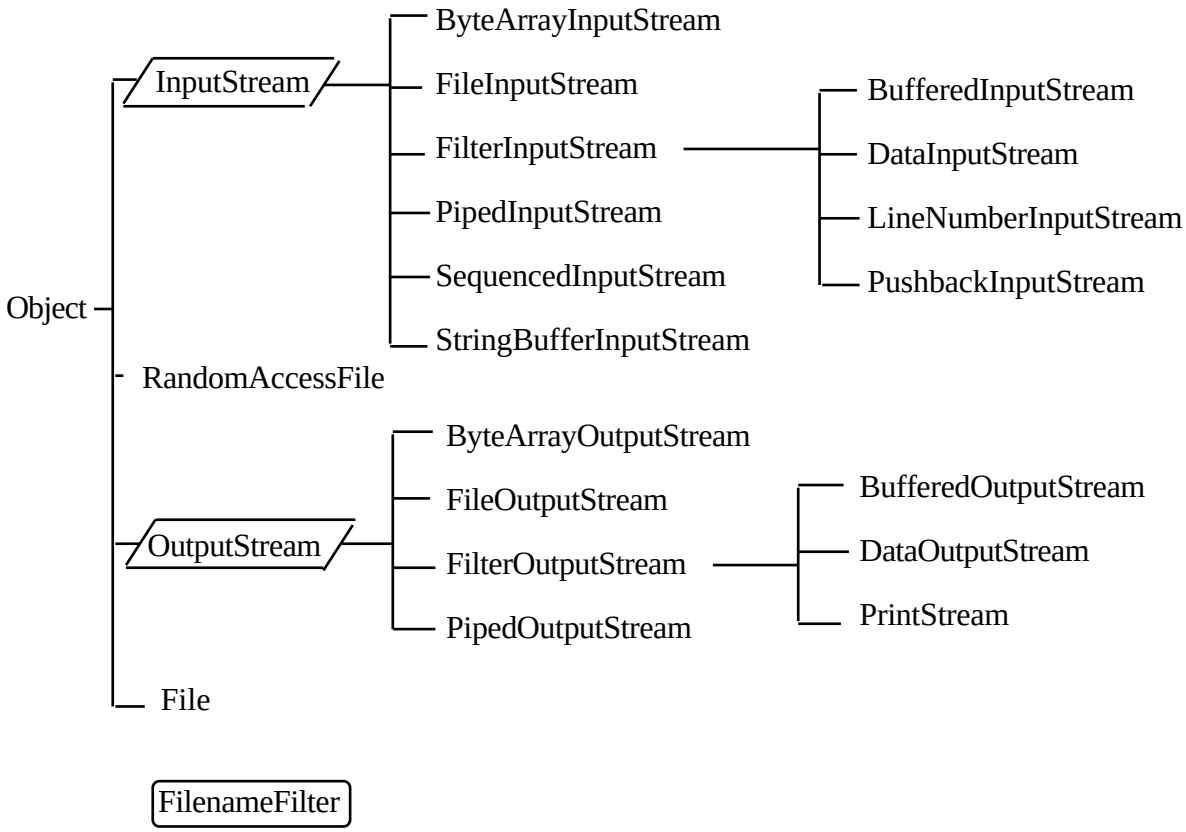
Entrées Sorties

Introduction

Java possède une grosse bibliothèque de classes et d'interfaces pour les entrées sorties. Cette bibliothèque se trouve dans le paquetage `java.io`. On l'utilise par la suite pour lire et écrire à partir de toutes source ou destination de lecture ou d'écriture comme les fichiers, les URL, les sockets, etc. L'arborescence complète en Java 1.0 est :

