Cours Java (INHM)

MASTER 1

Catherine RECANATI
L.I.P.N. (Laboratoire d'Informatique de Paris Nord)
Université de Paris 13

http://www-lipn.univ-paris13.fr/~recanati

Plan du cours

- Chapitre 1 : un survol de Java.
- **Chapitre 2 : compléments sur le langage**.
  (classes abstraites, interfaces, String et StringBuffer, Threads, le package d'entrées/sorties java.io)
- Chapitres 3, 4, 5 et 6 : Java Swing.
  (composants, afficheurs, événements, dessins, images , dialogues et animation).

Chapitre 2 : Compléments

- Les classes abstraites et les interfaces
- Les chaînes String & StringBuffer
- Le paquetage d'entrées/sorties
- Les Threads

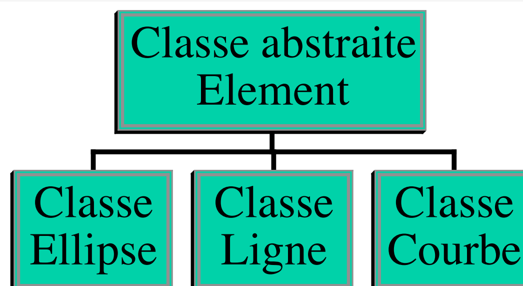
Classes abstraites et Interfaces

Classes abstraites

- Une classe abstraite n'est pas instanciable
- Les classes abstraites permettent de déclarer des classes ne définissant que partiellement leur implémentation, laissant ainsi à des sous classes le soin de la compléter
- Ces classes sont déclarées `abstract` et chaque méthode non implémentée est spécifiquement déclarée `abstract`

➢ *Si toutes les méthodes sont abstraites, on peut parfois préférer définir une interface*


```
/** pour gérer les élements d'une feuille d'un logiciel de dessin */
public abstract class Element {
    protected Color color;    // les éléments sont colorés
    public Element(Color color) {  this.color = color; }
    public Color getColor() {  return color; }

    public abstract Shape getShape();
    public abstract Rectangle getBounds();
    public abstract void modify (Point debut, Point fin);
}
```



```
abstract class Benchmark {
    abstract void benchmark ();

    public long repeat (int count)  {
        long start = System.currentTimeMillis ();
        for (int i = 0; i < count; i++)
                benchmark();
        return (System.currentTimeMillis () - start);
    }
}

class MethodeBenchmark extends Benchmark {
    void benchmark() { } // ici la méthode est vide ce qui va
        // permettre de mesurer le temps d'invocation d'une méthode
    public static void main (String [ ] args)  {
        int count = Integer.parseInt (args[0]); // nb d'invocations
        long time = new MethodeBenchmark().repeat(count);
        System.out.println(count + " methodes en " + time + "ms");
    }
}
```

Interfaces

- Toutes les méthodes d'une interface sont implicitement `public` et `abstract`. (Et elles ne peuvent bien sûr pas être statiques)

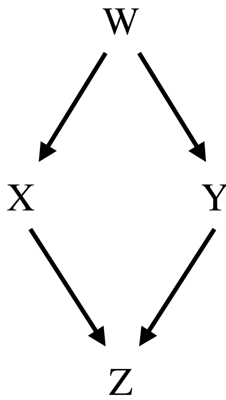
➢ Si une classe n'implémente pas toutes les méthodes d'une interface, elle doit être déclarée `abstract`

- Tous les champs d'une interface sont implicitement `static` et `final`: ils ne peuvent donc être que constants

```
import java.awt.*;

public interface Constants {

// Les types d'éléments composant une feuille de dessin
final static int LIGNE = 101;
final static int RECTANGLE = 102;
final static int CERCLE = 103;
final static int COURBE = 104;

 // Les conditions initiales
int DEFAULT_ELEMENT_TYPE = RECTANGLE;
Color DEFAULT_ELEMENT_COLOR = Color.green;

}
```

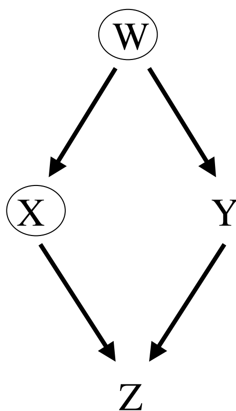
```
// On a introduit dans le chapitre précédent la classe Attribut
// pour gérer des paires nom-valeur
// On a vu comment l'étendre pour avoir un type spécialisé
// d'objet attribué. Mais pour permettre une plus grande utilisation
// de la notion d'Attribut, on peut créer un type Attribué à utiliser
// pour les objets pouvant avoir des attributs fournis en leur
// attachant des objets Attributs

class CorpsCelesteAttribue extends CorpsCeleste
                                        implements Attribue{
        // …
 }

interface Attribue  {
    void          add (Attribut newAtt);
    Attribut      find (String attrName);
    Attribut      remove(String attrName);
    java.util.Enumerations attrs(); // retourne les attributs attachés
}
```

Héritage multiple



On peut implémenter avec les interfaces, un héritage qui est analogue à celui-ci, communément appelé héritage *en diamant*

Héritage multiple




```
interface W { }
interface X extends W { }
class Y implements W { }
class Z extends Y implements X { }
```

Quand utiliser des interfaces ?

- Les interfaces fournissent une forme d'héritage multiple, car une classe peut implémenter plusieurs interfaces (mais elle ne peut étendre qu'une seule classe, même abstraite).
- Par contre une classe `abstract` peut avoir une implémentation partielle, des méthodes `static`, des parties `protected`, alors que les interfaces se limitent aux méthodes `public` et aux constantes.

