Java avancé #1 (classes anonymes, collections, entrées-sorties et sérialisation, Threads)

DESS TNI de Montpellier

Sept 2003 – version 1.1

Jacques Ferber

LIRMM – Université de Montpellier II

ferber@lirmm.fr

http://www.lirmm.fr/~ferber

Plan

- Classes anonymes
- Collections et itérations
- Introspection
- Entrées-sorties et sérialisation d'objets
- Threads

©Jacques Ferber 2003

Classes anonymes

- Possibilité de créer une classe anonyme et de l'instancier en même temps (analogue au lambda-expressions en Lisp)

```
interface Truc {
    void m();
}
```

```
class Chose {
    int b=80;
    void p(final int a){
            Truc t = new Truc(){
                    int c = a;
                    void m(){
                       super.m();
                       c++;
                       b++;
                    }};
            t.m();
     }
}
```

©Jacques Ferber 2003

Classes anonymes #2

- Permet de créer une classe issue d'une interface
- Attention: les variables locales externes à l'objet doivent être "final" (pas de modifications des variables "fermées"). Mais les attributs ne posent pas de problèmes

```
interface Truc {
    void m();
}
```

```
class Chose {
    int b=80;
    void p(final int a){            // doit être final pour pouvoir être utilisé
            final int r = a+1;      //  idem
            Truc t = new Truc(){
                    int c = a;
                    void m(){
                       c++;  //OK
                       b++; // OK
                       b = b + r; // OK
                    }};
            t.m();
     }
}
```

©Jacques Ferber 2003

Utilisation des classes anonymes

- Utilisation dans des interfaces graphiques pour définir des gestionnaires d'événements individualisés.

```
JButton b1 = new JButton("OK");

        ….

b1.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent e){
        }});
```

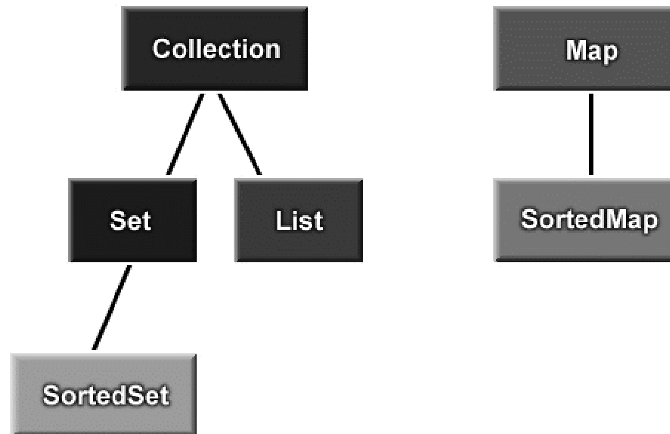
©Jacques Ferber 2003

Les collections

- Une collection est une abstraction pour une boîte contenant des éléments
  - Les éléments peuvent être uniques (Set) ou en plusieurs exemplaires (Collection, List)
  - Les éléments peuvent être ordonnés (Sorted) ou non
  - Les éléments peuvent être accessibles par des clés (Map) ou non
- Ces distinctions sont représentées par des interfaces Java
- Plusieurs implémentations distinctes correspondent à une même interface.

©Jacques Ferber 2003

La hiérarchie des interfaces



Les interfaces (I)

- Fonctionnalités de base d'une collection:

```
public interface Collection {
        // opérations de base
        int size();
        boolean isEmpty();
        boolean contains(Object o);          // appartenance
        boolean add(Object o);
        boolean remove(Object o);
        Iterator iterator();
        // opérations groupées
        boolean containsAll(Collection c);   // inclusion
        boolean addAll(Collection c);        // union
        boolean removeAll(Collection c);     // différence
        boolean retainAll(Collection c);     // intersection
        void clear();
        // tableaux
        Object[] toArray();
}
```

Les interfaces (II)

- Fonctionalités de base d’une collection ordonnée:

```
public interface SortedSet extends Set {
        // extraction
        SortedSet subSet(Object from, Object to);
        SortedSet headset(Object to);
        SortedSet tailSet(Object from);
        // extrémités
        Object first();
        Object last();
        // relation d'ordre
        Comparator comparator();
}
```

©Jacques Ferber 2003

Les interfaces (III)

- Fonctionalités de base d’une liste:

```
public interface List extends Collection {
        // accès indexé
        Object get(int index);
        Object set(int index, Object o);
         void add(int index, Object o);
        boolean addAll(int index, Collection c);
        Object remove(int index);
        // recherche
        int indexOf(Object o);
        int lastIndexOf(Object of);
        // itérateurs
        ListIterator listIterator()
        ListIterator listIterator(int index);
        // extraction
        List subList(int from, int to);
}
```

©Jacques Ferber 2003

Les interfaces (IV)

- Fonctionalités de base d'une association (Map):

```
public interface Map {
        // opérations de base
        Object put(Object key, Object val);
        Object get(Object key);
        Object remove(Object key);
        boolean containsKey(Object key);
        boolean containsValue(Object val);
        int size();
        boolean isEmpty();
        // opérations globales
        void putAll(Map t);
        void clear();
        // extraction
        Set keySet();
        Collection values();
        Set entrySet();
}
```