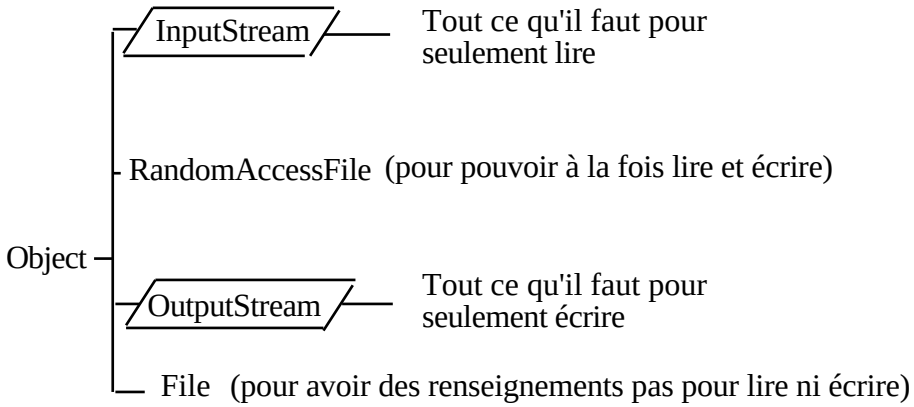


`InputStream` et `OutputStream` sont des classes abstraites
`FilenameFilter` est une interface.

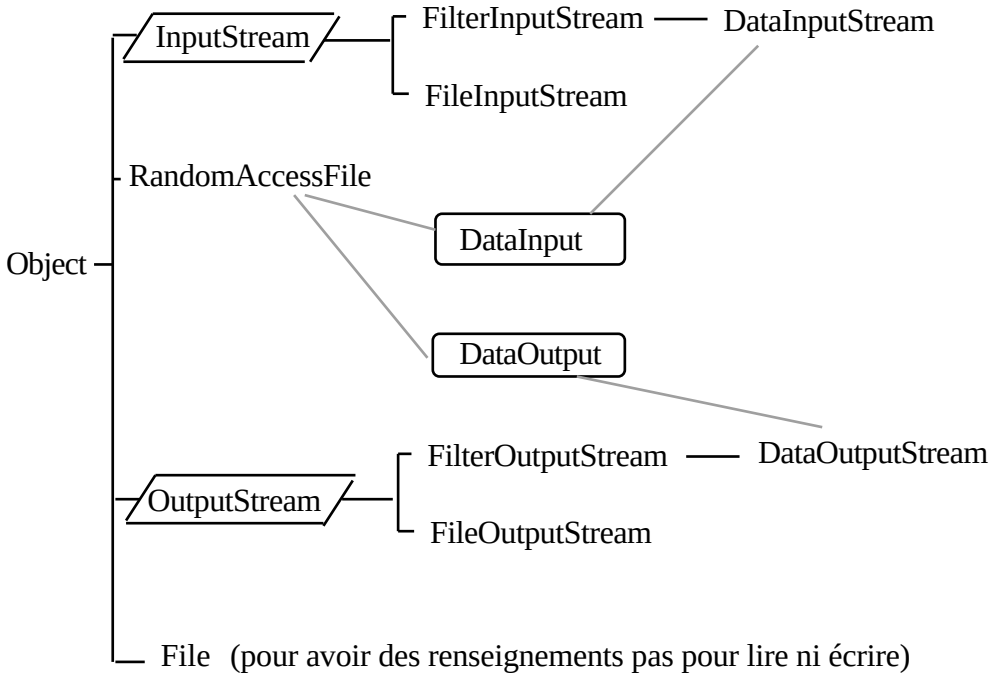
très souvent les méthodes de ces classes sont sous contrôle de `IOException`.

Les E/S en Java 1.0 : ce qu'il faut retenir

Le début de cette arborescence :



Les classes `DataXXXputStream` (et seulement elles) contiennent les méthodes pour lire ou écrire tout type primitif de données.



paquetage java.io

Ces classes et interfaces sont dans le paquetage `java.io`.

La technique la plus commune pour utiliser les entrées/sorties en Java est d'utiliser les classes `DataXXXputStream` (`XXX` = `In` ou `Out`). Le flot dans lequel on veut lire ou écrire est repéré par un objet de ces classes, initialisé par le seul constructeur `DataXXXputStream(XXXputStream)` et comme `XXXputStream` est classe de base des 2 arborescences fondamentales, on peut passer dans cet argument tout objet de classes dérivées (et donc des fichiers, des pipes entre threads, des sockets, ...).

De même pour des ressources dans lesquelles on peut lire ou écrire (URL, socket, ...), il suffit de récupérer un tel objet associé à cette ressource pour pouvoir la manipuler.

