Input/Output, sérialisation et réflection

**Entrées/sorties :** communication d’un programme avec son environnement

### Exemples d’entrées :

* clavier, souris
* scanner, caméra, etc.
* lecture d’un fichier sur disque
* réception d’une page web depuis un serveur distant

– …

### Exemples de sorties :

* affichage sur un écran
* écriture d’un fichier sur un disque local
* envoi d’une requête à un serveur
* envoi d’une commande à un robot
* impression d’un document vers un fax, une imprimante, etc.

# Autres difficultés

### Diversité des systèmes d’exploitation

* **Fichiers de natures différentes :**
  + texte : code source, script, configuration, courrier...
  + binaire : exécutables, fichiers compressés, etc.
  + spéciaux : /dev/ (périphériques)
  + répertoires : contient des références à d’autres fichiers

### Différents codages (fichier texte) :

* + latin1, utf8, ASCII, etc.
  + un caractère codé sur 1 octet, 2 octets, 4 octets, voir un nombre variable

– ...

# Solution : Abstraction des E/S

### Constatations :

* + - * Toute E/S peut être représentée par une suite de bits
      * Ceux-ci sont regroupés en octets (bytes)

### Abstraction des entrées/sorties : flux (stream) :

* + - * accès séquentiel aux données
      * flux d’entrée : InputStream
      * flux de sortie : OutputStream

### Abstraction des opérations possibles

* + - * opérations d’entrée (lecture : read)
      * opérations de sortie (écriture : write)

# Les entrées/sorties en Java

[**https://nanopdf.com/download/io-lotfi-chaari\_pdf**](https://nanopdf.com/download/io-lotfi-chaari_pdf)

L’API d’entrée/sortie est définie dans le paquetage java.io. Elle :

* fournit une interface standard pour gérer les flux d’entrée/sortie ;
* libère le programmeur des détails d’implantation liés à une plateforme particulière.

**Flux :** Séquence ordonnée de données qui a une source (input stream) ou une destination (output stream).

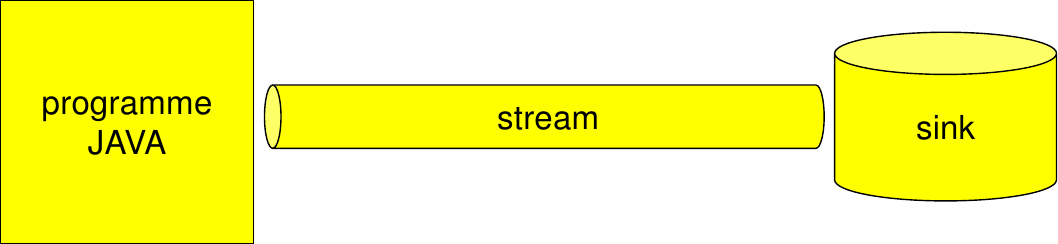
**Classes de base :** Java propose 4 classes principales (abstraites) pour les entrées/sorties qui seront ensuite spécialisées en fonction de la nature de la

source ou de la destination (Reader, Writer, InputStream, OutputStream).

# Les entrées/sorties en Java

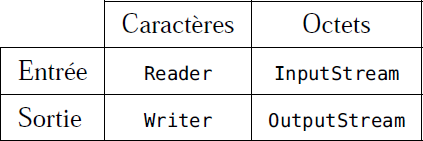
* JAVA gère les entrées/sortie en utilisant des objets appelés streams (flots)
* un stream est un flot de données entre le programme java et un container de données, appelé sink (évier)

\* un sink peut être une console (clavier/écran), un fichier, le réseau...

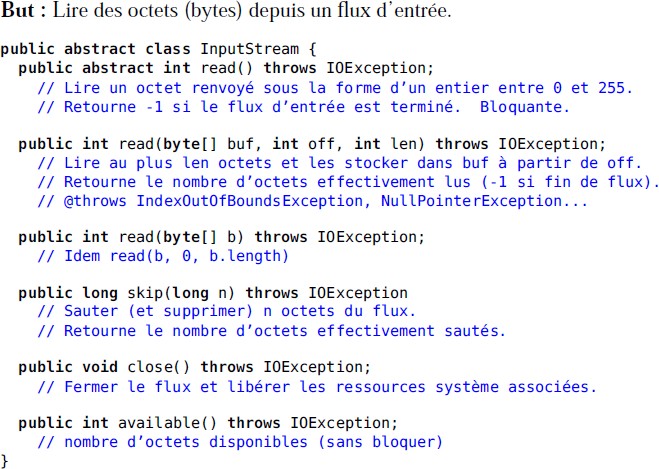


# Types de stream

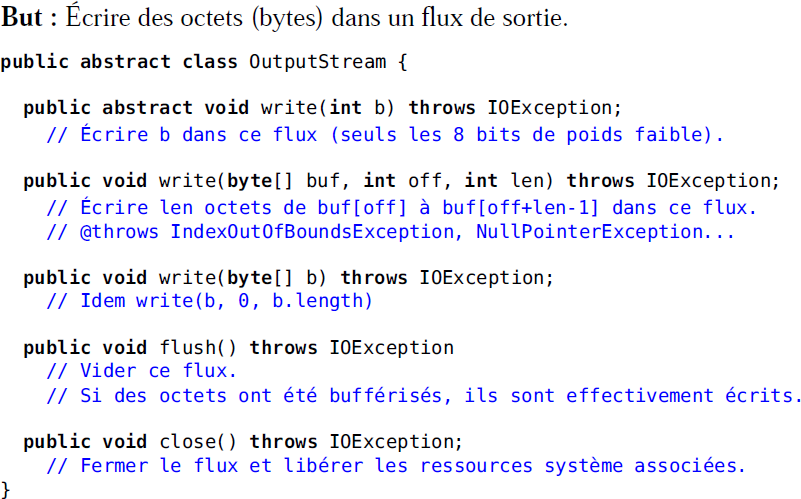
* + Il existe de multiples streams
  + Ils peuvent être classés par
    - sens du flot
      * entrée, sortie
    - type de données manipulées
      * données binaires, données caractères
    - nature de la manipulation des données
      * communication, traitement
    - avec ou sans tampon



**La classe** InputStream



**La classe** OutputStream



# Les classes Reader et Writer

**Principe :** Équivalentes à InputStream et OutputStream mais avec des caractères et non pas des octets.

public abstract class Reader {

public int read() throws IOException;

public abstract int read(char[] buf, int off, int len) throws IOException;

public int read(char[] b) throws IOException; public long skip(long n) throws IOException public abstract void close() throws IOException;

public boolean ready() throws IOException; // prêt à être lu ?