Les chaînes
`String` et `StringBuffer`

`String:` Opérations de base

La classe String fournit des chaînes qui ne peuvent qu'être lues

```
public String ()
public String (String value)
```

opérations de base: `length` et `charAt`

```
for (int i=0; i < str.length(); i++)
        counts [ str.chartAt (i)]++;
```

String: Opérations de base

Méthode	Retourne l'index de …
indexOf (char ch)	**première position** de ch
indexOf (char ch, int start)	première pos. de ch ≥ start
indexOf (String str)	première position de str
indexOf (String str, int start)	première pos. de str ≥ start
lastIndexOf (char ch)	**dernière position** de ch
lastIndexOf (char ch, int start)	dernière pos. de ch ≥ start
lastIndexOf (String str)	dernière position de str
lastIndexOf(String str, int start)	dernière pos. de str ≥ start

Comparaison de chaînes

- La comparaison des chaînes ne prend pas en compte l'internationalisation des langues et la localité. (Elles sont comparées numériquement sur leur valeur Unicode).
- Opérations de base : `equals`, `equalsIgnoreCase` et `compareTo` pour les trier.
- Mais aussi : `regionMatches`, `startsWith`, `endsWith`

Création de chaînes voisines

public String **concat**(String str)

public String **replace**(char oldChar, char newChar)

public String **toLowerCase** ()

public String **toUpperCase** ()

// la chaîne où les caractères blancs

// en début et en fin ont été supprimés :

public String **trim** ()

Conversion de chaînes

Type	conversion d'une `String` str en …
boolean	new Boolean(str).booleanValue()
int	Integer.parseInt(String str, int base)
long	Long.parseLong(String str, int base)
float	new Float(str).floatValue()
double	new Double(str).doubleValue()

Conversion en chaînes

- `String.valueOf` permet de traduire tous les types de base précédents en String.
- Il n'y a pas de méthode qui convertisse des `char` en `String` et vice-versa. Mais on peut appeler `String.valueOf` avec un caractère, pour obtenir une chaîne qui contient cet unique caractère.
- il n'y a pas non plus de méthodes pour décoder les chaînes numériques des formats du C.
- Une classe qui implante une méthode `toString` peut utiliser `valueOf(Object obj)`.

La classe `StringBuffer`

- La classe `StringBuffer` permet de créer et manipuler des chaines modifiables. On les crée avec

  public StringBuffer ( )

  public StringBuffer (String str)
- et on dispose de méthodes `setCharAt`, `setLength` (qui substitue des caractères nuls), `append` et `insert` pour insérer n'importe quel objet converti en `String` à un emplacement donné.

La classe `StringBuffer`

Un `StringBuffer` grandit automatiquement quand c'est nécessaire, mais il est plus efficace de lui allouer une capacité initiale avec

public StringBuffer (int capacity)

on dispose aussi des méthodes

public synchronized void ensureCapacity(int minimum)

public int capacity ( ) : capacité courante

Le paquetage `java.io` d'entrées/sorties

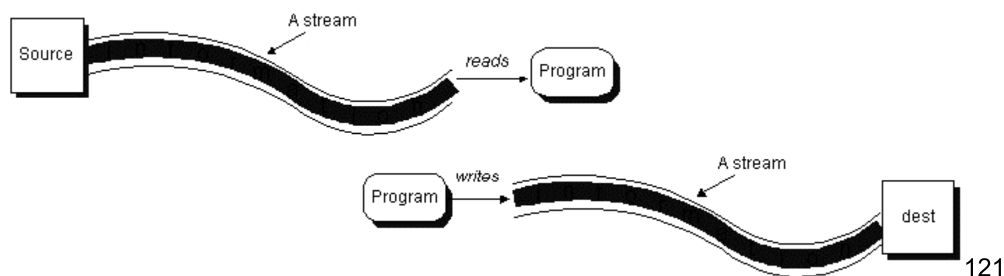
Notion de flux (`Stream`)

Le schéma d'un programme consommateur (ou producteur) d'une source (ou destination) d'information est toujours le suivant:

open *source (or destination)*
while *more information*
read (or write) *information*
close *source (or destination)*

Les flux (ou flots) d'E/S

Un consommateur (d'entrées) n'a pas besoin de connaître la source, et un producteur (de sorties) n'a pas besoin de connaître la destination. Les flots (`Stream`) servent d'intermédiaires.



Les flux (ou flots) d'E/S

- Un flux ou flot est un chemin de communication entre une source et une destination
- Java propose:
  – des flots d'entrées (création, utilisation, détection de fin)
  – des flots de sorties (ibid)

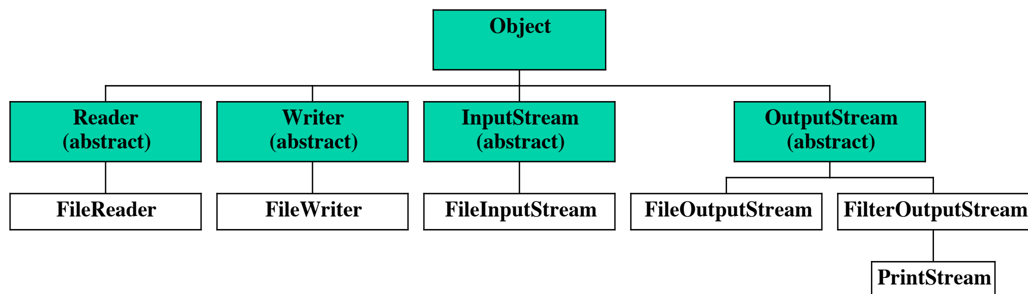
Une vue globale de java.io

- Java 1 propose des classes d'E/S dont l'unité est le caractère construites sur les classes abstraites `Reader` et `Writer`.
- Pour faire un pont entre les caractères et les bytes, leurs sous-classes :
  - `InputStreamReader` (=> `FileReader`)
  - `OutputStreamWriter` (=> `FileWriter`)

Une vue globale de java.io

- Java propose aussi des classes d'E/S dont l'unité est le byte construites sur les classes abstraites `InputStream` et `OutputStream`.
- Leurs sous-classes :
  - `FileInputStreamReader`
  - `FileOutputStreamWriter`

Une vue globale de java.io



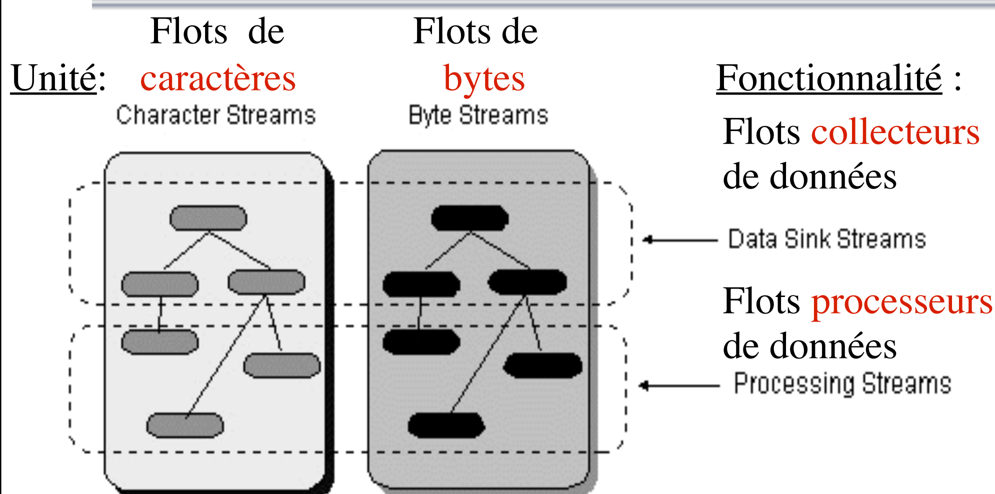
Les trois flux standard

- Les trois flux standard (entrée standard, sortie standard, sortie d'erreur) sont instanciées dans des variables de la classe `System` :
- `System.in` est de type `InputStream`
- `System.out` est de type `PrintStream`
- `System.err` est de type `PrintStream`