Pour écrire des chaînes de caractères on utilise plutôt la classe `PrintStream` (dont `out` et `err` sont des objets particuliers) et la méthode `println()`.

Entrées/sorties et applets

Rappelons qu'a priori les applets Java ne peuvent pas :
- lire ou écrire dans des fichiers locaux
- détruire des fichiers locaux (soit en se servant de `File.delete()` soit en lançant une commande système `rm` ou `del`)
- renommer des fichiers locaux (soit en se servant de `File.renameTo()` soit en lançant une commande système `mv` ou `rename`)
- créer un répertoire sur le système local (soit en se servant de `File.mkdir()` soit ou `File.mkdirs()` en lançant une commande système `mkdir`)
- lister le contenu d'un répertoire local.
- vérifier l'existence, la taille, le type, la date de dernière modification d'un fichier local

Ce chapitre est donc essentiellement utilisable pour les applications Java.

Classes pour les E/S

`File` : classe modélisant un fichier de manière indépendante du système. Cette classe fournit des méthodes pour lister les répertoires, interroger des attributs de fichiers (droits en lecture, en écriture), renommer ou détruire des fichiers.

`FileNameFilter` : interface possédant une méthode `accept()` (à implanter) permettant de n'accepter que certains noms de fichiers : utile pour implanter des filtres.

`RandomAccessFile` : permet de lire ou d'écrire dans un fichier et de s'y déplacer.

`InputStream` et `OutputStream` : classes abstraites qui définissent des méthodes pour lire et écrire.

`FileInputStream` et `FileOutputStream` : permet de lire et écrire sur fichiers.

Classes pour les E/S

`FilterInputStream` et `FilterOuputStream` sont des classes de base permettant de filtrer les entrées dans un `InputStream` ou un `OutputStream`. On se sert en fait de leurs classes dérivées.

`BufferedInputStream` et `BufferedOutputStream` font de la bufferisation sur les E/S de fichiers.

`DataInputStream` lit les octets (du binaire) et le renvoie dans le format indiqué (i.e. `readDouble()` renvoie un `double`). Peut être la classe la plus utilisée.

`DataOuputStream` écrit du format binaire.

`PrintStream` écrit sous forme ASCII ce qu'on lui passe comme du binaire (formatage).
`PushbackInputStream` permet de remettre un byte dans l'`InputStream` associé.
`LineNumberInputStream` indique et positionne le numéro de ligne dans un fichier texte.

La classe File

Le terme de fichier est pris dans son sens le plus général.
Les principaux contructeurs et méthodes de cette classe sont :

```
File(File rep, String nom)
File(String chemin)
File(String rep, String nom)
```

`public String getName()`
retourne le nom du fichier (sans le nom du répertoire)

`public String getParent()`
retourne le nom du répertoire père ou `null` quand le nom du fichier est donné sans répertoire ou quand ce fichier se trouve dans le répertoire racine du système de fichiers.

`public boolean exists()`
retourne `true` si le fichier existe, `false` sinon.

`public boolean canWrite()`
retourne `true` si l'utilisateur peut écrire dans le fichier, `false` sinon.

`public boolean canRead()`
retourne `true` si l'utilisateur peut lire dans le fichier, `false` sinon.

La classe File (suite)

`public boolean isFile()`
retourne `true` si le fichier est un fichier "normal" (pas un répertoire, ...), `false` sinon.

`public boolean isDirectory()`
retourne `true` si le fichier est un répertoire, `false` sinon.

`public long length()`
retourne la taille du fichier.

`public boolean mkdir()`
crée un répertoire et retourne true si la création s'est bien effectuée.

`public String[] list()`
retourne la liste des noms de fichiers (excepté . et ..) contenus dans l'instance (qui doit être un répertoire).

`public String[] list(FilenameFilter filtre)`
retourne la liste des noms de fichiers respectant le filtre contenus dans l'instance (qui doit être un répertoire).