©Jacques Ferber 2003

Les interfaces (V)

- Fonctionalités d'une association ordonnée:

```
Public interface SortedMap extends Map {
        // comparateur
        Comparator comparator();
        // extraction
        SortedMap subMap(Object fromKey, Object toKey);
        SortedMap headMap(Object toKey);
        SortedMap tailMap(Object fromKey);
        // accès
        Object first();
        Object last();
}
```

©Jacques Ferber 2003

Les implémentations (I)

Techniques:	Table de hashage	Tableau de taille variable	Arbre balancé	Liste chaînée
Set	HashSet		TreeSet	
List		ArrayList		LinkedList
Map	HashMap		TreeMap	

©Jacques Ferber 2003

Utilisation

- Création et ajout

```
List lst = new ArrayList();
lst.add("bonjour");
lst.add(new Compte(1200,2000));
lst.add(new Integer(34));
```

- Récupération

```
String s = (String) lst.first();
Compte c = (Compte) lst.get(1);
int a = ((Integer) lst.last()).intValue();
```

©Jacques Ferber 2003

Deux implémentations particulières

- La notion de vecteur (Vector extends AbstractList)
  - List plus une ancienne interface
  - Attention: Vector est TRES employé encore…
- La notion de pile (Stack extends Vector):
  - boolean empty()
  - Object peek()
  - Object pop()
  - Object push(Object o)
  - int search(Object o)

©Jacques Ferber 2003

Parcourir des collections à l'aide d'itérateurs

- Sert à fournir un moyen de parcourir une collection indépendamment de sa nature propre…
- Il devient alors possible d'écrire des algorithmes génériques qui ne reposent pas sur la structure de donnée utilisée

©Jacques Ferber 2003

Iterator

- L'interface

```
public interface Iterator {
            boolean hasNext();
            Object next();
            void remove(); // Optional
}
```

- Récupération d'iterateurs

```
// envoyé à un type de collection
iterator()
```

©Jacques Ferber 2003

Exemple d'utilisation

```
List lst = new ArrayList();
..

for (Iterator it = lst.iterator(); it.hasNext(); ) {
            System.out.println(it.next());
}
```

©Jacques Ferber 2003

Définition d'ordre dans les objets

- Permet de donner un ordre naturel entre des instances "comparables" d'une classe
  - Pour que deux objets soient comparables, il faut et il suffit qu'ils soient des instances d'une classe implémentant l'interface "comparable"

```
public interface Comparable {
        public int compareTo(Object o);
}
```

<0 si this<o

0 si égal

>0 si this>o

©Jacques Ferber 2003

Exemple de comparaison

```
String s1 = "bonjour";
String s2 = "bonsoir";

s1.compareTo(s2) →-9 ("bonjour" est avant "bonsoir")
```

©Jacques Ferber 2003

Ordre créé

- Un comparateur est une classe qui implémente un ordre
- Utilisé surtout pour faire des collections ordonnées (cf. plus loin)

```
public interface Comparator {
        int compare(Object o1, Object o2);
}
```

©Jacques Ferber 2003

Exemple d'ordre spécifique

```
Comparator comp = new Comparator(){
    int compare(Object o1, Object o2) {
        String s1 = (String) o1;
        String s2 = (String) o2;
        if (s1.length() < s2.length())
            return(-1);
        else …
    }};
```

```
int r = comp.compare("bonjour","bonsoir");
System.out.println(r);        → 0
```

©Jacques Ferber 2003

Les algorithmes

- Les algorithmes sont définis comme des méthodes statiques de la classe Collections
- Les algorithmes suivants sont définis:
  – Le tri avec ou sans comparateur: sort
  – Le mélange: shuffle
  – Les opérations de routine: reverse, fill et copy
  – La recherche: binarySearch, min et max
- Sauf min et max, tous les algorithmes travaillent sur des listes

©Jacques Ferber 2003

Le tri(I)

- Exemple:

```
public class Etudiant implements Comparable {
        String nom, prenom;
        int age;
        long numero;
        public int compareTo(Object e) {
                Etudiant p = (Etudiant) e;
                if (p.numero == numero) return 0;
                else if ( numero <p.numero) return –1;
                else return 1;
        }
        …
}
```

©Jacques Ferber 2003

Le tri(II)

- Exemple (suite):

```
{
        List l = new ArrayList();
        // ajoute des personnes
        …
         // Etudiants triés selon leur numero
        Collections.sort(l);

         // Etudiants triés selon leur  nom
        Collections.sort(l, new Comparator() {
                int compare(Object ob1, Object ob2) {
                        Etudiant o1=(Etudiant) ob1;
                        Etudiant o2 = (Etudiant) ob2;
                        if (o1.getNom().equals(o2.getNom()))
                                return 0;
                        …
                }
        });
        …
}
```


Le mélange

- Permet de mélanger des éléments avec une fonction aléatoire
- Utilisable dans les jeux de hasard (~mélange des cartes) ou les tests
- Exemple:

```
{
        List l = new ArrayList(52);
        // création des cartes
        ….
        // mélange
        Collections.shuffle(l);
        …
}
```