Résumé

- Le package java.io définit des classes abstraites pour les flots d'entrées/sorties. Elles sont ensuite étendues pour donner des types de flots utilisables. Ces types vont généralement par paires (flot d'entrées/flot de sorties)
- Ce package contient aussi des classes pour la manipulation des fichiers et un analyseur lexical pour convertir un flot d'entrée en unités lexicales
- Presque toutes les méthodes de ces classes peuvent générer des `IOException`

Organisation des flots



Collecteurs de données

Type de collecteur	Character Streams	Byte Streams
Mémoire	CharArrayReader, CharArrayWriter	ByteArrayInputStream, ByteArrayOutputStream
	StringReader, StringWriter	StringBufferInputStream
Pipe	PipedReader, PipedWriter	PipedInputStream, PipedOutputStream
File	FileReader, FileWriter	FileInputStream, FileOutputStream

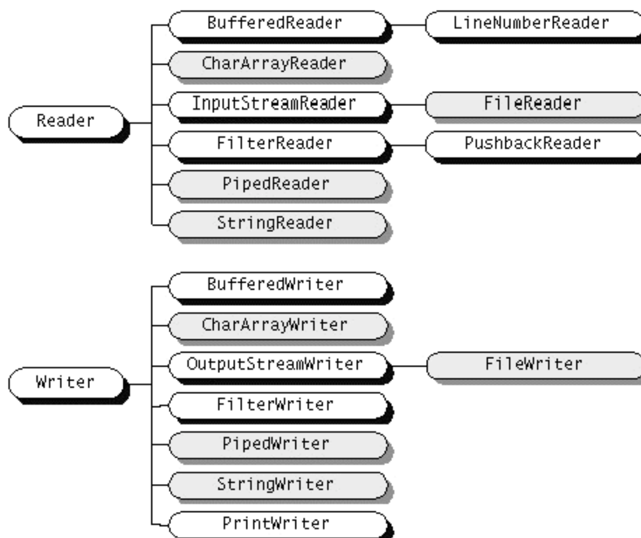
Processeurs de données

Processus	Character Streams	Byte Streams
Serialisation d'objets		ObjectInputStream, ObjectOutputStream
Conversion de données		DataInputStream, DataOutputStream
Comptage	LineNumberReader	LineNumberInputStream
Lire en avance	PushbackReader	PushbackInputStream
Imprimer	PrintWriter	PrintStream

Processeurs de données (suite)

Processus	Character Streams	Byte Streams
Bufferisation	BufferedReader, BufferedWriter	BufferedInputStream, BufferedOutputStream
Filtrage	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream
Conversion entre bytes et character	InputStreamReader, OutputStreamWriter	
Concaténation		SequenceInputStream

Flots d'E/S de caractères



Rappel

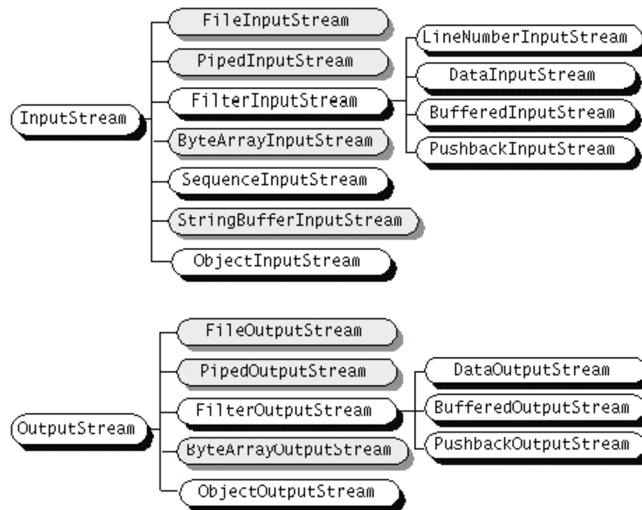
Le schéma d'un programme consommateur ou producteur est toujours le suivant:

```
open a stream
    while more information
        read (or write) information
close the stream
```


Utilisation des classes `FileReader` et `FileWriter`

```
public class Copy {/** copie farrago.txt dans outagain.txt **/
  public static void main(String[] args) throws IOException {
      File inputFile = new File("farrago.txt");
      File outputFile = new File("outagain.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();
  }
}
```

Flots d'E/S de bytes



Classe `InputStream`

C'est une classe abstraite qui donne les moyens de lire un flot d'octets :

```
public abstract int read(); // lit l'octet suivant
public int read(byte [ ] b ); // lit et stocke des octets dans
                               // le tampon b
public int read (byte b [ ], int off, int leng); // lit jusqu'à
    // leng octets de données et les stocke dans le tableau
    // b à partir de la cellule  b [off]
public long skip(long n) ; // saute  n octets
```

Classe `InputStream`

public int available(); // le nombre d'octets disponibles
public void close(); // pour fermer le flux