Filtre sur les noms de fichiers

On utilise `public String[] list(FilenameFilter filtre)` de la classe `File` qui contient comme argument un filtre. C'est un objet d'une classe qui implémente l'interface `FilenameFilter`. Cette classe doit donner une définition de la méthode `public boolean accept(File rep, String nom)` et seuls les noms de fichiers qui renvoie `true` par cette méthode seront affichés. Voici un exemple.

```
import java.io.*;

class Filtre implements FilenameFilter {
	public boolean accept(File rep, String nom) {
		if (nom.endsWith(".java")) return true;
		return false;
	}
}

class AffFicJava {
	public static void main(String args[ ] )
		(new AffFicJava()).affiche(".");
	}
	public void affiche(String rep) {
		String nomFics[ ] = (new
File(rep)).list(new Filtre());
		for (int i = 0; i < nomFics.length; i++)
			System.out.println(nomFics[i]);
	}
}
```

les classes XXXputStream

Ce sont des classes abstraites qui fournissent des E/S de bas niveau (lecture ou écriture d'octets seulement). Elles sont souvent en arguments des constructeurs de `DataXXXputStream` pour avoir des lecture/écriture plus structurées. Refermer les flots lorsqu'on ne s'en sert plus par `close()`. Vider les flots d'écriture bufferisé par `flush()`.

classes FilterXXXputStream

Ces classes contiennent les méthodes de base pour faire des E/S structurés. En fait, on utilise plutôt leurs classes dérivées `DataXXXputStream` qui contiennent les E/S de type primitifs.

Classes FileXXXputStream

Ces classes permettent de manipuler les fichiers. Elles possèdent comme principaux constructeurs : `FileXXXputStream(File fic)`, `FileXXXputStream(String nomFic)`.

```
source = new FileInputStream(source_file);
destination = new FileOutputStream(destination_file);
buffer = new byte[1024];
while(true) {
  bytes_read = source.read(buffer);
  if (bytes_read == -1) break;
  destination.write(buffer, 0, bytes_read);
}
```

les classes DataXXXputStream

Ce sont ces classes qui possèdent des écritures ou lectures de plus au niveau que de simples octets (des entiers, des flottants, ...). On a par exemple :

dans DataInputStream ,

```
public final boolean readBoolean()
throws IOException,
public final char readChar() throws
IOException,
public final int readInt() throws
IOException, ...
```

dans DataOutputStream,

```
public final void
writeBoolean(boolean) throws
IOException,
public final void writeChar(int)
throws IOException,
public final void writeInt(int) throws
IOException, ...
```

Par exemple :

```
FileOutputStream outf = new FileOutputStream("test.bin");
DataOutputStream ds = new DataOutputStream( outf );
boolean varb = true;
char varchar = 'a';
int varint = 56;
ds.writeBoolean(varb):
ds.writeChar(varchar):
ds.writeInt(varint):
```

Exemple récapitulatif : programme de copie binaire

```
// Exemple provenant du livre _Java 1.0
// in a Nutshell_ par David Flanagan.

import java.io.*;

public class FileCopy {
  public static void main(String[] args) {
    if (args.length != 2)
      System.err.println("Usage: java FileCopy " +
            "<fichier source> <fichier destination>");
    else {
      try { copy(args[0], args[1]); }
      catch (IOException e) {
System.err.println(e.getMessage()); }
    }
  }

  public static void copy(String source_name,
String dest_name) throws IOException
  {
    File source_file = new File(source_name);
    File destination_file = new File(dest_name);
    FileInputStream source = null;
    FileOutputStream destination = null;
    byte[] buffer;
    int bytes_read;