}

public abstract class Writer {

public void write(int b) throws IOException;

public abstract void write(char[] buf, int off, int len) throws IOException;

|  |  |  |  |
| --- | --- | --- | --- |
| public | void | write(char[] | b) throws IOException; |
| public | void | write(String | str) throws IOException; |
| public | void | write(String | str, int off, int len) throws |

IOException;

public void flush() throws IOException

public abstract void close() throws IOException;

} 9

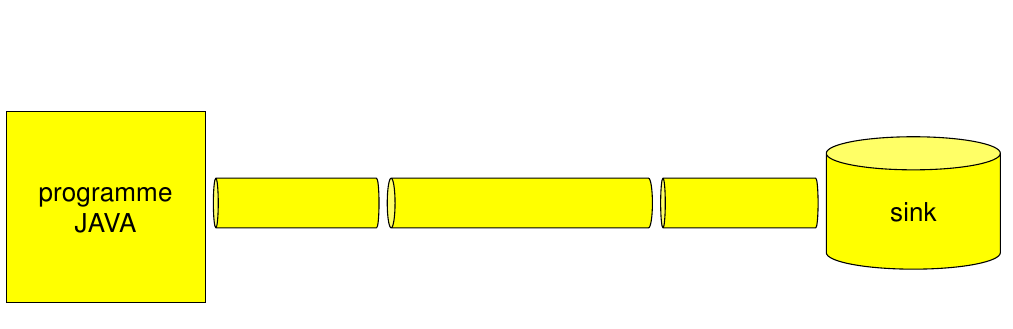
# Chaînage de streams

* L'intérêt des streams c'est qu'ils peuvent être chaînés

– un stream de communication peut être enchaîné avec un stream de traitement permettant de mettre en œuvre le traitement sur flot

–L'enchaînement se fait par la création d'objet stream prenant un autre stream comme paramètre

10



**Streams de caractères**

**Création d'un fichier texte**

* + FileWriter fw = new FileWriter("data.txt" );
  + Cette instruction écrase systématiquement le fichier data.txt

(s'il existe)

* + Il serait souhaitable de vérifier l'existence du fichier au préalable
    - pour cela on utilise un objet File

11

# File

* + Classe permettant de créer des objets qui décrivent des fichiers

File f = new File("Data.txt");

* + Attention, cette instruction ne crée pas le fichier data.txt. Elle permet juste d'accéder à ses informations systèmes (existence, Taille...)
  + Ainsi, on peut créer le fichier

FileWriter fw = new FileWriter(f);

**Opération sur un** FileWriter

* + Ecriture
* fw.write(int)
  + Fermeture
* fw.close(); 12

# Exemple

import java.io.\*; public class Fichier {

public static void main(String [] args) throws IOException

{

FileWriter fw = new FileWriter("Fichier.txt"); for( int i=0; i<args.length; i++){

fw.write(args[i]);

fw.write(‘\n’);

}

fw.close();

}

}

# Ecrire des voitures

class Voiture {

private String plaque, marque, modele; Voiture (String p, String ma, String mo){

|  |  |  |
| --- | --- | --- |
| plaque | = | p; |
| marque | = | ma; |
| modele | = | mo; |

}

public String toString(){

return plaque + " : " + marque + " : " + modele;

}

//getters et setters ici

}

* + Ecrire un programme qui crée des objets de type Voiture et écrit leurs représentations String dans un fichier texte.

import java.io.\*;

public class TestVoitures {

public static void main(String [] args) throws IOException

{ String p = "AA 111 BB"; String ma = "Peugeot"; String mo = "207";

Voiture v1 = new Voiture(p, ma, mo); FileWriter fw = new FileWriter("voiture.txt"); fw.write(v1.getPlaque());

fw.write("\n"); fw.write(v1.getMarque()); fw.write("\n"); fw.write(v1.getModele()); fw.close();

String voit = v1.toString(); System.out.println(voit);

}

}

# Exercice

* + Écrire un programme qui génère un nombre donné de couples **login/password** qu'il stocke dans un fichier. Il s'exécute comme suit :

$java Password n m fichier

\*n est le nombre de couples login/password

\*m est la longueur du mot

\*fichier le fichier de stockage

Les logins seront de la forme user1, user2, …

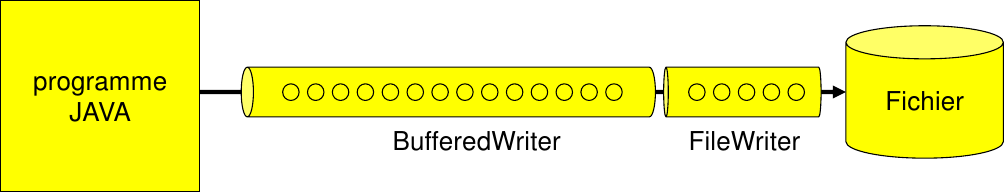
On utilisera une Map pour stocker les couples <login,password>

# Utilisation d'un tampon

* + L'accès au disque est plus lent que l'accès à la mémoire
  + L'écriture serait plus rapide en utilisant un tampon (buffer)
  + Pour cela, on enveloppe le stream FileWriter dans un stream

BufferedWriter

# BufferedWriter



**Créer un stream de sortie bufférisé**

* + File f = new File("donnees.txt");
  + FileWriter fw = new FileWriter (f);
  + BufferedWriter b = new BufferedWriter(fw);
  + Ecriture plus compacte

BufferedWriter b = new BufferedWriter(new FileWriter (f));

# Manipulation d'un BufferedWriter

* + Ecriture
* b.write(String);
* b.write(char[]);
* b.write(int );
  + Fermeture
* b.close();

# Exemple :

import java.io.\*; public class Fichier {

public static void main(String [] args) throws

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| IOException | { |  | | |
| FileWriter | fw | = | new | FileWriter("Fichier.txt"); |

BufferedWriter bw = new BufferedWriter (fw); for( int i=0; i<args.length; i++){

bw.write(args[i]);

bw.write("\n");

}

bw.close();

}

}

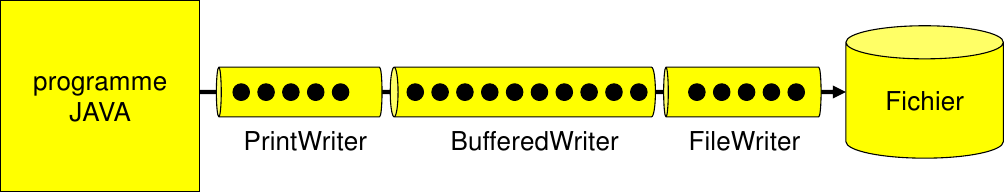
# Exercice :

Modifier les programmes précédents pour utiliser un buffer.