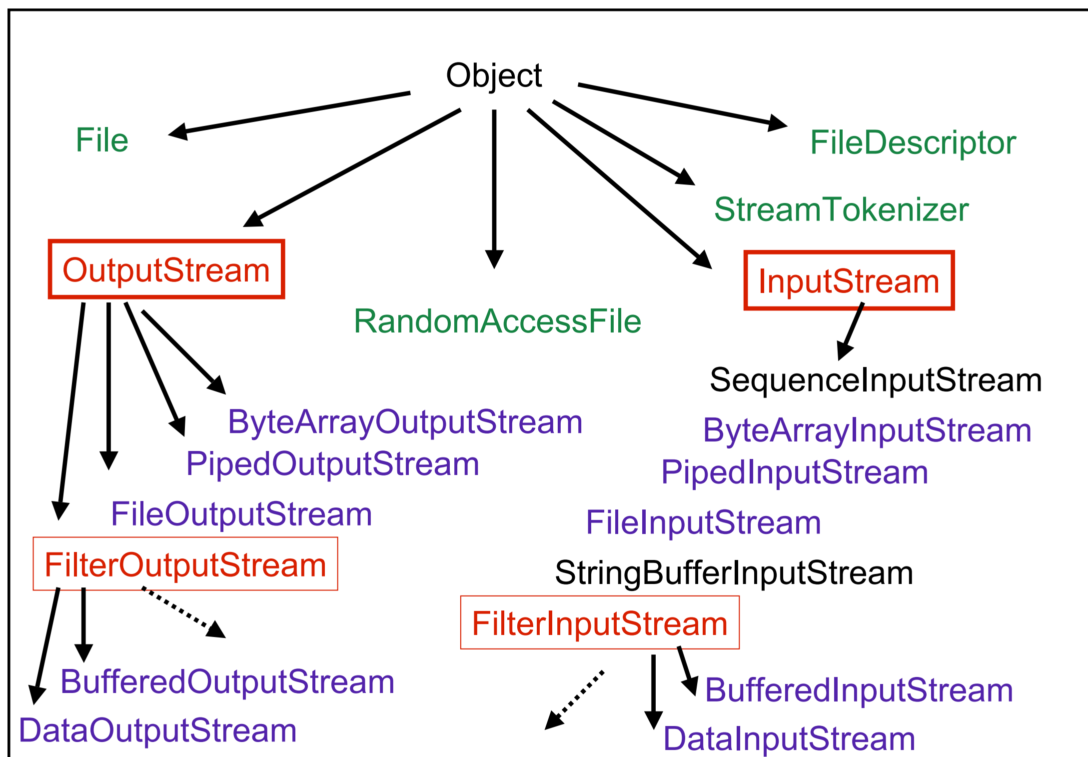
On peut poser une marque sur un flot :

```
boolean  markSupported();  // indique si les méthodes
            // mark et reset sont implantées
public void mark (int readlimit);  // marque la position
            // courante pour y revenir (et relire les mêmes)
public abstract  void reset(); // repositionne sur la
            // marque la plus récente
```


Classe `OutputStream`

C'est une classe abstraite qui donne les moyens à un producteur d'écrire des octets à une destination donnée:

```
public void write (int b );
public void write (byte b [ ]);
public void write (byte b [ ], int off, int len);
     // écrit len octets des données de b
     // à partir de l'offset off dans  b [ ]
public void flush();  // force l'écriture et vide le tampon
public void close (); // ferme le flux
```



Exemples d' `InputStream`

- `System.in` : est une instance d'InputStream

  Pour lire facilement des caractères, des entiers ou des réels sur l'entrée standard (et vérifier que l'entrée standard n'est pas vide), on peut écrire une classe In qui permettra d'initialiser le flot d'entrée avec

  In.init(), et de le tester avec In.isEmpty() on lira ensuite avec getString(), getDouble() et getInt()

```
import java.io.*;
public class In {
   private static int c;
```

```
private static boolean isWhite()
      { return Character.isWhitespace ( (char) c);}
private static void readC () {
      try { c = System.in.read ();} catch (IOException e) {c = -1;}}
public static void init() {readC();}
public static boolean isEmpty() {return c == -1;}
public static String getString () {
      String s = "";
      while (!(isEmpty()) && (isWhite() ))
        readC ();    // manger les blancs
      while (!(isEmpty() | | isWhite() ))
        { s += (char) c; readC (); } // stocker les char non blanc
      return s;
}
public static int getInt() {return Integer.parseInt(getString());}
public static double getDouble ()
      {return Double.parseDouble(getString());}
```

Exemples d' `InputStream`

- `ByteArrayInputStream`: permet de créer un flot à partir d'un tableau d'octets

  public ByteArrayInputStream (byte buff [ ]);

  public ByteArrayInputStream (byte buff [ ], int off, int len);

- `FileInputStream`: créer un flot à partir d'un fichier

  public FileInputStream (String name);

  public FileInputStream (File file);

  public FileInputStream (int fd);

```
import java.awt.*;
import java.io.*;

public class FluxDeFichier {
   public static void main(String[] args) throws IOException {
        FileDialog f = new FileDialog (new Frame(), "ouvrir",
                                                  FileDialog.LOAD );
        f.show ();
        String fichier = f.getFile ();
        String dir = f.getDirectory ();

        InputStream s = new FileInputStream( dir + fichier);
        byte buffer [ ] = new byte [s.available()];
        s.read(buffer);
        for (int i=0; i!=buffer.length; i++)
                System.out.print ( (char) buffer[i]);
        }
}
```

Exemples d' OutputStream

- `ByteArrayOutputStream` : envoie un flot dans un tableau d'octets
  public ByteArrayOutputStream ( );
  public ByteArrayOutputStream (int size);

- `FileOutputStream` : envoie un flot dans un fichier
  public FileOutputStream (String name);
  public FileOutputStream (File file);
  public FileOutputStream (FileDescriptor fdObj);

ByteArrayOutputStream : envoie un flot dans un tableau de bytes

```
…
FileDialog f = new FileDialog (new Frame(), "ouvrir",
                                   FileDialog.LOAD );
f.show ();
String fichier = f.getFile ();
String dir = f.getDirectory ();

InputStream s = new FileInputStream( dir + fichier);
byte buffer [ ] = new byte [s.available()];
s.read(buffer);

ByteArrayOutputStream b = new ByteArrayOutputStream();
b.write(buffer); b.write(buffer);
System.out.println (b.size () ) ;
byte result [ ] = b.toByteArray (); // contient 2 fois le fichier
System.out.println (result.length);
```

FileOutputStream : envoie un tableau de bytes dans un fichier

```
…

File fichier = new File ("resultat");     // dans le répertoire
                                          // courant
FileOutputStream   f = new FileOutputStream (fichier);
f.write(result);      // result est un tableau de bytes
f.close();

System.out.println (fichier.length);      // affiche la taille
                                          // du fichier
```

FilterInputStream FilterOutputStream

Ces deux classes donnent accès aux méthodes des classes **InputStream** et **OutputStream** à travers 7 sous-classes bien particulières :

- **BufferedInputStream** / **BufferedOutputStream** permettent de lire et écrire à travers un tampon pour améliorer les performances

  public BufferedInputStream (InputStream in);

  public BufferedInputStream (InputStream in, int size);

  public BufferedOutputStream (OutputStream out);

  public BufferedOutputStream (OutputStream out, int size);

FilterInputStream FilterOutputStream (suite)

- **DataInputStream** et **DataOutputStream** permettent de lire et écrire des données formatées (byte, int, char, float, double, etc.)

  public DataInputStream (InputStream in);

  public DataOutputStream (OutputStream out);