```

```
    try {
      // On teste d'abord si le fichier existe et
      // si on a les droits de lecture.
      if (!source_file.exists() || !source_file.isFile())
        throw new FileCopyException("FileCopy: le
fichier : " + source_name + " n'existe pas");
      if (!source_file.canRead())
        throw new FileCopyException("FileCopy:
vous n'avez pas les droits de lecture sur le fichier :
" + source_name);

      // verifications similaires pour
      // le fichier dans lequel on va écrire.
      if (destination_file.exists()) {
        if (destination_file.isFile()) {
          DataInputStream in = new
DataInputStream(System.in);
          String response;

          if (!destination_file.canWrite())
            throw new FileCopyException("FileCopy:
vous n'avez pas les droits d'ecriture sur le fichier :
" + dest_name);

          System.out.print("Le fichier " + dest_name +
              " existe deja. Voulez vous l'effacer (O/N):
");
          System.out.flush();
          response = in.readLine();
          if (!response.equals("O") &&
!response.equals("o"))
            throw new FileCopyException("FileCopy:
copie annullee.");
        }
        else
          throw new FileCopyException("FileCopy: " +
dest_name + " n'est pas un fichier");
      }
```

```
      else {
       // le fichier n'existe pas.
      // On compte le creer dans le repertoire
      // donne en parametre. Encore faut il que
      // ce repertoire existe et qu'on puisse
      // ecrire dedans.
         File parentdir = parent(destination_file);
         if (!parentdir.exists())
           throw new FileCopyException("FileCopy: le
repertoire destination : " + dest_name + " n'existe
pas");
         if (!parentdir.canWrite())
           throw new FileCopyException("FileCopy:
vous ne pouvez pas ecrire dans le repertoire
destination : destination " + dest_name);
      }
       // Arrive ici tout est OK et la copie peut
       // etre effectuee.
       source = new FileInputStream(source_file);
       destination = new
FileOutputStream(destination_file);
       buffer = new byte[1024];
       while(true) {
         bytes_read = source.read(buffer);
         if (bytes_read == -1) break;
         destination.write(buffer, 0, bytes_read);
      }
    }
    // Dans tous les cas refermer les streams :
    // c'est plus propre.
    finally {
      if (source != null)
        try { source.close(); } catch (IOException e) { }
      if (destination != null)
        try { destination.close(); } catch (IOException
e) { ; }
    }
  }
```

```
  // File.getParent() retourne null quand le nom du
  // fichier est donne sans repertoire ou
  // quand ce fichier
  // se trouve dans le repertoire racine du systeme
  // de fichiers. Cette methode teste ces cas.
  private static File parent(File f) {
    String dirname = f.getParent();
    if (dirname == null) {
      if (f.isAbsolute()) return new
File(File.separator);
      else return new
File(System.getProperty("user.dir"));
    }
    return new File(dirname);
  }
}

class FileCopyException extends IOException {
  public FileCopyException(String msg) {
super(msg); }
}
```

classe RandomAccesFile

C'est la classe qui permet de :
- de manipuler des fichiers en lecture et en écriture
- de se déplacer dans un fichier (autrement que par `mark()` et `reset()` :

`RandomAccesFile` n'est pas un stream.

Les constructeurs sont :
`RandomAccessFile(String nomFic, String mode)`, `RandomAccessFile(File fic, String mode)`
les seules valeurs possible pour mode sont `"r"` (pour seulement lire) ou `"rw"` (pour lire et écrire).

On peut alors :
obtenir la position courante en nombre d'octets par rapport au début du fichier par `long getFilePointer()`, se déplacer à une position donnée par `void seek(long pos)`, avoir la taille du fichier par `long length()`.

Cette classe possède aussi toutes les lectures et écritures de haut niveau `XXXBoolean(...)`, `XXXChar(...)`, `XXXInt(...)`, ... (ici `XXX` = `write` ou `read`)

classe StreamTokenizer

associé à un InputStream grâce à :

```
public StreamTokenizer(InputStream in);
```

Java possède cette classe qui permet de découper l'entrée lue à partir du stream en token. Un objet de cette classe est donc un analyseur lexical (élémentaire).

Cette classe reconnaît comme fin de token :
StreamTokenizer.TT_EOF : fin de fichier
StreamTokenizer.TT_EOL : fin de ligne (\n, \r, \r\l,)
On peut avoir les tokens :
StreamTokenizer.TT_NUMBER : token représentant un flottant et déposé dans le champ double nval.
StreamTokenizer.TT_WORD : token représentant une String et déposé dans le champ String sval.