# Sortie formatée

* + Les streams FileWriter et BufferedWriter permettent d'écrire des caractères (unicode) ou des chaînes de caractères
  + Le programmeur souvent souhaite écrire des données formatées (int, long, float...)
  + PrintWriter est un stream qui permet cela
    - on enveloppe le BufferedWriter dans un PrintWriter

# PrintWriter



* **Créer un fichier de sortie formatée**
* File f = new File("donnees.txt");
* FileWriter fw = new FileWriter (f);
* BufferedWriter b = new BufferedWriter(fw);
* PrintWriter p = new PrintWriter (b, true);

Ecriture plus compacte

PrintWriter p = new PrintWriter (

new BufferedWriter(new FileWriter (f)),true);

# Ecriture

* p.print(int); p.print(float); p.print(boolean); p.print(String)...
* p.println(int); p.println(float); p.println(boolean); p.println(String);
* p.write(String); p.write(int);

# Fermeture

* p.close();

# Exemple

import java.io.\*; public class Fichier {

public static void main(String [] args) throws IOException {

FileWriter fw = new FileWriter("Fichier.txt"); BufferedWriter bw = new BufferedWriter (fw); PrintWriter pw = new PrintWriter (bw, true); for( int i=0; i<10; i++){

pw.println(i);

}

pw.close();

}

}

# Récapitulatif des streams de sortie de caractères

### Streams de communication

* + PipedWriter
    - connexion entre une sortie et une entrée
  + CharArrayWriter
    - écriture avec un tampon indexé
  + StringWriter
    - écriture dans une chaîne
  + FileWriter
    - sous-classe de OutputStreamWriter avec tampon et encodage par

défaut

### Streams de traitement

* + OutputStreamWriter
    - convertit un stream binaire en caractère
  + FilterWriter (classe abstraite)
    - parent de classes de traitement
  + BufferedWriter
    - écriture de caractères avec tampon
  + PrintWriter
    - écriture de caractères formatés 23

# Ouverture d'un fichier texte

* FileReader fr = new FileReader("data.txt" );
* Cette instruction suppose que le fichier existe. Une erreur se

produira si le fichier n'existe pas.

* Il serait souhaitable de vérifier l'existence du fichier au préalable

– pour cela on utilise un objet File

# File

File f = new File("data.txt");

* Attention, cette instruction n'ouvre pas le fichier data.txt. Elle permet juste d'accéder à ses informations systèmes (existence, taille...)
* Ainsi, on peut créer le stream vers le fichier

FileReader fr = new FileReader(f);

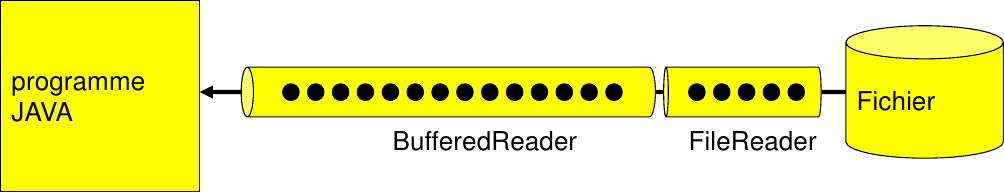
# Opération sur un FileReader

* + Lecture
    - int n = fr.read( ); //lit un caractère
  + Fermeture
    - fr.close();

# Utilisation d'un tampon

* On accélère la lecture en utilisant un tampon
* Pour cela, on enveloppe le stream FileReader dans un stream

BufferedReader



# Créer un stream d'entrée bufférisé

* File f = new File("data.txt");
* FileReader fr = new FileReader (f);
* BufferedReader br = new BufferedReader(fr);

Ecriture plus compacte

BufferedReader br = new BufferedReader(new FileReader (f));

# Manipulation d'un BufferedReader

* Ecriture
* int c = br.read();
* String s = br.readLine();
* Fermeture
* br.close();

# Récapitulatif des streams d'entrée de caractères

### Streams de communication

* + PipedReader
    - connexion entre une sortie et une entrée
  + CharArrayReader
    - lecture avec un tampon indexé
  + StringReader
    - lire à partir d'une chaîne
  + FileReader
    - sous-classe de InputStreamReader avec tampon et encodage par

défaut

### Streams de traitement

* + InputStreamReader
    - convertit un stream binaire en caractère
  + FilterReader (classe abstraite)
    - PushBackReader : lecture avec revoie du dernier caractère lu
  + BufferedReader
    - lit des caractères avec tampon
    - LineNumberReader : lire des caractères en comp2t7ant les lignes

# Exercice

* Ecrire un « Grep » qui permet de trouver un motif (ensemble de caractères) à l'intérieur d'un fichier, et d'afficher les lignes contenant ce motif ainsi que le numéro de la ligne dans le fichier.
* Lire le fichier de voitures et recréer les objets Voiture.

28

import java.io.\*; public class Grep {

public static void main(String[] args) throws IOException {

assert args.length > 0; String motif = args[0];

for (int i = 1; i < args.length; i++) { grep(".\*" +motif+".\*", args[i]);

}

}

public static void grep(String motif, String fichier) throws IOException {

BufferedReader in = null; try {

in = new BufferedReader(new FileReader(fichier)); int numero = 1;

String ligne;

while ((ligne = in.readLine()) != null) { if (ligne.matches(motif)) {

System.out.println(fichier + ":" + numero + " " +

ligne);

}

}

numero++;

} finally {

if (in != null) in.close();

}

} 29

}

import java.io.\*;

public class TestVoitures {

public static void main(String [] args) throws IOException

{String p = "AA 111 BB"; String ma = "Peugeot"; String mo = "207";

Voiture v1 = new Voiture(p, ma, mo);

// écriture

FileWriter fw = new FileWriter("voiture.txt"); fw.write(v1.getPlaque());

fw.write("\n"); fw.write(v1.getMarque()); fw.write("\n"); fw.write(v1.getModele()); fw.close();

//lecture

Voiture v2 = new Voiture("", "", ""); File f = new File("voiture.txt");

BufferedReader br = new BufferedReader(new FileReader (f)); String entree;

entree = br.readLine(); v2.setPlaque(entree); entree = br.readLine(); v2.setMarque(entree); entree = br.readLine(); v2.setModele(entree);

System.out.println(v1.toString()); System.out.println(v2.toString());

}

}

**Streams d'octets**

**Lecture d'un fichier binaire**

* Stream simple

## FileInputStream f = new FileInputStream("img.gif");

* lecture avec f.read();
* Stream avec tampon
* BufferedInputStream bis = new BufferedInputStream(f);
* lecture avec bis.read();
* Stream avec lecture de primitifs java