Exemple:

```
IntputStream  s = new FileInputStream (dir + fichier);
DataInputStream  data = new DataInputStream  (s);

double  valeur = data.readDouble ();
```

FilterInputStream FilterOutputStream (suite)

- **LineNumberInputStream** numérote automatiquement les lignes lues.

```
public LineNumberInputStream (InputStream s);
public int  getLineNumber ();
public void  setLineNumber ();
```



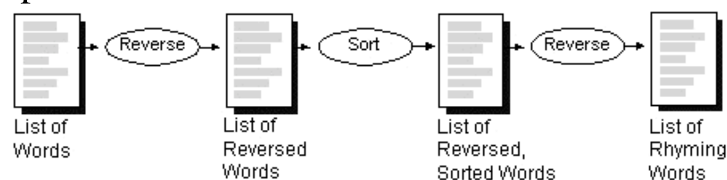
```
FileDialog f = new FileDialog (new Frame(), "ouvrir",
                                            FileDialog.LOAD );
f.show ();
String fichier = f.getFile ();
String dir = f.getDirectory ();

InputStream s = new FileInputStream( dir + fichier);
LineNumberInputStream  line =
                      new LineNumberInputStream (s);
DataInputStream  data = new DataInputStream  (line);

String  buffer;
buffer = data.readLine ();
System.out.println (buffer);              // affiche ligne 0
System.out.println (line.getLineNumber());   // affiche 1
```

FilterInputStream FilterOutputStream (suite)

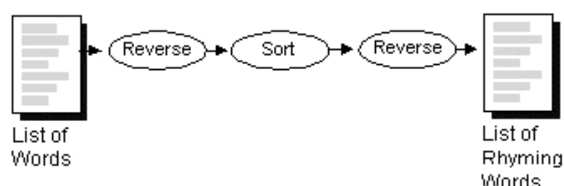
- La classe **`PushBackInputStream`** possède une méthode `unread()` qui permet de relire une seconde fois le dernier caractère lu
- La classe **`PrintStream`** possède des méthodes qui permettent d'afficher des éléments de différents types (de façon analogue à `System.out.println`)

Connexion avec pipe entre deux Threads

- sans pipe, il faudrait stocker les résultats intermédiaires quelque part:



- avec pipe:



PipedInputStream PipedOutputStream

- Ces deux classes permettent d'établir une connexion entre deux threads.

```
public PipedInputStream ( );
public PipedInputStream (PipedOutputStream src);

public PipedOutputStream ( );
public PipedOutputStream (PipedInputStream snk);
```



```
import java.io.*;

public class pipe  {
    public static void main(String args [ ]) throws IOException {
        PipedOutputStream sOut = new PipedOutputStream ();
        DataOutputStream envoie = new DataOutputStream(sOut);
        MyThread  thrd = new MyThread (sOut);
        thrd.start ();
        envoie.writeChars ("envoie de caracteres\n");
    }
}
```

```
class MyThread extends Thread  {
      DataInputStream  recoit;

      MyThread (PipedOutputStream  sOut) throws IOException {
         recoit =
               new DataInputStream (new PipedInputStream
(sOut));
      }
      public void run ()  {
         try {
              System.out.println(recoit.readLine());
         } catch (IOException  e)  {  }
      }
}
```

La classe `StreamTokenizer`

- Cette classe permet de découper un `InputStream` source en unités lexicales
- Elle est prévue à l'origine pour analyser des entrées de type Java
- Certains types d'unités lexicales ont des valeurs associées que l'on peut récupérer dans des champs de l'objet `StreamTokenizer`

La classe `StreamTokenizer`

- `nextToken()`: retourne le type de la prochaine unité lexicale et le met également dans le champ `ttype`. Les types prévus sont `TT_WORD`, `TT_NUMBER`, `TT_EOL`, `TT_EOF`.
- La valeur est placée dans le champ `sval` de type `String` pour les mots, et dans le champs `nval` de type `double` pour les nombres.


```
/** sumStream fait la somme des nombres envoyés dans un flot
   d'entrée jusqu'à lire autre chose qu'un nombre */
   static double sumStream(InputStream in)
                                         throws IOException {
       StreamTokenizer nums = new StreamTokenizer (in);
       double result = 0.0;
       while (nums.nextToken() ==
                            StreamTokenizer.TT_NUMBER )
            result += nums.nval;
       return result;
   }
```

```
/** readAttrs lit un fichier, à la recherche d'attributs de la forme
   nom=valeur, pour les stocker dans des objets de type
   AttribueImpl qui implémente l'interface Attribue qui
   stocke une liste d'Attribut (introduites pp.62 et  81-82 */
   public static AttribueImpl readAttrs(String fileName)
                                              throws IOException {
        FileInputStream fileIn = new FileInputStream (fileName);
        StreamTokenizer in = new StreamTokenizer (fileIn);
        AttribueImpl attrs = new AttribueImpl ();
        Attribut attr = null;
        in.commentChar('#'); // début de ligne de commentaire
        in.ordinaryChar('/'); // ne sera pas traité comme unité lexicale
                                  // (mais comme '=' ci-après)
```

```
        while (in.nextToken () != StreamTokenizer.TT_EOF  {
             if (in.ttype == StreamTokenizer.TT_WORD  {
                  if (attr != null) {
                       attr.valueOf(in.sval);
                       attr = null;    // on l'a totalement utilisé
                  } else {
                       attr = new Attr(in.sval);  // attr != null
                       attrs.add(attr); // on attend un  '='
                  }
             } else if (in.ttype == '=') {
                  if (attr == null)
                     throw new IOException (" '=' mal placé ")
             }
```

```
            else {  // on attend une valeur
                    if (attr == null) // on attendait un mot ??
                            throw
                              new IOException ("bad Attribut
 name");
                    attr.valueOf(new double(in.nval));
                    attr = null;    // on attend un mot maintenant
            }
    } // end of while
    return attrs;
}
```

La classe StreamTokenizer

reconnaît plusieurs types de caractères, qui sont définis par les méthodes suivantes:

```
public void wordChars(int low, int hi)
public void whitespaceChars(int low, int hi)
public void ordinaryChar (int ch)
public void ordinaryChars (int low, int hi)
public void commentChar (int ch)
public void quoteChar (int ch)
public void parseNumbers()
public void resetSyntax()
```

La classe `StreamTokenizer`

Valeurs par défaut pour un nouvel objet :

```
wordChars('a', 'z');
wordChars('A', 'Z');
wordChars(128 + 32, 255);
whitespaceChars(0, ' ');
commentChar('/');
quoteChar('"');
quoteChar(' \' ');
parseNumbers();
```