La recherche

- Exemple:

```
{
		List l = new ArrayList();
		// insère les personnes
		…
		// la liste doit être triée!!!
		Collections.sort(l);
		// recherche d'un élément
		Collections.binarySearch(l,unePersonne)
		// recherche par nom
		Collections.binarySearch(l, new comparator() {
			// comment avant
		});
}
```

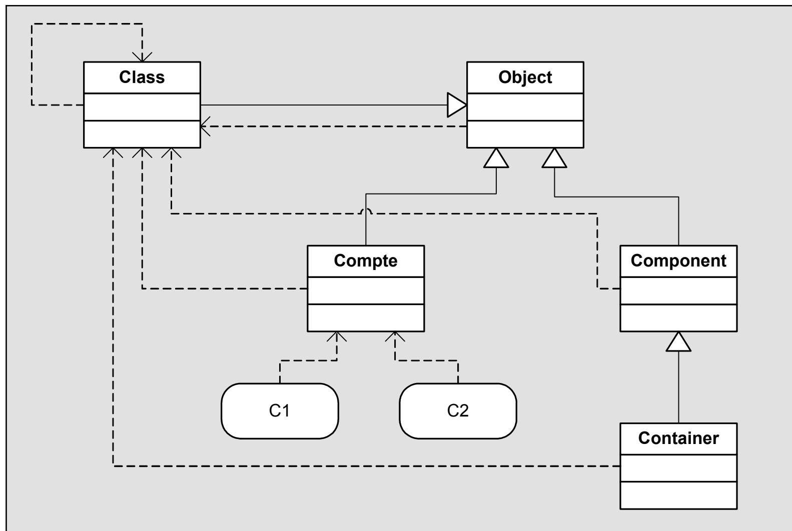
©Jacques Ferber 2003

Introspection

- On appelle "réflexif" un système capable de se représenter lui-même.
  – Pour les langages informatiques, la réflexivité s'exprime comme la capacité pour un langage de décrire les aspects considérés comme implicite dans le langage.
- Dans le cas d'un langage objet:
  – Accès à la classe d'un objet: les classes comme des objets
  – Instanciation de classes dont le nom est connu seulement à l'exécution
  – Accès aux attributs et aux méthodes des objets
  – Invocation de méthode (envoi de message) dont le nom n'est connu qu'à l'exécution.
- En Java la réflexivité s'appelle "introspection"

©Jacques Ferber 2003

La classe comme objet



La classe comme objet #2

- Tout objet peut accéder à sa classe par la méthode: getClass() qui retourne un objet instance de Class.
- On peut accéder à la superclasse d'une classe par la méthode getSuperclass()

```
Truc o = new Truc(10,"Machin");
Class c = o.getClass();
Class d = c.getSuperclass();
System.out.println(o +  " est instance de " +           c.getName()+ " qui herite de " +
d.getName());
```

Récupérer une classe

- On peut aussi manipuler directement une classe dont on connaît le nom complet: Class c = javax.swing.Jbutton;
- Si le nom est inconnu lors de la compilation on peut retrouver une classe en utilisant la méthode statique "forName":

```
String s = "javax.swing.Jbutton";
Try {
   Class c = Class.forName(s);              // retourne la classe de nom s
   System.out.println("la classe est : " + c);
} catch (ClassNotFoundException e) {  // si elle existe
              // Exception a recupérer..
}
```

©Jacques Ferber 2003

Accès aux attributs

- la classe **Field** représente un attribut d'une classe.
- Pour obtenir les champs d'une classe c:
  - **Field[] c.getFields()** // retourne tous les champs publiques d'une classe
  - **Field[] c.getDeclaredFields()** // retourne tous les champs déclarés d'une classe (champs publiques de la classe + champs des superclasses)
  - **Field c.getField(String s)** // retourne le champ de nom **s** de la classe.
  - **Field c.getDeclaredField(String s)** // retourne le champ déclaré **s** d'une classe
- *Exceptions levées*: **NoSuchFieldException, SecurityException**

©Jacques Ferber 2003

Attributs #1

- Accès aux informations de l'attribut:
  - **String getName()**
  - **Class getType()**
  - **int getModifier()**
  - **Class getDeclaringClass()**
- Lecture de la valeur d'un champ par introspection
  - **Object <Field>.get(Object o)**qui retourne la valeur d'un champ sous la manière d'un objet.
    - *Exceptions levées*: **IllegalArgumentException**, **IllegalAccessException**
- Pour les champs contenant des données primitives dont on connaît le type on peut utiliser des expressions plus précises:
  - **int <Field>.getInt(Object o)**

Attributs #2

- Modifications des champs
  - **<Field>.set(Object o, Object value)** qui affecte une valeur à un champ d'un objet.
    - Note: il faut passer l'objet en premier argument du set car la récupération d'un Field se fait dans une classe qui ne connaît pas l'instance o dont on veut modifier l'état.
    - *Exceptions levées*: **IllegalArgumentException**, **IllegalAccessException**
- On peut aussi modifier les champs contenant des éléments statiques directement sans utiliser des Wrappers:
  - **<Field>.setInt(Object o, int value)**
  - **<Field>.setBoolean(Object o, int value)**

