: [Dialog warn: 'le fichier ', nomFic asString, ' existe déjà'.]
  ifFalse: [flot ← nomFic writeStream.
            flot nextPutAll: 'bonjour ma chaîne'. flot close.].
Transcript show: nomFic fileSize printString .
```

# Entrées et sorties en Smalltalk

## Quelques méthodes sur fichiers

Méthode	Description
<code>canBeWritten</code>	accessible en écriture.
<code>isDirectory</code>	répertoire.
<code>isWritable</code>	test d'écriture .
<code>isReadable</code>	test de lecture.
<code>isRelative</code>	le nom est relatif.
<code>isAbsolute</code>	le nom est absolu.

## Opérations sur fichiers

Méthode	Description
<code>copyTo:</code>	copie.
<code>delete</code>	suppression.
<code>makeDirectory</code>	créer répertoire.
<code>makeWritable</code>	accès écriture.
<code>fileSize</code>	taille.
<code>definitelyExists</code>	test d'existence.

# Entrées et sorties en Smalltalk

## Exemple de lecture fichier

---

```
"read the entire file"
```

```
file ← 'myTestFile.txt' asFilename.  
contents ← file contentsOfEntireFile .  
↑contents.
```

```
"read line by line"
```

```
file ← 'myTestFile.txt' asFilename.  
stream ← file readStream.  
lines ← OrderedCollection new.  
[stream atEnd] whileFalse: [| line |  
    line ← stream upTo: Character cr. lines add: line ].  
stream close .  
↑lines .
```

---

source <http://www.cincomsmalltalk.com/files/jarober/simple-file-io.ws>

# Entrées et sorties en Smalltalk

## Exemple de test fichier

---

```
" finding the file "
directory ← '.' asFilename.
files ← directory filesMatching: 'myTest*'.
↑ files .
```

---

source <http://www.cincomsmalltalk.com/files/jarober/simple-file-io.ws>

## 2e exemple [Hun98]

---

```
load: filename
|newFile newLineChar stream readingBlock item student |
newFile ←filename value asFilename.
```

# Entrées et sorties en Smalltalk

## 2e exemple

```

newFile exists
  ifTrue :
    [newLineChar ←Character cr.
     stream ←newFile readStream.
     contents ←OrderedCollection new: 250.
     readingBlock ←
      [[stream atEnd]
       whileFalse :
         [item ←stream upTo: newLineChar.
          contents add: item .]].
     Cursor read showWhile:
      [readingBlock valueNowOrOnUnwindDo:
       [stream close ]]].
     ifFalse : [Dialog warn: 'File does not exist!!']

```

# Entrées et sorties en Smalltalk

## 2e exemple (suite)

- ▶ La lecture se fait ici ligne à ligne par l'envoi du message `upTo:` avec `newLineChar` en paramètre.
- ▶ Si le fichier existe, un bloc est créé qui contient le code qui va lire les données dans le fichier. Ce bloc est affecté à une variable `readingBlock`. Il est évalué en lui envoyant le `valueNowOrOnUnwindDo:` message. Le paramètre donné, qui est aussi un bloc, sera exécuté en cas d'erreur. Dans ce cas, le flux et le fichier sont fermés.
- ▶ La boucle `showWhile:` spécifie ici le curseur à afficher durant l'opération. Dans ce cas, c'est le curseur de lecture qui s'affiche pendant la lecture du fichier.



# BOSS

BOSS est l'acronyme de *Binary Object Storage System*.

- ▶ Il est une des méthode de stockage d'objets dans des flots binaires, y compris pour les transmissions en réseau.
- ▶ Noter qu'avec VisualWorks, tous les objets "vivants" sont stockés dans l'image.
- ▶ Le problème est similaire à celui de la sérialisation en Java
- ▶ Nous reprenons des éléments de [Hun98].

# BOSS

## Création

La procédure de création d'un fichier BOSS est relativement simple :

1. Créer un flot de données pour stocker les objets. C'est un flot d'écriture spécifié sur un nom de fichier.
2. Créer une instance de `BinaryObjectStorage` (BOSS) en utilisant par exemple la méthode de création `onNew` paramétré par le flux de données.
3. Chaque objet est stocké par le message `nextPut:`.
4. L'opération se termine par le message `close`.

---

```
valuesFileStream ← bosFileName asFilename writeStream.
bosFileStream ← BinaryObjectStorage onNew: valuesFileStream.
bosFileStream nextPutAll: aList.
valuesFileStream close.
```

---

avec test d'insertion

---

```
[bosFileStream nextPutAll: aList] valueNowOrOnUnwindDo:
[valuesFileStream close].
```

---

# BOSS

## Lecture

La procédure de lecture d'un fichier BOSS est inverse :

1. Créer un flot de données par le message `readStream` pour lire le fichier BOSS.
2. Créer un objet BOSS en envoyant un message de création `onOld:` avec le flux en paramètre ou plus simplement `onOldNoScan:` s'il n'y a pas d'écriture.
3. On récupère les objets par le message `next` ou `contents`.
4. L'opération se termine par le message `close`.

---

```

| valuesFileStream bosFileStream bosFileName aBag |
bosFileName ← 'fichier_bin'.
valuesFileStream ← bosFileName asFilename readStream.
bosFileStream ← BinaryObjectStorage onNew: valuesFileStream.
aBag ← Bag new.
[[ bosFileStream atEnd] whileFalse: [aBag add: bosFileStream next]]
valueNowOrOnUnwindDo: [valuesFileStream close].

```

---

## Généralités sur les entrées/sorties

### Entrée/sorties en Java

### Entrée/sorties en C++

### Entrée/sorties en Smalltalk

## Résumé

## Synthèse du chapitre

- ▶ Nous avons donné un aperçu des entrées-sorties dans les langages à objets. Les approches ont en commun l'abstraction via les flots de données (caractères, octets ou objets).
  - ▶ Flux standards ou fichiers et sérialisation en Java.
  - ▶ Flux standards ou fichiers et sérialisation via STL en C++.
  - ▶ Les entrées/sorties utilisent l'interface graphique de Smalltalk et les outils de compilation pour l'analyse du texte. Le reste se fait avec divers flux de données.
- ▶ On distingue le stockage de textes du stockage d'objet pour lequel des procédures propres au langage ou à son implantation sont mises en œuvre.
- ▶ La persistance à grande échelle se traite via les bases de données, et des interfaces appropriées.



Irène Charon.

*Le langage Java 2 - Concepts et pratique.*

Hermès, 2 édition, 2003.

ISBN 2-7462-0629-3.



Gilles et al. Clavel.

*Java, la synthèse - Concepts, architectures, frameworks.*

Dunod, 4 édition, 2003.

ISBN 2-10-007102-5.



John Hunt.

*Smalltalk and Object Orientation: An Introduction.*

Springer, 1998.



Peter Sestoft.

*Java Precisely.*

The MIT Press, 2 édition, 2005.

ISBN 0-262-69325-9.



Pierre-Yves Saumon and Antoine Mirecourt.

*Le guide du développeur Java 2 - Meilleures pratiques avec Ant, Junit et les design patterns.*

Eyrolles, 1 edition, 2003.

ISBN 2-212-11275-0.