La classe `StreamTokenizer`

On a aussi :

```
public void eolIsSignificant (boolean flag)
public void slashStarComments(boolean flag)
public void slashSlashComments(boolean flag)
public void lowerCaseMode(boolean flag)
```

➤ Attention: valeur par défaut de flag = False

```
public void pushBack ( ) // pour une seule unité lex.
public int lineno( ) // numéro de ligne (en cas d' erreur)
public String toString() // pour la dernière unité lexicale
                         // rencontrée, + n° ligne inclus
```

Sérialisation d'objets

- Elle s'effectue *via* les classes ObjectOutputStream et ObjectInputStream
- Les objets sont écrits dans un fichier en appelant la méthode `writeObject()` de l'objet `ObjectOutputStream` correspondant au fichier
- Ils sont relus en appelant la méthode `readObject()` de l'objet `ObjectInputStream` correspondant au fichier

Résumé sur les flots

- Un chemin d'accès à un fichier est représenté par un objet de la classe `File` et un fichier physique par un objet du type `FileDescriptor`
- Les flots de sortie d'octets sont représentés par des sous-classes de la classe abstraite `OutputStream`. Ces classes permettent d'écrire dans un fichier
- Les flots de sortie de caractères sont représentés par des sous-classes de la classe abstraite `Writer`
- Les opérations de flots d'entrée d'octets sont définies par des sous-classes de la classe abstraite `InputStream`, et celles sur les flots d'entrée de caractères par les sous-classes de `Reader`

Résumé (suite)

- Les classes de flot de sorties filtrées peuvent être utilisées pour améliorer les fonctionnalités des classes de flot de sortie de base qui représentent un fichier physique
- Les classes de flots d'entrée filtrés complètent les classes de flots de sortie filtrés
- On peut utiliser la classe `StreamTokenizer` pour lire des données formatées à partir d'un flot
- Pour formater les données que vous écrivez dans un flot, vous devez mettre en œuvre vous-même une fonction de formatage

Les *Threads*

Les *Threads*

- *thread* = flot d'exécution dans un processus. Multi-processing ≠ multi-threading
- un seul *thread* est exécuté à la fois. C'est le code de la méthode `run()` qui est exécuté concurrement par le thread
- le *scheduler* java gère la vie des *threads* avec des procédures qui permettent de passer d'un état à un autre. On a ainsi l'impression de la simultanéïté.

Priorité des *Threads*

Une priorité est attribuée à chaque *thread*:

- Si deux `threads` ont même priorité, ils partagent le processeur sur la base de premier entré, premier servi.
- Un `thread` de haute priorité prend immédiatement le contrôle.
- Un `thread` de basse priorité doit attendre la fin, ou la mise en sommeil des autres.

La classe `ThreadGroup`

- Les threads sont séparés en groupes pour des raisons de sécurité.
- Les groupes sont inclus les uns dans les autres.
- Un thread ne peut pas modifier de threads en dehors de son groupe et de ceux qu'il contient.
- Le comportement par défaut consiste à créer chaque nouveau thread dans le même groupe que le thread qui l'a créé.

Création d'un *Thread*

Classe de base : java.lang.Thread

Mais deux manières de créer des *threads* :

- Ecrire une sous-classe de Thread qui implémente la méthode `run()`
- Ecrire une classe qui implémente l'interface `Runnable`

Création d'un *Thread*

- Utilisation de la classe java.lang.Thread.
-
```
class MonThread extends Thread {
//  … constructeurs éventuels …
        public  void run () {
                // corps du thread
        }
}
MonThread thread = new MonThread ();
thread.start ();        // lance la méthode run()
```

```
import java.io.IOException;
public class TryThread extends Thread {
   private String FirstName;
   private String SecondName;
   private long aWhile;
   public TryThread(String firstName, String SecondName, long delay ) {
        this.firstName = firstName;  this.secondName = secondName;
        aWhile = delay;
        setDaemon(true); // On ne peut pas le tuer! C'est un démon.
   }
   public static void main(String[] args) {
        Thread first = new TryThread(«Hopalong», «Cassidy», 200L);
        Thread second = new TryThread(«Marylin», «Monroe», 300L);
        Thread third = new TryThread(«Slim», «Pickens», 500L);
```

```
        System.out.println(«Return pour arrêter…\n»);
        first.start(); second.start(); third.start();      // lance run()
        try {
                System.in.read();
                System.out.println(«Vous avez appuye sur Return…\n);
        }
        catch(IOException e ) {              // traiter l'exception IO
                System.out.println(e);          // l'imprimer
        }
        System.out.println("Fin de main()");
        return;
   }
   //  fin de la méthode main()
```

```
public void run() {
    try {
        while(true) {
            System.out.printl(firstName);
            sleep(aWhile);
            System.out.println(secondName +   "\n");
        }
    }
    catch(InterruptedException e )  {
        // traiter les interruptions de Thread
        // imprimer l'exception
        System.out.println(fisrtName + secondName + e );
    }
  }
}
```

Démon et *user thread*

- Dans l'exemple précédent, avec l'appel à `setDaemon(true)` le thread fils ne se terminera que quand le processus parent aura terminé son exécution.
- *user thread* : dans ce cas, il y a moyen de l'arrêter.

Arrêt d'un *Thread*

- Un *Thread* peut faire positionner un drapeau (*flag*) à un autre thread en appelant la méthode `interrupt()`.
- Le code de la méthode `run()` peut alors utiliser ce drapeau : la méthode `sleep()` vérifie si le thread a été interrompu et lance dans ce cas une exception `InterruptException`


```
import java.io.IOException;
public class TryThread extends Thread {
   private String Firstname;
   private String SecondName;
   private long aWhile;
   public TryThread(String firstName, String SecondName,
                                                   long delay ) {
            this.firstName = firstName;
            this.secondName = secondName;
            aWhile = delay;      /* setDaemon(true); */
   }
   public static void main(String[] args) {
            Thread first = new TryThread(«Butch»,
                                             «Cassidy», 200L);
            Thread second = new TryThread(«Marylin»,
                                             «Monroe», 300L);
            Thread third = new TryThread(«Slim», «Pickens»,
                                                   500L);
```

```
        System.out.println("Return pour finir…\n»);
        first.start(); second.start(); third.start();
        try {
                System.in.read();
                System.out.println («Vous avez appuye sur
  \Return…\n);
                first.interrupt();
                second.interrupt();
                third.interrupt();
        }
        catch(IOException e ) {       // traiter l'exception io
                System.out.println(e);  // montrer l'exception
        }
        System.out.println(«Fin de main()»);
        return;
      }
      // methode run() inchangée (cf. écran suivant)
  }
```

```
public void run() {
      try {
              while(true) {
                      System.out.printl(firstName);
                      sleep(aWhile);
                      System.out.println(secondName +   "\n");
              }
      }
      catch(InterruptedException e )  {
              // traiter les interruptions de Thread
              // montrer l'exception et de qui elle provient
           System.out.println(fisrtName + secondName + e );
      }
   }
}
```

Commentaire de l'exemple

- Le bloc `catch` est en dehors du `while` de la méthode run : donc après une interruption, la méthode run de chaque thread termine et donc le thread termine.