Accès aux méthodes

- **Method[] <Class>.getMethods()**
- **Method[] <Class>.getDeclaredMethods()**
- **Method <Class>.getMethod(String name, Class[] parTypes)**
- **Method <Class>.getDeclaredMethod(String name, Class[] parTypes)**

Méthodes

- Accès aux informations de la méthode
  - **String getName()**
  - **Class getReturnType()**
  - **int getModifier()**
  - **Class[] getParameterTypes()**
- Invocation de méthodes:
  - On utilise la méthode : **Object invoke(String nom, Object[] args** définie dans la classe **Method,** après avoir récupérer la méthode dans la classe.
    - Exceptions: **InvocationTargetException**.

Les entrées-sorties et la sérialisation d'objets

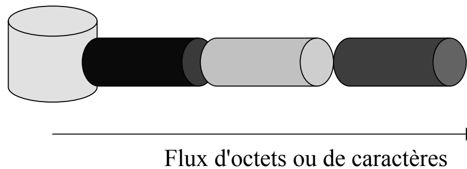
- Introduction
- Les flux de base
- Les tuyaux de plus haut niveau
- La sérialisation d'objets
- Conclusion

Introduction

- Les entrées/sorties sont décrites sous la forme générique de flux (stream):
  - Accès séquentielle à des objets en lecture ou en écriture
  - Indépendant du type d'entrées-sorties: fichiers, buffers, pipes, sockets, etc.
- Deux catégories de flux:
  - Les flux par octets: codage direct des données byte à byte
  - Les flux caractères: formats textuels (un caractère = 2 bytes!,, unicode oblige)

Principe

- Ecrire/lire c'est faire transiter des flux d'octets ou de caractères dans des tuyaux
- 2 tailles de tuyaux: les tuyaux d'1 octet, les tuyaux d'1 caractère
  - Octet: finissent en outputStream ou inputStream (ex: FileInputStream)
  - Caractères: finissent en Reader ou Writer (ex: FileReader, BufferedWriter)



Flux d'octets ou de caractères

Types de flux

- Deux sens de fonctionnement
  - En lecture ("aspiration"): xxxInputStream, xxxReader
  - En écriture ("envoi"): xxxOutputStream, xxxWriter
- Différents types de tuyaux:
  - Les tuyaux qui se trouvent près des éléments d'entrée/sortie (File, Socket, Pipe, String, etc.)
  - Les tuyaux de haut niveau qui implémentent des services (BufferedReader, FilterOutputStream, ObjectInputStream, DataOutputStream...)
  - Les tuyaux de conversion: InputStreamReader (o -> c), OutputStreamWriter (c -> o)…`

Utilisation exemple

- Copie de fichier:
  - Pas très efficace, car trop d'accès au device.

```
public class Copy {
    public static void main(String[] args) throws
IOException {
        File inputFile = new File("fichier entree.txt");
        File outputFile = new File("fichier sortie.txt");

         FileReader in = new FileReader(inputFile);
         FileWriter out = new FileWriter(outputFile);
         int c;

         while ((c = in.read()) != -1)
            out.write(c);

         in.close();
         out.close();
    }
}
```


Exemple (2)

- Amélioration en utilisant un BufferedReader

```
public class Copy {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("fichier entree.txt");
        File outputFile = new File("fichier sortie.txt");

         FileReader in = new BufferedReader(new
FileReader(inputFile));
         FileWriter out = new FileWriter(outputFile);
         int c;

         while ((c = in.read()) != -1)
            out.write(c);

         in.close();
         out.close();
    }
}
```

Flux d'octet: vision abstraite

- Classes abstraites pour les flux octets:

```
public interface InputStream {
        public int read();
        public int read(byte[] buff);
        public int read(byte[] buff,int off,int len);
        public void close();
}
public interface OutputStream {
        public void write();
        public void write(byte[] buff);
        public void write(byte[] buff,int off,int len);
        public void flush();
        public void close();
}
```

- Problème: trop bas niveau pour sauver des données en octets

Les flux de caractères

- Classes abstraites pour les flux caractères:

```
public interface Reader {
        public int read();
        public int read(char[] buff);
        public int read(char[] buff,int off,int len);
        public void close();
}
public interface Writer {
        public void write();
        public void write(char[] buff);
        public void write(char[] buff,int off,int len);
        public void write(String str);
        public void write(String str, int off, int len);
        public void flush();
        public void close();
}
```

Les entrées-sorties

- Introduction
- Les flux de base
- Les tuyaux de plus haut niveau
- La sérialisation d'objets

©Jacques Ferber 2003

Les flux de bases (I)

- Java fournit en standard un ensemble de flux:
  - De et vers les fichiers
  - De et vers les tableaux unidimensionnels
  - De et vers les chaînes de caractères
  - De et vers les "threads" à travers des "pipes"

©Jacques Ferber 2003

Les flux de base (II)