On passe de token en token à l'aide de la méthode nextToken().
Remarque
Indiquer que la fin de fichier est significative en lançant la méthode eolIsSignificant(true)

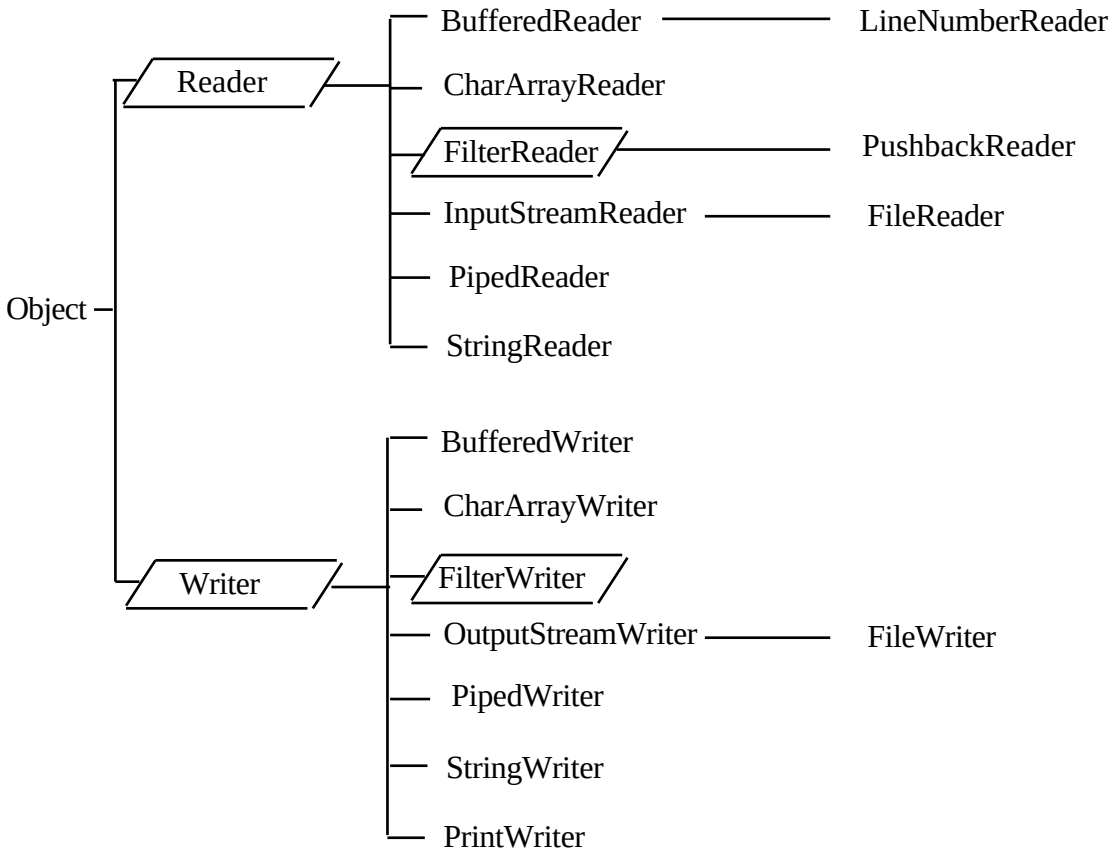
classe StreamTokenizer

Exemple

```
int i;
FileInputFile fic = new FileInputFile("toto.txt");
StreamTokenizer tf = new StreamTokenizer(fic);
tf.eolIsSignificant(true);
while ((i = tf.nextToken()) != StreamTokenizer.TT_EOF) {
        switch(i) {
        case StreamTokenizer.TT_EOL :
                ++nblignes; ++nbcars; break;
        case StreamTokenizer.TT_NUMBER :
                ++nbmots;
                nbcars += (new Double(tf.nval)).toString().length();
                break;
        case StreamTokenizer.TT_WORD :
                ++nbmots;
                nbcars += tf.sval.length();
                break;
        }
}
...
```

Les E/S texte en Java 1.1

Des arborescences `Reader` et `Writer` ont été ajoutés en Java 1.1 pour traiter les fichiers textes en accord avec l'internationalisation pris en compte dans Java 1.1.



La classe `FileReader` convertit les caractères d'un fichier écrits dans le codage "local" en unicode.

Un pb : comment lire au clavier !!

En Java 1.0 :

```
import java.io.*;

try {
    DataInputStream diclav = new DataInputStream
(System.in);
    String ligne = diclav .readLine();
} catch (IOException e) {•••}
```

En Java 1.1 :

```
import java.io.*;

try {
InputStreamReader isr = new InputStreamReader (System.in);
BufferedReader br = new BufferedReader (isr);
String ligne = br.readLine();
} catch (IOException e) {•••}
```

En fait les méthodes ci dessus sont celles à utiliser pour lire du "texte".