## DataInputStream in = new DataInputStream(bis);

* lecture avec readBoolean, readByte, readChar, readDouble,readFloat, readInt, readLong, readShort

# Ecriture d'un fichier binaire

* Stream simple

## FileOutputStream f = new FileOutputStream("img.gif");

* écriture avec f.write();
* Stream avec tampon

## BufferedOutputStream bis = new BufferedOutputStream(f);

* écriture avec bis.write();
* Stream avec écriture de primitifs java

## DataOutputStream out = new DataOutputStream(bis);

* écriture avec writeBoolean, writeByte, writeChar, writeDouble, writeFloat, writeInt, writeLong, writeShort

# Ecriture à l'écran

* La classe java.lang.System fournit un objet (out) de type

PrintStream représentant la sortie standard

* Il peut être utilisé directement pour afficher des caractères à l'écran
* System.out.print()
* System.out.println()

# Lecture au Clavier

* La classe java.lang.System fournit un objet (in) de type InputStream

représentant l'entrée standard

* InputStream est une classe abstraite, super-classe de toutes les classes de streams d'entrée binaire
* On peut convertir cet objet en stream d'entrée de caractères.

BufferedReader stdin = new BufferedReader ( new InputStreamReader( System.in) );

* Ainsi on peut lire des caractères depuis le clavier en utilisant les

méthodes de BufferedReader

# Exercices

* + Écrire un programme qui lit deux nombres au clavier et qui les imprime dans l'ordre à l'écran.
  + Créer un codeur / décodeur de fichiers binaires qui inverse chaque octet.

import java.io.\*; import java.lang.System;

public class IO\_Clavier {

public static int lireEntier(BufferedReader stdin) { try {return(Integer.parseInt(stdin.readLine())); } catch (Exception e)

{ System.out.println("Erreur lecture entier"); return 0;

}

}

public static float lireFloat(BufferedReader stdin) {

try {return(Float.valueOf(stdin.readLine()).floatValue());

}

catch (Exception e)

{ System.out.println("Erreur lecture réel (float)"); return (float) 0.0;

}

}

public static void main(String[] args) throws IOException { BufferedReader stdin = new BufferedReader(new

InputStreamReader(System.in)); int entier1,entier2;

entier1 = lireEntier(stdin); System.out.println("Tapez un deuxième entier\n"); entier2 = lireEntier(stdin);

if (entier1 < entier2){ System.out.println(entier1 + "<=" + entier2);}

else{System.out.println(entier2 + "<=" + entier1);}

} 35

}

# Récapitulatif des streams de sortie d'octets

* **Streams de communication**
* FileOutputStream
  + écriture séquentielle d'octets dans un fichier
* PipedOutputStream
  + connecter un PipedInputSTream et un stream de sortie
* ByteArrayOutputStream
  + écriture d'octets avec un tampon indexé
* **Streams de traitement**
* FilterOutputStream
  + BufferedOutputStream : écrit avec un tampon
  + CkeckedOutputStream : écrit et vérifie l'intégrité
  + DataOutputStream : écrit au format des types java
  + DigestOutputStream : écriture avec création d'une table de hachage permettant de vérifier l'intégrité avec DigestInputStream
  + DeflaterOutputStream (écriture avec compression)
    - GZIPOutputStream
    - ZipOutputStream
    - JarOutputStream
  + PrintStream : écriture avec conversion en octets (en fonction du système

hôte)

* ObjectOutputStream : écrit un objet sérialisé

36

# Récapitulatif des streams d'entrée d'octets

* **Streams de communication**
* FileInputStream
  + lecture séquentielle d'octets dans un fichier
* PipedInputStream
  + connecter un stream d'entrée à PipedOutputSTream
* ByteArrayInputStream
  + lecture d'octets avec un tampon indexé
* **Streams de traitement**
* FilterInputStream
  + BufferedInputStrea : lit avec un tampon
  + CkeckedInputStream (java.util.zip) : lit et vérifie l'intégrité
  + DataInputStream : lit au format des types java
  + DigestInputStream (java.security) : lecture et vérification à l'aide de table de hachage créée par DigestOutputStream
  + InflaterInputStream (lecture avec décompression)
    - GZIPInputStream
    - ZipInputStream
    - JarInputStream
  + ProgressMonitorInputStream (java.swing) : lecture et affichage de barre de progression
  + PushBackInputStream
* SequenceInputStream
* ObjectInputStream : lit un objet sérialisé 37

# Design pattern Decorator

* **Limitations de la notion d’héritage :**
* Changement d’un comportement après instanciation?
* Extensions multiples d’une même classe?
* Ajout d’une capacité seulement à certaines instances de la Classe?

# Solution :

- Créer une classe apportant une réponse de fonctionnalité

# Avantages :

Empêche l’explosion des sous-classes

Permet d’attribuer des capacités individuellement, dynamiquement et de manière transparente

# Exemple :

-Décoration d’un InputStream par un BufferedInputStream

# Décorer un décorateur:

C’est possible parce que, selon le pattern décorateur,

* + le décorateur décore un InputStream
  + le décorateur et le décoré **sont des** InputStream (par héritage)

# Exemple :

* + FileInputStream : classe de base pour la lecture d’un fichier ; décorée par un
  + BufferedInputStream : décorateur qui ajoute un buffer pour la lecture du flot ; décoré par un
  + DataInputStream : décorateur qui décode les types primitifs Java codés dans un format standard, indépendant du système

# Remarques :

* Utile surtout lors de la conception de classes qui risquent d’évoluer fortement (ajout ou modification de fonctionnalités)
* Quand un décorateur reçoit un message, il remplit sa fonctionnalité (la « décoration »). **En cas de besoin, il fait appel à l’objet décoré pour remplir les fonctionnalités de base.**

Decorateur.java

Abstraction.java

/\*\*

\*Définit l'interface générale.

\*/

public interface Abstraction {

/\*\*

\* Méthode générale.

\*/

public void operation();

}

/\*\*

* + Définit l'interface du décorateur.

\*/

public abstract class Decorateur

implements Abstraction { protected Abstraction abstraction;

/\*\*

\* Le constructeur du "Decorateur" reçoit

* + - un objet "Abstraction"
    - @param pAbstraction

\*/

public Decorateur(final Abstraction pAbstraction){

abstraction = pAbstraction;

Implementation.java }

/\*\* }

* Implémente l'interface générale.

\*/

public class Implementation implements Abstraction {

/\*\*

* Implémentation de la méthode
* pour l'opération de base

\*/

public void operation() { System.out.println("Implementation");

}

} 41

# Sérialisation

Java permet d'écrire ou de lire directement des objets dans un fichier. On utiliser ce système pour sauvegarder un objet complexe, presque sans effort.