- De et vers les fichiers:
  - En octet:
    - En lecture:
      - FileInputStream(String|File|FileDescriptor)
    - En écriture:
      - FileOutputStream(String|File|FileDescriptor)
  - En caractères:
    - En lecture:
      - FileReader(String|File|FileDescriptor)
    - En écriture:
      - FileWriter(String|File|FileDescriptor)

©Jacques Ferber 2003

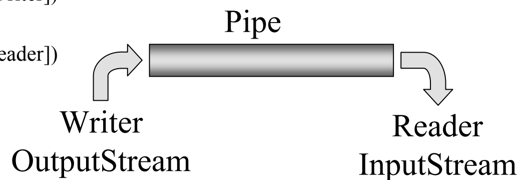
Les flux de base (III)

- De et vers les tableaux unidimensionnels:
  - En octet:
    - En lecture:
      - ByteArrayInputStream(byte[] buff,int off,int len)
    - En écriture:
      - ByteOutputStream()
      - public byte[] toByteArray();
  - En caractères:
    - En lecture:
      - CharArrayReader(char[] buff,int off,int len)
    - En écriture:
      - CharArrayWriter()
      - Public char() toCharArray();

©Jacques Ferber 2003

Les flux de base (IV)

- De et vers les processus:
  - En octet:
    - En lecture:
      - PipedInputStream([PipedOutputStream])
    - En écriture:
      - PipedOutputStream([PipedInputStream])
  - En caractères:
    - En lecture:
      - PipedReader([PipedWriter])
    - En écriture:
      - PipedWriter([PipedReader])
  - Connexion a posteriori:
    - pipe1.connect(pipe2);



Les flux de base (V)

- De et vers les chaînes de caractères:
  - En octet:
    - En lecture:
      - StringReader(String)
    - En écriture:
      - StringWriter()
      - public String toString();

Les entrées-sorties

- Introduction
- Les flux de base
- Les tuyaux de plus haut niveau
- La sérialisation d'objets

©Jacques Ferber 2003

Les tuyaux de plus haut niveau (1)

- Les tuyaux de plus haut niveau se connectent à:
  - Des tuyaux de base
  - D'autres tuyaux de plus haut niveau
- Pour leur donner des interfaces ou fonctionnalités de plus haut niveau:
  - Interfaces déterminées
  - Bufferisation
  - Filtrage
  - Compression
  - Etc.

©Jacques Ferber 2003

Les tuyaux de plus haut niveau (2)

- En entrée octet:
  - FilterInputStream (classe abstraite)
    - BufferedInputStream
    - CheckedInputStream (checksum)
    - DataInputStream
    - LineNumberInputStream
    - PushbackInputStream
  - Transformation de bytes en caractères:
    - InputStreamReader

Les tuyaux de plus haut niveau (3)

- En sortie octet:
  - FilterOutputStream (classe abstraite)
    - BufferedOutputStream
    - CheckedOutputStream
    - DataOutputStream
  - Transformation de caractères en bytes:
    - OutputStreamWriter
  - Pour l'impression simplifiée:

```
Public class PrintWriter {
        public PrintWriter(InputStream is);
        public void print(String str);
        public void println(String str);
}
```

Les tuyaux de plus haut niveau (4)

- En entrée caractères:
  - BufferedReader
  - FilterReader (classe abstraite)
    - PushbackReader
- En sortie caractères:
  - BufferedWriter
  - FilterWriter (classe abstraite)

Les tuyaux de plus haut niveau (5)

- Compression/décompression:
  - Deux sous-classes de java.util.zip.InflaterInputStream (extends FilterInputStream):
    - java.util.zip.GZIPInputStream
    - java.util.zip.ZIPInputStream
  - Deux sous-classes de java.util.zip.DeflaterOutputStream (extends FilterOutputStream):
    - Java.util.zip.GZIPOutputStream
    - Java.util.zip.ZIPOutputStream

Les tuyaux de plus haut niveau (6)

- Exemple:
  - Sauvetage de données compressées:

```
try {
        FileOutputStream fos = new FileOutputStream("…");
        GZIPOutputStream gos = new GZIPOutputStream(fos);
        DataOutputStream dos = new DataOutputStream(gos);
        dos.writeBoolean(b);
        dos.writeLong(l);
        …
        dos.close();
}
catch (IOException e1) { … }
catch (FileNotFoundException e2) { … }
```


Les tuyaux de haut niveau (7)

- Classes de haut niveau pour lire/sauvegarder des données primaires:

```
class DataInputStream
        extends FilterInpuStream{
    public boolean readBoolean();
    public byte readByte();
    public int readUnsignedByte();
    public short readShort();
    public int readUnsignedShort();
    public int readInt();
    public long readLong();
    public float readFloat();
    public double readDouble();
    public char readChar();
    public String readLine();
}
```

```
class DataOutputStream
        extends FilterOutputStream{
    public void writeBoolean(boolean);
    public void writeByte(byte);
    public void writeBytes(String);
    public void writeShort(short);
    public void writeInt(int);
    public void writeLong(long);
    public void writeFloat(float);
    public void writeDouble(double);
    public void writeChar(int);
    public void writeChars(String);
}
```