Synchronisation

- Deux *threads* accédant à une même donnée doivent synchroniser leurs accès
- La technique est de poser un verrou (*lock*) sur un objet
- Quand un objet est *verrouillé par un thread*, seul *ce thread* peut y accéder

➢ La synchronisation assure qu'une même ressource sera correctement traitée par plusieurs *threads* concurrents

Méthodes synchronisées

- Quand les méthodes d'une classe sont synchronisées, elles deviennent mutuellement exclusives sur une instance :

```
Class MaClasse {
        synchronized public void method1() {   }
        synchronized public void method2() {   }
        public void method3() {
        }
}
```


Méthodes synchronisées

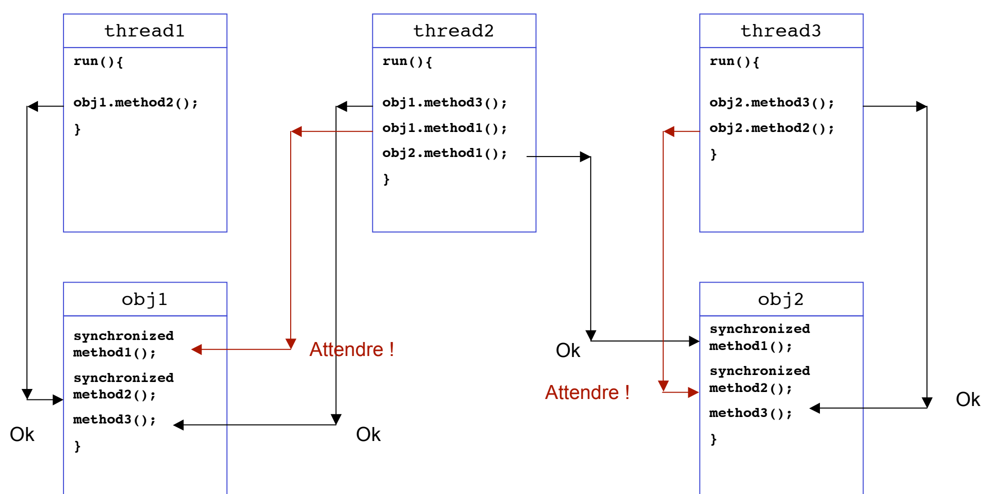
Si un *thread* invoque une méthode synchronisée sur un objet, cet objet est verrouillé :

```
class Compteur  {
    private int valeur;

    public Compteur(int initial) { valeur = initial;}
    synchronized void incremente ()  {
            valeur  += 1;
    }
    public synchronized getValeur() {return valeur;}
}
```

Méthodes synchronisées

- Il n’y a pas de contraintes sur l’exécution de méthodes synchronisées pour deux objets différents, instances (différentes) d’une même classe.
- C’est l’accès concurrent d’un seul objet (ou instance) qui est contrôlé par un thread.

Méthodes synchronisées



Méthodes synchronisées

- Si une classe étendue redéfinit une méthode synchronisée, la nouvelle méthode peut l'être ou ne pas l'être.
- Les méthodes de classes (`static`) peuvent aussi être synchronisées. Dans ce cas, le verrou est posé sur la classe.
- Les méthodes statiques synchronisées n'ont aucun effet sur les objets de cette classe. Seules les autres méthodes statiques synchronisées sont bloquées.

Bloc synchronisé

- Même idée. On peut déclarer `synchronized` une instruction ou un bloc de code dans le programme.
- Une instruction `synchronized` a deux parties: un objet à verrouiller et une instruction ou un bloc à exécuter.

```
synchronized (object )
        instruction;
```

Bloc synchronisé

```
public static void abs (int [ ] values)  {
  synchronized (values)  {
        for (int i = 0; i < values.length; i++)  {
              if (values [i] <0 )
                    values [i] =  -values [i];
        }
  }
}
```


<u>Exemple1:</u>

Synchronisation sur une instance (ici *this*) : évite de déclarer toutes les procédures d'accès `synchronized`

```
class Point  {
      private float  x, y;
      float x () { return x;}  // pas besoin de synchronized
      float y () { return y;}  // pas besoin de synchronized
      void print () {
            float safeX, safeY;
            synchronized (this) {
                  safeX = x; safeY = y;
            }
            System.out.println ("voilà x et y : " + x + y);
      }
}
```

Exemple2 : Protection d'une variable de classe

```
class Compteur {
        private static int valeur; // variable de classe

        void incremente () {
                synchronized (getClass()) {
                        valeur += 1;
                }
        }
        …
}
```

`public final class getClass()`: renvoie la classe de l'objet. *Tous* les objets de classe `Compteur` pourront être bloqués lors d'une exécution par l'un d'entre eux de la méthode `incremente`.

Déclaration `volatile`

- Si l'on décide de ne pas utiliser de mécanisme de synchronisation, plusieurs threads peuvent potentiellement modifier un champ au même instant.
- La déclaration `volatile` permet d'empêcher le compilateur de faire des optimisations sur une valeur qui pourrait en réalité être modifiée par d'autres threads (mais sembler invariable).