On peut, par exemple, écrire :

TreeSet s;

// Code qui remplit s

//...

ObjectOutputStream o= new ObjectOutputStream(New FileOutputStream("toto.sav"));

o.writeObject(s); o.close();

Le contenu de notre TreeSet sera sauvé dans le chier « toto.sav ». Pour relire les donnée:

ObjectInputStream f= new ObjectInputStream(New FileInputStream("toto.sav"));

TreeSet s= (TreeSet)f.readObject();

* 1. lose(); 42

# Conditions d'emploi

* + Pour sauver un objet dans un ObjectOutputStream, la classe de l'objet doit implémenter l'interface Serializable.
  + De plus, les champs de l'objet doivent eux aussi appartenir à des classes Serializable.

-Ainsi, comme TreeSet est Serializable (voir doc) notre exemple précédent ne fonctionnera que si les données contenues dans le TreeSet sont elles mêmes Serializable.

* + Il est donc conseillé de déclarer Serializable toute classe potentiellement utilisable avec la sérialisation.
  + Si une classe est Serializable, toutes ses héritières le seront.

# Exemple :

public class Personne implements java.io.Serializable { private String nom = "";

private String prenom = ""; private int taille = 0; private int age = 0;

public Personne(String nom, String prenom, int age, int taille) { this.nom = nom;

this.taille = taille; this.prenom = prenom; this.age = age;

}

public String toString() {

return prenom + " " + nom + " " + age + " ans, " + taille + " cm";} public String getNom() {return nom;}

public void setNom(String nom) {this.nom = nom;} public int getTaille() {return taille;}

public void setTaille(int taille) {this.taille = taille;} public String getPrenom() {return prenom;}

public void setPrenom(String prenom) {this.prenom = prenom;} public int getAge() {return age;}

public void setAge(int age) { this.age = age;} public void AfficheToi(){

System.out.println("Je m'appelle " + this.prenom + " " + this.nom + ", j'ai "

+ this.age + " ans, je mesure " + this.taille + " cm.");

}

}

import java.io.\*;

public class Serialisation {

public static void main(String argv[]) {

Personne personne = new Personne("Dupond","Jean",30,170); System.out.println("creation de : " + personne);

Personne personne2 = new Personne("Dupond2","Jean2",31,171); System.out.println("creation de : " + personne2);

Personne personne3 = new Personne("Dupond3","Jean3",32,172); System.out.println("creation de : " + personne3);

//Sérialisation

try { FileOutputStream fichier = new FileOutputStream("personne.ser"); ObjectOutputStream oos = new ObjectOutputStream(fichier); oos.writeObject(personne);

oos.flush(); oos.writeObject(personne2); oos.flush(); oos.writeObject(personne3); oos.flush();

oos.close();

System.out.println(personne + " a ete serialise");

}

catch (java.io.IOException e) {e.printStackTrace(); }

//Désérialisation

try { FileInputStream fichier = new FileInputStream("personne.ser"); ObjectInputStream ois = new ObjectInputStream(fichier); Personne personneb = (Personne) ois.readObject(); personneb.AfficheToi();

Personne personne2b = (Personne) ois.readObject(); personne2b.AfficheToi();

Personne personne3b = (Personne) ois.readObject(); personne3b.AfficheToi();

}

catch (java.io.IOException e) { e.printStackTrace(); } catch (ClassNotFoundException e) { e.printStackTrace(); }

} 45

}

creation de : Jean Dupond 30 ans, 170 cm creation de : Jean2 Dupond2 31 ans, 171 cm creation de : Jean3 Dupond3 32 ans, 172 cm Jean Dupond 30 ans, 170 cm a ete serialise

Je m'appelle Jean Dupond, j'ai 30 ans, je mesure 170 cm. Je m'appelle Jean2 Dupond2, j'ai 31 ans, je mesure 171 cm. Je m'appelle Jean3 Dupond3, j'ai 32 ans, je mesure 172 cm.

46

# Comment interdire la sérialisation ?

* + Le mot clé **transient** permet d'interdire la sérialisation d'un attribut d'une classe. Il est en général utilisé pour les données "sensibles" telles que les mots de passe ou tout simplement pour les attributs n'ayant pas besoin d'être sérialisé.

public class Personne implements java.io.Serializable { private String nom = "";

private String prenom = ""; private int taille = 0; transient private int age = 0;

.

.

.

}

Testez...

creation de : Jean Dupond 30 ans, 170 cm creation de : Jean2 Dupond2 31 ans, 171 cm creation de : Jean3 Dupond3 32 ans, 172 cm Jean Dupond 30 ans, 170 cm a ete serialise

Je m'appelle Jean Dupond, j'ai **0 ans**, je mesure 170 cm. 47

Je m'appelle Jean2 Dupond2, j'ai **0 ans**, je mesure 171 cm.

Je m'appelle Jean3 Dupond3, j'ai **0 ans**, je mesure 172 cm.

# Sérialisation XML

* + Les données générées sont dans un format textuel (XML) et non binaire.

### Avantages :

* + La portabilité : il n'y a aucune dépendance avec l'implémentation propre des classes qui peuvent donc être échangées entre différentes MVs.
  + Les documents XML générés sont compacts grâce à l'utilisation d'un algorithme d'élimination des redondances (par exemple les attributs ayant leur valeur par défaut ne sont pas écrits).
  + La tolérance aux fautes : si une erreur non structurelle est détectée, la partie affectée n'est pas prise en compte.
  + Étant un document XML, le résultat est lisible par l'humain et exploitable par d'autres programmes.

### Inconvénients :

* + La classe à sérialiser doit posséder un constructeur par défaut (sans paramètre).
  + Les attributs de la classe à sérialiser doivent être accessibles via des accesseurs/modifieurs.
  + Le résultat de la sérialisation XML est plus gros que celui de la sérialisation binaire.
  + La sérialisation XML repose principalement sur les classes XMLEncoder et XMLDecoder qui permettent de procéder à la sérialisation et la désérialisation d'objets.

**XMLEncoder :** classe qui permet de sérialiser un objet vers un document XML. Son utilisation est très simple et similaire à celle de l'ObjectOutputStreamde la sérialisation binaire.