Exemple avec des DataOutputStream

```
DataOutputStream out = new DataOutputStream(new
                              FileOutputStream("facture.txt"));

double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
int[] units = { 12, 8, 13, 29, 50 };
String[] descs = { "Java T-shirt",
                      "Java Mug",
                      "Duke Juggling Dolls",
                      "Java Pin",
                      "Java Key Chain" };

for (int i = 0; i < prices.length; i ++) {
   out.writeDouble(prices[i]);
   out.writeChar('\t');
   out.writeInt(units[i]);
   out.writeChar('\t');
   out.writeChars(descs[i]);
   out.writeChar('\n');
}
out.close();
```


Exemple avec des DataInputStream

```
DataInputStream in = new DataInputStream(new
                              FileInputStream("facteure.txt"));
double price;
int unit;
StringBuffer desc;
double total = 0.0;

try {
    while (true) {
        price = in.readDouble();
        in.readChar();       // throws out the tab
        unit = in.readInt();
        in.readChar();       // throws out the tab
       char chr;
       desc = new StringBuffer(20);
       char lineSep = System.getProperty("line.separator").charAt(0);
       while ((chr = in.readChar()) != lineSep)
              desc.append(chr);
           total = total + unit * price;
        }
} catch (EOFException e) {}
  in.close();
```

Les entrées-sorties

- Introduction
- Les flux de base
- Les tuyaux de plus haut niveau
- La sérialisation d'objets

La sérialisation #1

- Les objets peuvent être transformés et "sérialisés" sous la forme d'une suite d'octets.
- Cette suite d'octets peut ensuite être utilisée pour reconstruire l'objet.
- Cette fonctionnalité est surtout utilisée pour:
  - sauvegarder des objets sur disque ou dans des "conteneurs d'objets" (persistance, stockage d'objets, EJB)
  - transférer des objets par le réseau pour la réalisation d'applications distribuées (RMI, etc..)

Exemples de base de sérialisation #2

- Ecriture d'objets dans un flux
  - *Attention*: tous les objets liés sont sauvés aussi !! (sauf pour les attributs marqués transient)

```
FileOutputStream out = new FileOutputStream("fichierTemps");
ObjectOutputStream sout = new ObjectOutputStream(out);
String aujourd = "Aujourd'hui";
sout.writeObject(aujourd);
sout.writeObject(new Date()); s.flush();
```

- Lecture d'objet à partir d'un flux

```
FileInputStream in = new FileInputStream("fichierTemps");
ObjectInputStream sin = new ObjectInputStream(in);
String today = (String)sin.readObject();
Date date = (Date)sin.readObject();
```


La sérialisation #2

- Pour être sérialisé, un objet doit implémenter l'interface Serializable.
  - L'interface Serializable ne contient aucune méthode, elle est utilisée uniquement comme marqueur de sérialisation.
- Ce sont les ObjectOutputStream et ObjectInputStream qui se chargent du travail de sérialisation et désérialisation.

Adapter le processus de sérialisation

- On peut adapter les mécanismes de sérialisation en redéfinissant des méthodes (privées!!) qui gèrent la transformation objet↔octet.

```
private void writeObject(ObjectOutputStream s) throws IOException {
		s.defaultWriteObject();
		// code spécifique de sérialisation à ajouter ici
 }
```

```
private void readObject(ObjectInputStream s) throws IOException, ClassNotFoundException {
		s.defaultReadObject();
		// code spécifique de désérialisation à ajouter ici
		// code pour mettre à jour l'objet ou le réinitialiser si nécessaire
}
```

©Jacques Ferber 2003

Threads et programmation concurrente en Java

- Introduction
- Threads
- Synchronisation
- Gestion des threads
- Lecteur/rédacteur
- Conclusion

©Jacques Ferber 2003

Introduction

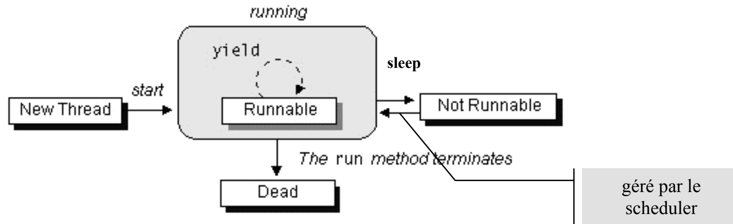
- Dès que l'on veut que plusieurs unités de programmes travaillent en parallèle, il faut gérer la concurrence
- La concurrence repose sur:
  - La possibilité de définir des processus concurrents:
    - en Java la technique des threads: processus léger
  - Des mécanismes de communication entre ces processus
    - Communication synchrone par appel de méthode
    - Communication asynchrone par envoi de message
  - Des mécanismes de synchronisation de ces processus

Les threads

- Threads ou processus léger
  - Permet de définir des processus de programmes qui travaillent en concurrence à l'intérieur d'une JVM (pseudo-parallélisme)
  - Gestion "relativement" portable de ces threads sur différents systèmes
  - Comportent des mécanismes de synchronisation fondés sur la technique des moniteurs de Hoare
  - Possibilité de gérer l'ordonnancement de ces threads