Exemple

Ici `currentValue` est affichée toutes les secondes par un thread graphique, et on sait qu'elle peut être modifiée par d'autres threads.

```
volatile currentValue = 5;
for ( ;  ; ) {
        display.showValue(currentValue);
        Thread.sleep(1000);
}
// sans la mention volatile, un compilateur
// pourrait optimiser la boucle en utilisant la valeur 5 …
```

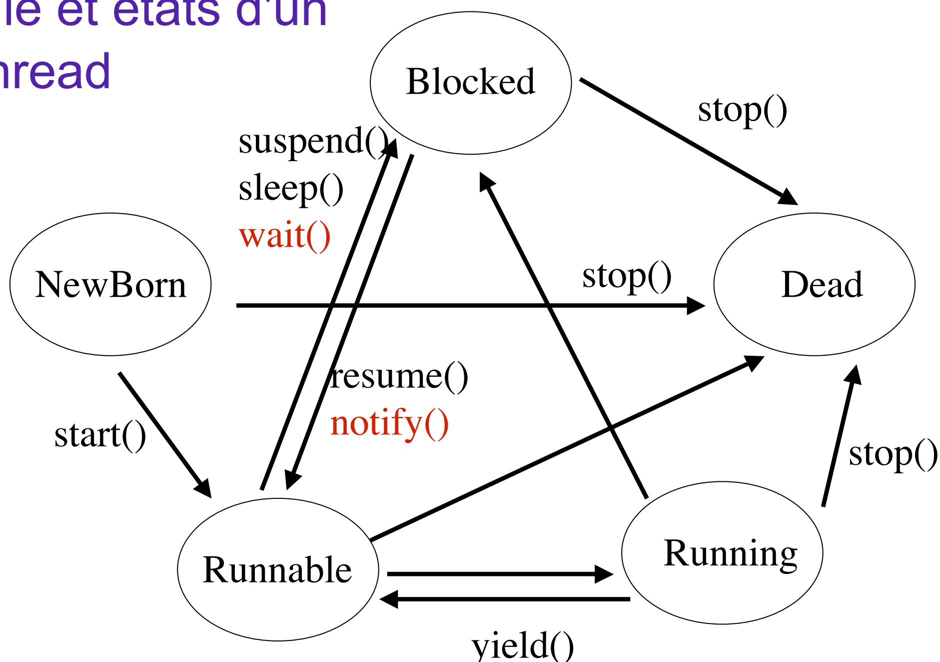
Ordonnancement des *threads*

- Si on utilise des synchronisations avec des verrous, il faut prendre garde aux inter-blocages.
- On peut ordonner les *threads* en utilisant des méthodes explicites pour endormir le *thread* courant, ou abandonner son exécution au profit d'un autre (`sleep` et `yield`)

Gestion de l'état d'un thread

start ()	lance la méthode run() du thread
stop ()	détruit le thread
sleep (long [,int])	endort pour (millis[,nanos]) approxim.
suspend ()	suspend le thread jusqu'à un resume()
resume ()	reprend une exécution suspendue
yield ()	interrompt pour passer au thread suivant
destroy ()	détruit brutalement le thread

Vie et états d'un thread



`wait()` et `sleep()`

- Différence entre ces deux méthodes : `wait()` attend et libère tout objet sur lequel le thread courant a posé un verrou, contrairement à `sleep()` qui ne fait qu'endormir le thread sans libérer les objets bloqués

Communication entre *threads*

- La granularité des méthodes ou des blocs synchronisés est souvent grossière. Il est possible d'être plus efficace en organisant la communication entre threads
- La classe racine de java, `Object`, définit les méthodes `wait()`, `notify()` et `notifyAll()` qui permettent cette communication

Communication entre *threads*

- Puisque toutes les classes dérivent de `Object`, toutes les classes héritent des méthodes `wait()`, `notify()` et `notifyAll()`.
- Mais vous ne pouvez utiliser ces méthodes qu'à l'intérieur d'une méthode ou d'un bloc synchronisé
- Sinon : `IllegalMonitorStateException`

Communication entre *threads*

- L'idée derrière `wait()` et `notify()` est simple et elle permet à des méthodes ou des blocs synchronisés de communiquer.
- Un bloc appelle `wait()` pour suspendre l'exécution d'un thread, jusqu'à ce qu'un bloc (ou une méthode) synchronisé sur le même objet appelle `notify()` pour réveiller le thread.

`wait()`

- Suspend le thread courant jusqu'à ce que `notify()` ou `notifyAll()` soient appelées

- On dispose également de :
  - `wait(long timeout)`
  - `wait(long timeout, int nanos )`

`notify()`

- Redémarre un thread qui a été arrêté par `wait()` (=> celui qui attend depuis le plus longtemps).
- S'il y a plusieurs threads, on ne peut pas contrôler finement le redémarrage de seulement certains d'entre eux car on ne dispose que de `notifyAll()`qui les réveille tous.

```
synchronized doWhenCondition()  {
        while  (!condition)
                wait();
        // puis on fait ce qu'il faut faire quand la condition vaut true
}
```

De l'autre côté, un notify signale la modification sur des données qu'un autre thread attend :

```
synchronized changeCondition() {
        //  change une valeur figurant dans un test de condition
        notify();
}
```

1. Le verrou est relâché "atomiquement" quand le thread est suspendu pour attendre (le verrou peut être relâché et un notify arriver avant que le thread n'ait été suspendu).
2. Ne pas remplacer le while par un if (si on a été réveillé, ça ne signifie pas que la condition ait été satisfaite)

Exemple: Classe implémentant une queue. On peut ajouter ou supprimer des éléments de la queue dans des méthodes synchronisées.

```
class Queue  {
        // le premier et le dernier Element de la Queue
        Element  head, tail;

        public synchronized void append (Element p) {
                if (tail == null)
                        head = p;
                else
                        tail.next = p;
                p.next = null;
                tail = p;
                notify(); // prévient ceux qui attendent
        }
```

```
    public synchronized Element get ( ) {
        try  {
            while (head == null)
                wait ();   //  attente d'un élement
        }   catch  (InterruptedException e)  {
            return  null;
        }

        Element p = head;  // on garde le premier
        head = head.next;  // on le retire de la queue
        if (head == null)  // on regarde si la queue est vide
            tail = null;
        return p;
    }
}
```

Interface Runnable

- C'est une autre solution pour implémenter des *threads* : écrire d'abord une classe qui implémente l'interface `Runnable`
- Avantage : on bénéficie d'une classe de *threads* sans pour autant hériter de la classe `Thread` : on peut ainsi hériter d'une autre classe tout en profitant des *threads* (rappel : pas d'héritage multiple en java)

Interface `Runnable`

Cette interface ne définit que la méthode run(). On dispose ensuite de quatre constructeurs de `Thread` qui utilisent un objet `Runnable` :

```
public Thread (Runnable  target)
public Thread (Runnable  target, String  name)
public Thread (ThreadGroup group, Runnable  target)
public Thread (ThreadGroup group, Runnable  target,
                    String  name)
```


Interface `Runnable`

```
class MesThreads extends ClasseChose implements Runnable {
        public  void run () {
                // corps du thread
        }
}

MesThreads objet = new MesThreads ();
Thread thread = new Thread (objet);
thread.start ();
```