Nous allons créer une classe XMLTools contenant une méthode statique permettant de sérialiser un objet dans un fichier.

import java.beans.PersistenceDelegate; import java.beans.XMLEncoder;

import java.io.FileOutputStream; import java.io.FileNotFoundException; import java.io.IOException;

public final class XMLTools { private XMLTools() {}

public static void encodeToFile(Object object, String fileName) throws

FileNotFoundException, IOException {

// ouverture de l'encodeur vers le fichier

XMLEncoder encoder = new XMLEncoder(new FileOutputStream(fileName)); try {// serialisation de l'objet

encoder.writeObject(object); encoder.flush();

System.out.println(object + " a ete serialise");

} finally {

// fermeture de l'encodeur encoder.close();

}

}

public static Object decodeFromFile(String fileName) throws FileNotFoundException, IOException {

Object object = null;

// ouverture de decodeur

XMLDecoder decoder = new XMLDecoder(new FileInputStream(fileName)); try {// deserialisation de l'objet

object = decoder.readObject(); System.out.println(object.toString() + " a ete deserialise");

} finally {

// fermeture du decodeur decoder.close();

}

return object;

} 50

}

**Dans le main...**

//Sérialisation XML

Personne p = new Personne("Dupond","Nicolas",30,170); System.out.println(p); System.out.println("Sérialisation XML");

try {

XMLTools.encodeToFile(p, "Personne.xml");

} catch(Exception e) { e.printStackTrace();

}

//Désérialisation XML System.out.println("Désérialisation XML");

try {

p = new Personne("aaa", "bbb",40,160); System.out.println(p);

p = (Personne) XMLTools.decodeFromFile("Personne.xml"); System.out.println(p);

} catch(Exception e) { e.printStackTrace();

}

**Personne.xml**

<java version="1.7.0" class="java.beans.XMLDecoder">

<object class="serialisation.Personne">

<void property="age">

<int>30</int>

</void>

<void property="nom">

<string>Dupond</string>

</void>

<void property="prenom">

<string>Nicolas</string>

</void>

<void property="taille">

<int>170</int>

</void>

</object>

</java>

**Remarques :**

* + Obligation d'avoir des getters et des setters pour tous les attributs
  + Obligation d'avoir un constructeur par défaut

**Sérialisation XML avec JDOM (*Java Document Object Model*):**

Il vous faut dans un premier temps télécharger la dernière version de JDOM disponible à cette adresse : <http://www.jdom.org/dist/binary/>*(version 1.1.3)*

**Objectif** : Manipuler des donnés XML **+ simplement** qu'avec les API classiques.

### Origines :

* + **SAX** (*Simple API for XML)* : ce type de parseur utilise des événements pour piloter le traitement d'un fichier XML (**gestion évènementielle**).
  + Un objet doit implémenter des méthodes particulières définies dans une interface de l'API pour fournir les traitements à réaliser : selon les événements, le parseur appelle ces méthodes.
  + **DOM :** utilise des collections SAX pour parser les fichiers XML (spécification pour proposer une API qui permet de modéliser, de parcourir et de manipuler un document XML : **gestion par arborescence**).
  + **JDOM** (spécifique à Java)**:** construire des documents, naviguer dans leur structure, ajouter, modifier, ou supprimer leur contenu : **gestion par arborescence**.

**Création d'un fichier XML en partant de zéro** : il suffit de construire chaque élément puis de les ajouter les uns aux autres de façon logique. Un nœud est une instance de org.jdom.Element.

<personnes>

<etudiant classe="P2">

<nom>CynO</nom>

53

<etudiant>

<personnes>

JDOM1.java

import java.io.\*; import org.jdom.\*;

import org.jdom.output.\*; public class JDOM1

{ //Nous allons commencer notre arborescence en créant la racine XML

//qui sera ici "personnes".

static Element racine = new Element("personnes");

//On crée un nouveau Document JDOM basé sur la racine que l'on vient de créer

static org.jdom.Document document = new Document(racine);

public static void main(String[] args)

{

//On crée un nouvel Element etudiant et on l'ajoute

//en tant qu'Element de racine

Element etudiant = new Element("etudiant"); racine.addContent(etudiant);

//On crée un nouvel Attribut classe et on l'ajoute à etudiant

//grâce à la méthode setAttribute

Attribute classe = new Attribute("classe","P2"); etudiant.setAttribute(classe);

//On crée un nouvel Element nom, on lui assigne du texte

//et on l'ajoute en tant qu'Element de etudiant Element nom = new Element("nom"); nom.setText("CynO");

etudiant.addContent(nom);

affiche(); enregistre("Exercice1.xml");

}

} 54

### Afficher et enregistrer son fichier XML

Pour afficher puis enregistrer notre arborescence, nous allons utiliser une unique classe pour ces deux flux de sortie : org.jdom.output.XMLOutputter, qui prends en argument un org.jdom.output.Format.

En plus des trois formats par défaut (PrettyFormat, CompactFormat et RawFormat), la classe Format contient une panoplie de méthodes pour affiner votre sérialisation.

**Class to encapsulate XMLOutputter format options**

**Outputs a JDOM document as a stream of bytes**

55

JDOM1.java

Ajouter ces deux méthodes à notre classe JDOM1

static void affiche()

{

try

{ //On utilise ici un affichage classique avec getPrettyFormat() XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat()); sortie.output(document, System.out);

}

catch (java.io.IOException e){}

}

static void enregistre(String fichier)

{

try

{ //On utilise ici un affichage classique avec getPrettyFormat() XMLOutputter sortie = new XMLOutputter(Format.getPrettyFormat());

//Remarquez qu'il suffit simplement de créer une instance de FileOutputStream

//avec en argument le nom du fichier pour effectuer la sérialisation. sortie.output(document, new FileOutputStream(fichier));

}

catch (java.io.IOException e){}

}

Voici le résultat obtenu (affichage sur la sortie standard et contenu du fichier « Exercice1.xml ») :

<?xml version="1.0" encoding="UTF-8"?>

<personnes>

<etudiant classe="P2">

<nom>CynO</nom>

</etudiant>

</personnes> 56

### Parcourir un fichier XML

* + Parser un fichier XML revient à transformer un fichier XML en une arborescence JDOM.
  + Nous utiliserons pour cela le constructeur SAXBuilder, basé, comme son nom l'indique, sur l'API SAX (Simple API for XML.).

Créez tout d'abord le fichier suivant dans le répertoire contenant votre future classe JDOM2 :

Exercice2.xml

<?xml version="1.0" encoding="UTF-8"?>

<personnes>

<etudiant classe="P2">

<nom>CynO</nom>

<prenoms>

<prenom>Nicolas</prenom>

<prenom>Laurent</prenom>

</prenoms>

</etudiant>

<etudiant classe="P1">

<nom>Superwoman</nom>

</etudiant>

<etudiant classe="P1">

<nom>Don Corleone</nom>

</etudiant> 57

</personnes>

* + Notre objectif ici est d'afficher dans un premier temps le nom de tous les élèves. Nous allons créer pour cela une nouvelle classe: JDOM2.