Cycle de vie du thread

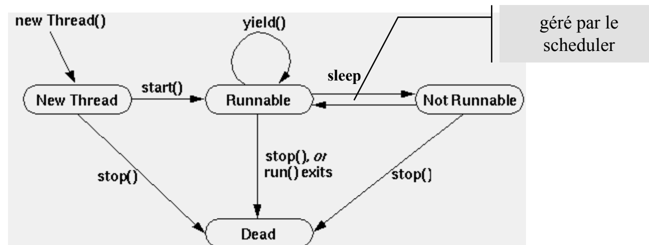
- La vision actuelle du cycle de vie d'un thread



©Jacques Ferber 2003

Cycle de vie du thread

- La vision ancienne (mais toujours valable) du cycle de vie du thread



©Jacques Ferber 2003

Action sur un thread

- **start()**: démarre l'exécution d'un thread
- **yield()**: redonne explicitement le contrôle au scheduler.
- **sleep(long)**: fait passer le thread en état d'attente pendant une certaine durée (comptée en millisecondes).
- **run()**: la méthode qui est exécutée lorsque le thread s'exécute. C'est souvent une boucle.
- **stop()**: termine son exécution (il passe dans l'état "mort") deprecated
- **suspend()**: suspend l'exécution (doit être relancé par un "resume" ensuite. Fait passer le thread dans un état d'attente (not runnable). deprecated
- **resume()**: relance et "résume" l'exécution. deprecated

©Jacques Ferber 2003

Le scheduler (l'ordonnanceur)

- Programme qui gère l'exécution (l'ordonnancement) des tâches les unes après les autres:

```
Tant que (vrai){
        Prend le premier thread avec la priorité la plus forte
                qui se trouve "runnable" → t
        Execute t pendant un certain temps
        replace t en "runnable"
}
```

©Jacques Ferber 2003

Création d'un thread (I)

- Deux méthodes:
  - Création d'une instance d'une sous-classe de Thread
  - Création d'une instance de Thread avec une implémentation de Runnable

©Jacques Ferber 2003

Première méthode

- Définit une classe sous-classe de Thread

```
class SimpleThread extends thread {
        SimpleThread(String name) {
                super(name)
        }

        public void run() {
                for(int=0; i < 10; i++)
                     System.out.println(getName()+i);
        }
}

class EssaiThread {
        public static void main(String argv[]) {
                SimpleThread essaiA = new SimpleThread("c
                SimpleThread essaiB = new SimpleThread("Thread B:");
                SimpleThread essaiC = new SimpleThread("Thread C:");

                essaiA.start();
                essaiB.start();
                essaiC.start();
        }
}
```

©Jacques Ferber 2003

Résultat

- Produit normalement

```
Thread A:1
Thread B:1
Thread C:1
Thread A:2
Thread B:2
Thread C:2
Thread A:3
Thread B:3
Thread C:3
```

- Produit parfois

```
Thread A:1
Thread A:2
Thread A:3
Thread B:1
Thread B:2
Thread B:3
Thread C:1
Thread C:2
Thread C:3
```

©Jacques Ferber 2003

Pour éviter le problème

- Faire des 'sleep' (préféré) ou des 'Yield'
- Attention: un processus continu (while (true) {…}) consomme beaucoup de ressources processeurs!!

```
class SimpleThread extends thread {
        essaiThread(String name) {
                super(name)
        }

        public void run() {
                for(int=0; i < 10; i++){
                      System.out.println(getName()+i);
                      sleep(200);
                }
        }
}
```

©Jacques Ferber 2003

Deuxième méthode

- Faire une implémentation de runnable et passer cette implémentation au thread lors de sa création

```
class RunningThread implements Runnable {

        public void run() {
                for(int=0; i < 10; i++){
                     System.out.println(getName()+i);
                     sleep(200);
                }
        }
}

RunningThread r = new RunningThread();
Thread t = new Thread(r);
r.start();
```

©Jacques Ferber 2003

Priorité

- On peut aussi changer la priorité du thread avec la méthode **setPriority(int)**.
  - Le thread avec la plus grande valeur (le plus prioritaire) est activé préférentiellement.

©Jacques Ferber 2003

Exclusion mutuelle (I)

- Problème:
  - L'accès simultané en modification sur un objet peut laisser l'objet dans un état incohérent

```
public class MyThread extends Thread {
    Compte c;
    MyThread(Compte aCompte){
      c = aCompte;
    }

    public void run(){
          c.deposer(100);
          c.retirer(50);
    }
}
```

```
public class Compte
        int solde = 0;
        public void deposer(int s) {
                int so = solde + s;
                solde = so;
        }
        public void retirer(int s) {
                int so = solde - s;
                solde = so;
        }
}
```

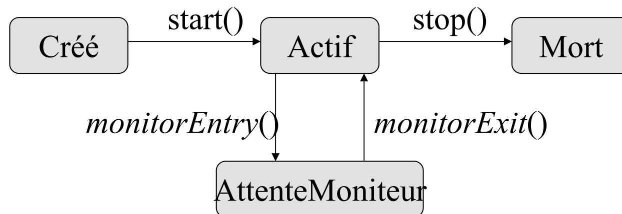
```
Compte c1 = new Compte();
MyThread t1, t2;
t1 = new MyThread(c1);
t2 = new MyThread(c1);

t1.start();
t2.start();
```