Les pipes à la Unix/Dos

On utilise les classes `Runtime` (pour récupérer l'environnement d'exécution) et `Process` :

```
import java.io.*;

class PipeEntreProcess {
  public static void main(String args[]) throws
IOException {
    Runtime r;
    Process p;
    InputStreamReader isr;
    BufferedReader br;
    String st;

    r = Runtime.getRuntime();
    p = r.exec("/bin/ls");
    isr = new
InputStreamReader(p.getInputStream());
    br = new BufferedReader (isr);

    while ((st = br.readLine()) != null)
      System.out.println(st);
  }
}
```

E/S avec des URL

Java propose des lectures de fichiers repérés par leur URL. La méthode `openStream()` de la classe `URL` retourne un `InputStream` qui sert ensuite pour lire. Exemple de code dans une applet :

```
import java.io.*;
// Pour la classe URL. En Java 1.1 il faut importer java.net.

InputStream is;
String nomFic = "index.html";
byte buffer[ ] = new byte[1024];

try {
        is = (new URL(getDocumentBase(),
nomFic)).openStream();
        is.read(buffer, 0, buffer.length);
} catch (IOException e) { }
```

Sérialisation

C'est une notion ajoutée en Java 1.1.
C'est le mot Java pour persistance i.e. garder les données même après la fin de l'exécution du programme qui les a fait naître. Il provient du monde Windows/Unix où les fichiers ne sont vus que comme suite d'octets les uns après les autres (i.e. en série).

En Java, seules les données (et les noms de leur classe ou type) sont sauvegardées (pas les méthodes ou constructeurs). Si un objet contient un champ qui est un objet, cet objet inclus est aussi sérialisé et on obtient ainsi un arbre (un graphe) de sérialisation d'objets.
Des objets de certaines classes ne peuvent pas être sérialisés : c'est le cas des Threads, des FileXXXputStream, ... On dit parfois que ces objets sont transient. Il faut indiquer qu'il le sont s'ils sont inclus dans un autre objet à sérialiser par le mot Java `transient`. De même si on veut qu'un champ d'un objet ne soit pas sérialisé, on l'indique à l'aide de ce modificateur.

Sérialisation : syntaxe

Pour indiquer que les objets d'une classe peuvent être persistents on indique que cette classe implémente l'interface `Serializable`. Il y a aucune méthode à implanter dans cet interface qui joue seulement le rôle de marqueur pour dire que les objets peuvent être sérialisés.
Exemple :

```
import java.io.*;

class MaClasse implements Serializable {
        public transient Thread maThread;
        private String nom;
        private int total;
        •••
}
```

Par la suite, on peut sauvegarder les objets dans un fichier (souvent appelé `.ser`) à l'aide de la méthode `writeObject(Object)` de la classe `ObjectOutputStream`. On lie le fichier à un tel objet à l'aide du constructeur `ObjectOutputStream(OutputStream)`. De même pour lire un objet persistant avec `readObject(Object)` de la classe `ObjectInputStream`.

Beaucoup de classe de Java 1.1 implémente l'interface `Serializable`. C'est le cas de la classe `java.util.Date`.

Sérialisation : exemples

écriture d'objet persistant

```
import java.util.*;
import java.io.*;
public class EcritObjetPers {
    public static void main(String args[]) {
        Date d = new Date();
        FileOutputStream f;
        ObjectOutputStream s;
        try {
                f = new FileOutputStream ("date.ser");
                s = new ObjectOutputStream(f);
                s.writeObject(d);
                s.close();
        } catch (IOException e) { }
    }
}
```

lecture d'objet persistant

```
import java.util.*;
import java.io.*;
public class LitObjetPers {
    public static void main(String args[]) {
        Date d = null;
        FileInputStream f;
        ObjectInputStream s;
        try {
                f = new FileInputStream ("date.ser");
                s = new ObjectInputStream(f);
                d = (Date) s.readObject();
                s.close();
                System.out.println("date stockee : " + d);
        } catch (IOException e) { }
        } catch (ClassNotFoundException e) { }
    }
}
```