JDOM2.java

import java.io.\*;

import org.jdom.\*;

import org.jdom.input.\*; import org.jdom.filter.\*; import java.util.List; import java.util.Iterator;

import org.jdom.input.SAXBuilder;

**Builds a JDOM Document using a SAX parser.**

public class JDOM2

{ static org.jdom.Document document; static Element racine;

public static void main(String[] args)

{//On crée une instance de SAXBuilder SAXBuilder sxb = new SAXBuilder(); try

**This builds a document from the supplied filename.**

{//On crée un nouveau document JDOM avec en argument le fichier XML

//Le parsing est terminé

document = sxb.build(new File("Exercice2.xml"));

}

catch(Exception e){}

//On initialise un nouvel élément racine avec l'élément racine du document. racine = document.getRootElement();

afficheALL();

}

58

}

static void afficheALL()

{

//On crée une List contenant tous les noeuds "etudiant" de l'Element racine List listEtudiants = racine.getChildren("etudiant");

//On crée un Iterator sur notre liste Iterator i = listEtudiants.iterator(); while(i.hasNext())

{

//On recrée l'Element courant à chaque tour de boucle afin de

//pouvoir utiliser les méthodes propres aux Element comme :

//sélectionner un nœud fils, modifier du texte, etc... Element courant = (Element)i.next();

//On affiche le nom de l’élément courant System.out.println(courant.getChild("nom").getText());

}

}

A l'exécution vous devriez voir s'afficher CynO, Superwoman et Don Corleone.

### Filtrer les éléments

* + Notre nouvel objectif est d'afficher la classe des étudiants dont le prénom est Laurent et le nom est CynO.
  + Les seuls filtres que nous avons faits pour le moment étaient directement implémentés dans les méthodes que nous utilisions.
    - List listEtudiants = racine.getChildren("etudiant") nous a permis de filtrer les éléments de racine selon leur nom.
  + Les filtres permettent des sélections d'éléments selon plusieurs critères.
  + Nous allons donc créer un filtre qui permettra de ne prendre en compte que les éléments qui possèdent :
    1. Un sous élément nom qui doit avoir pour valeur "CynO"
    2. Un sous élément prenom qui doit posséder au moins un sous élément prenom dont la valeur est "Laurent"
  + Une fois le filtre créé nous pourrons récupérer une liste contenant les éléments répondant à ces critères.

JDOM2.java

//Ajouter cette méthode à la classe JDOM2

//Remplacer la ligne afficheALL(); par afficheFiltre(); static void afficheFiltre()

{ //On crée un nouveau filtre Filter filtre = new Filter()

{ //On défini les propriétés du filtre à l'aide

//de la méthode matches

public boolean matches(Object ob)

{ //1 ère vérification : on vérifie que les objets

//qui seront filtrés sont bien des Elements if(!(ob instanceof Element)){return false;}

//On crée alors un Element sur lequel on va faire les

//vérifications suivantes. Element element = (Element)ob;

//On crée deux variables qui vont nous permettre de vérifier

//les conditions de nom et de prenom int verifNom = 0;

int verifPrenom = 0;

//2 ème vérification: on vérifie que le nom est bien "CynO" if(element.getChild("nom").getTextTrim().equals("CynO"))

{

verifNom = 1;

}

//3 ème vérification: on vérifie que CynO possède un prenom "Laurent"

//On commence par vérifier que la personne possède un prenom Element prenoms = element.getChild("prenoms");

if(prenoms == null){return false;}

//On constitue une liste avec tous les prénoms List listprenom = prenoms.getChildren("prenom");

//On effectue la vérification en parcourant notre liste de prénoms Iterator i = listprenom.iterator();

while(i.hasNext())

{

Element courant = (Element)i.next(); if(courant.getText().equals("Laurent"))

{

verifPrenom = 1;

}

}

//Si nos conditions sont remplies on retourne true, false sinon if(verifNom == 1 && verifPrenom == 1)

{

return true;

}

return false;

}

};//Fin du filtre

//getContent va utiliser notre filtre pour créer une liste d'étudiants répondant

//à nos critères.

List resultat = racine.getContent(filtre);

//On affiche enfin l'attribut classe de tous les éléments de notre list Iterator i = resultat.iterator();

while(i.hasNext())

{

Element courant = (Element)i.next(); System.out.println(courant.getAttributeValue("classe"));

}

}

À l'exécution vous devriez voir s'afficher P2 à votre écran.

# Réflexion

* + En programmation, la réflexion est la capacité d'un programme à examiner, et éventuellement à modifier, ses structures internes de haut niveau (par exemple ses objets) lors de son exécution.
  + La réflexion désigne la possibilité de manipuler objets, classes, méthodes et champs de façon dynamique.
  + La classe centrale permettant la mise en œuvre de la réflexion est la classe :

Class.

* + Dans cette classe, figurent toutes les méthodes permettant d’accéder aux (définitions de) membres de la classe.

# Possibilités de la réflexion

On distingue deux techniques utilisées par les systèmes réflexifs :

* + L'**introspection**, qui est la capacité d'un programme à **examiner** son propre état. Elle est utilisée pour effectuer des mesures de performance, inspecter des modules ou déboguer un programme.
  + L'**intercession**, qui est la capacité d'un programme à **modifier** son propre état d'exécution ou d'altérer sa propre interprétation ou signification. Elle permet à un programme d'évoluer automatiquement en fonction des besoins et de l'environnement.

### Intérêt :

* + La réflexion sera particulièrement utile si vous projetez de développer des outils de développements tels que :
    - des debuggers
    - des navigateurs (browsers) de classes
    - des environnements de développement d’interfaces graphiques

### La classe Class

[***http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Class.html***](http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Class.html)

* + Pour toute classe utilisée par une application, un (seul) objet particulier instance de **Class** est créé par la JVM.
  + Étant instance de la classe **Class**, il est possible d’utiliser toutes les méthodes définies dans cette classe.
  + En fait, l'objet **Class** est utilisé pour créer tous les objets « habituels » d'une classe.
  + Il y a un objet **Class** pour chacune des classes d'un programme. A chaque fois qu'une classe est écrite et compilée, un unique objet de type **Class** est aussi créé
  + Durant l'exécution, lorsqu'un nouvel objet de cette classe doit être créé, la JVM qui exécute le programme vérifie d'abord si l'objet **Class** associé est déjà chargé.
  + Si non, la JVM le charge en cherchant un fichier **.class** du même nom. Ainsi un programme Java n'est pas totalement chargé en mémoire lorsqu'il démarre, contrairement à beaucoup de langages classiques.
  + Une fois que l'objet **Class** est en mémoire, il est utilisé pour créer tous les objets

de ce type. 65

### La classe Class

* + **Class.forName("NomClasse");**

Cette méthode est une méthode **static** de **Class** (qui appartient à tous les objets

**Class**).

* + Un objet **Class** est comme tous les autres objets, il est donc possible d'obtenir sa référence et de la manipuler (c'est ce que fait le chargeur de classes).
  + Un des moyens d'obtenir une référence sur un objet **Class** est la méthode **forName()**, qui prend en paramètre une chaîne de caractères contenant le nom de la classe dont vous voulez la référence. Elle retourne une référence sur un objet **Class**.
  + La classe **Class** supporte le concept de *réflexion*, et une bibliothèque additionnelle, **java.lang.reflect**, contenant les classes **Field**, **Method**, et **Constructor** (chacune implémentant l'interface **Member**).
  + Les objets de ce type sont créés dynamiquement par la JVM pour représenter les membres correspondants d'une classe inconnue.