Exclusion mutuelle (II)

- Notion de moniteur:
  - Un moniteur (inventé par Hoare) est un ensemble de variables et de méthodes qui ne peuvent être accédées que par un thread à la fois
- Implémentation:
  - Un moniteur peut être libre ou occupé
  - Avant d'entrer, un thread demande l'autorisation (monitorentry):
    - Si le moniteur est libre, l'autorisation est donnée et il devient occupé
    - Si le moniteur est occupé, le thread est mis dans une file d'attente
  - Avant de sortir (monitorexit):
    - Le moniteur devient libre
    - Le thread suivant est réactivé

Exclusion mutuelle (III)

- Schéma:



- En java, tout objet est un moniteur utilisable par:
  - La déclaration synchronized
  - L'instruction synchronized

Exclusion mutuelle (IV)

- La déclaration synchronized:
  - La déclaration:

```
public synchronized void toto(…) {
        …
}
```

  - est équivalente à:

```
public void toto(…) {
        monitorEntry();
        …
        monitorExit();
}
```

- Exemple:

```
public class Compte
   int solde = 0;

  public synchronized void deposer(int s) {
        int so = solde + s;
        solde = so;
  }
  public synchronized void retirer(int s) {
        int so = solde - s;
        solde = so;
  }
}
```

Exclusion mutuelle (V)

- L'instruction synchronized:
  - L'instruction synchronized:
    - synchronized(obj){…}
  - est équivalente à:

    object.monitorEntry()

    …

    object.monitorExit()
- Mais rien n'empêche un processus d'accéder de manière non synchronisée à obj…
  - Cette technique est donc à éviter autant que possible…

```
public class MyThread extends Thread {
      Compte c;
      MyThread(Compte aCompte){
           c = aCompte;
       }

       public void run(){
              try {
                  synchronized(c){
                        c.deposer(100);
                        c.retirer(50);
                  }
       }
}
// et il n'est plus nécessaire de
synchroniser les méthodes de Compte
```

©Jacques Ferber 2003

Exclusion mutuelle (VI)

- Utilisation:
  - Instruction synchronized:
    - Quand on utilise une instance d'une classe qui n'est pas "Thread-safe"
    - Si on veut rendre une transaction atomique
  - Méthodes synchronized:
    - Pour rendre une classe "Thread-safe"
    - Rend l'occupation du moniteur plus ponctuelle
- Remarque:
  - La deuxième méthode est en général préférable à la première

©Jacques Ferber 2003

Synchronisation par Attente/Notification (I)

- Methodes
  - wait() ou wait(long timeout)
    - suspend le threads courant, et attend une notification sur le moniteur associé à o.
  - notify(), notifyAll()
    - réveille le (tous les) threads en attente sur le moniteur associé
- Utilisation
  - Ces méthodes ne peuvent être utilisées que dans un bloc synchronized (instruction ou méthode)

©Jacques Ferber 2003

Lecteurs-rédacteurs (I)

- N threads accèdent en lecteur ou en écriture à une structure de données partagée
- Problèmes:
  - Il ne faut pas accéder en écriture simultanément
  - Il ne faut pas accéder en lecture si il y a un accès en écriture
  - Les lecteurs ne doivent pas empêcher les rédacteurs d'accèder (famine)

©Jacques Ferber 2003

Lecteurs-rédacteurs (II)

- Protocole:
  - Tout lecteur demande l'autorisation:
    - Accepté si il n'y a pas de rédacteur
  - Tout lecteur annonce sa sortie
  - Tout rédacteur demande l'autorisation:
    - Accepté si il n'y a pas d'autres rédacteurs
  - Tout rédacteur accepté attend qu'il n'y ait pas plus de lecteurs avant d'écrire
  - Tout rédacteur annonce sa sortie

©Jacques Ferber 2003

Lecteurs-rédacteurs (III)

- Implémentation du protocole:

```
public class LecteurRedacteur {
        private int nbLecteurs=0;
        private boolean redacteur=false;
        private synchronized void startRead() {
                while (redacteur) { wait(); } ++nbLecteurs;
        }
        private synchronized void stopRead() { --nbLecteurs; notifyAll();}
        private synchronized void startWrite() {
                while (redacteur) { wait(); } redacteur=true;
        }
        private synchronized void waitNoReader() {
                while (nbLecteurs!=0) { wait(); }
        }
        private synchronized void stopWrite() { redacteur=false; notifyAll();}
}
```

©Jacques Ferber 2003

Lecteurs-rédacteurs (IV)

- Utilisation du protocole:

```
public class LecteurRedacteur {
        …
        private Object o;
        private Object read() {
                startRead(); Object o=this.o.clone(); stopRead(); return o;
        }
        private void write(Object o) {
                startWrite(); waitNoReader(); this.o = o.clone(); stopWrite();
        }
}
```

- Remarque: ne jamais compter sur le fait que quelqu'un d'autre va respecter le protocole => encapsulation dans la classe

©Jacques Ferber 2003