### La classe Class

* + On peut alors utiliser les constructeurs pour créer de nouveaux objets, les méthodes **get()** et **set()** pour lire et modifier les champs associés à des objets **Field**, et la méthode **invoke()** pour appeler une méthode associée à un objet **Method**.
  + De plus, on peut utiliser les méthodes très pratiques **getFields()**, **getMethods()**, **getConstructors()**… retournant un tableau représentant respectivement des champs, méthodes et constructeurs. Voir la documentation en ligne de la classe **Class** :

<http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Class.html>.

* + Ainsi, l'information sur la classe d'objets inconnus peut être totalement déterminée dynamiquement, sans rien en savoir à la compilation.

### Exemple

public class ShowMethods {

public static void main(String[] args) { try { Class c = Class.forName(args[0]);

Method[] m = c.getMethods(); Constructor[] ctor = c.getConstructors(); if(args.length **=** 1) {

for (int i = 0; i < m.length; i++) System.out.println(m[i]);

for (int i = 0; i < ctor.length; i++) System.out.println(ctor[i]);

}

else {

for (int i = 0; i < m.length; i++) if(m[i].toString().indexOf(args[1])!= -1)

System.out.println(m[i]); for (int i = 0; i < ctor.length; i++)

if(ctor[i].toString().indexOf(args[1])!= -1) System.out.println(ctor[i]);

}

}

catch(ClassNotFoundException e) {

System.err.println("Classe non trouvée : " + e);

}

}

} 68

### Obtenir des informations sur une classe : introspection

Les méthodes de la classe Class permettent de connaître :

### Les attributs

\*public Field[] getDeclaredFields() les attributs définis dans la classe

\*public Field[] getFields(),les attributs ( y compris ceux qui sont hérités)

### Les constructeurs

\*public Constructor[] getDeclaredConstructors(), les constructeurs définis dans

### La classe

\* public Constructor[] getConstructors(), les constructeurs de la super classe

### Les méthodes

\*public Method [] getDeclaredMethods(), les méthodes définies dans la classes.

\*public Method [] getMethods(), les méthodes .

### Les interfaces implémentées

\*public Class[] getInterfaces(),

### La classe mère

\*public Class getSuperclass(),

### Le paquetage

\*Package getPackage() …

* **Les paramètres de la classe** (pour une classe générique) **:**

\*TypeVariable<?>[] getTypeParameters() retourne un tableau des types paramètres (vide si la classe n'est pas générique). Ce sont les types du code source et non pas les types instanciés à l'exécution.

69

### Étude de cas

Supposons définies les classes suivantes :

class A {

public int aDeA; public void fDeA(){} public A(){

System.out.println(" chargement de A ");

}

}

class B extends A{ private int prive; public int aDeB;

public String toString(){ return "B : "+aDeB;

}

public B(Integer a) {aDeB = a;

public B(){System.out.println(" chargement de B ");}

}

class C {

public C(){System.out.println(" chargement de C ");}

} 70

public static void main(String[] args) { A a = new A(); B b = new B(); try {Cla**s** c = Class.forName("C");} catch(ClassNotFoundException cnfe){

System.out.println(" classe pas trouvée : "+cnfe.getMessage() );}

//Obtenir une référence à l'instance de Class Class cA1 = A.class ; System.out.println(cA1);

Class cA2 = a.getClass(); System.out.println(cA2); Class cB = b.getClass();

System.out.println(" les attributs de : "+ cB.getName()); Field fs[] = cB.getFields();

if( fs.length!=0)

for( int i = 0; i < fs.length; ++i){System.out.println(fs[i]); } else System.out.println(" pas d'attributs");

System.out.println(" les méthodes de : "+ cB.getName()); Method ms[] = cB.getMethods();

if( ms.length!=0)

for( int i = 0; i < ms.length; ++i){ System.out.println(ms[i]); } else System.out.println(" pas de méthode");

System.out.println(" les constructeurs de : "+ cB.getName()); Constructor cs[] = cB.getConstructors();

if ( cs.length!=0)

for( int i = 0; i < cs.length; ++i){System.out.println(cs[i]); } else System.out.println(" pas de constructeur"); System.out.println(" les interfaces de : "+ cB.getName());

Class is[] = cB.getInterfaces(); if( is.length!=0)

for ( int i = 0; i < is.length; ++i){System.out.println(is[i]); } else System.out.println(" pas d'interface");

System.out.println(" nom de la classe mère :"+cB.getSuperclass().getName()); Type ts = cB.getGenericSuperclass();

System.out.println(" nom de la classe mère générique : "+ts.toString()); Type[] tis = cB.getGenericInterfaces();

System.out.println(" nom des interfaces génériques implémentées : ");

for( int i = 0; i < tis.length; ++i) System.out.println(" "+tis[i]);

71

}

chargement de A chargement de A chargement de B

classe pas trouvée : C class reflexion.A

class reflexion.A

les attributs de : reflexion.B public int reflexion.B.aDeB public int reflexion.A.aDeA

les méthodes de : reflexion.B

public java.lang.String reflexion.B.toString() public void reflexion.A.fDeA()

public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException

public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException

public final void java.lang.Object.wait() throws java.lang.InterruptedException

public boolean java.lang.Object.equals(java.lang.Object) public native int java.lang.Object.hashCode()

public final native java.lang.Class java.lang.Object.getClass() public final native void java.lang.Object.notify()

public final native void java.lang.Object.notifyAll() les constructeurs de : reflexion.B

public reflexion.B()

les interfaces de : reflexion.B

72

|  |  |  |  |
| --- | --- | --- | --- |
| pas | d'interface |  | |
| nom | de la classe | mère | : reflexion.A |
| nom | de la classe | mère | générique : class reflexion.A |

nom des interfaces génériques implémentées :