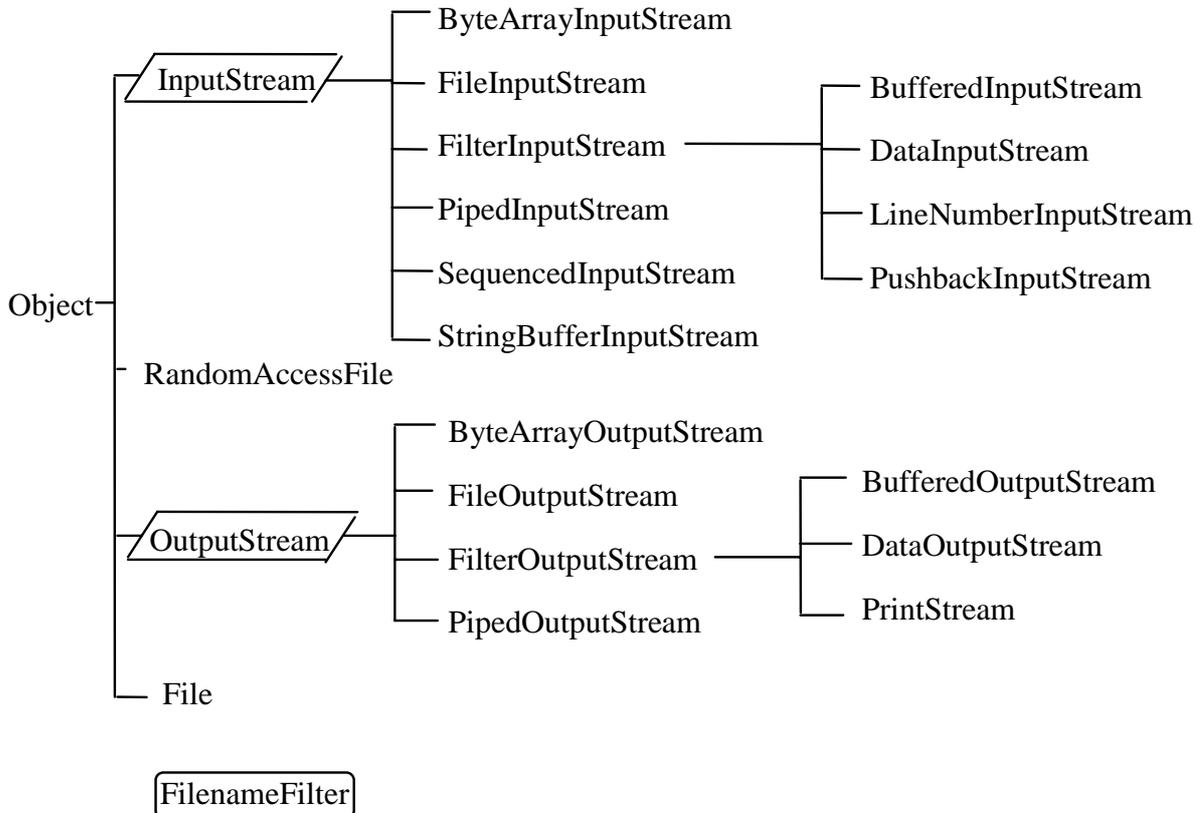


Entrées

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 toute source ou destination de lecture ou d'écriture comme les fichiers, les URL, les sockets, etc. Une partie de l'arborescence des classes est :



`InputStream` et `OutputStream` sont des classes abstraites

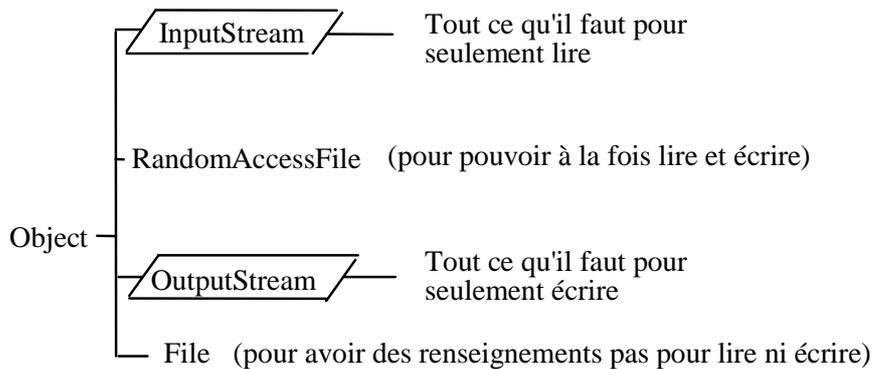
`FilenameFilter` est une interface.

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

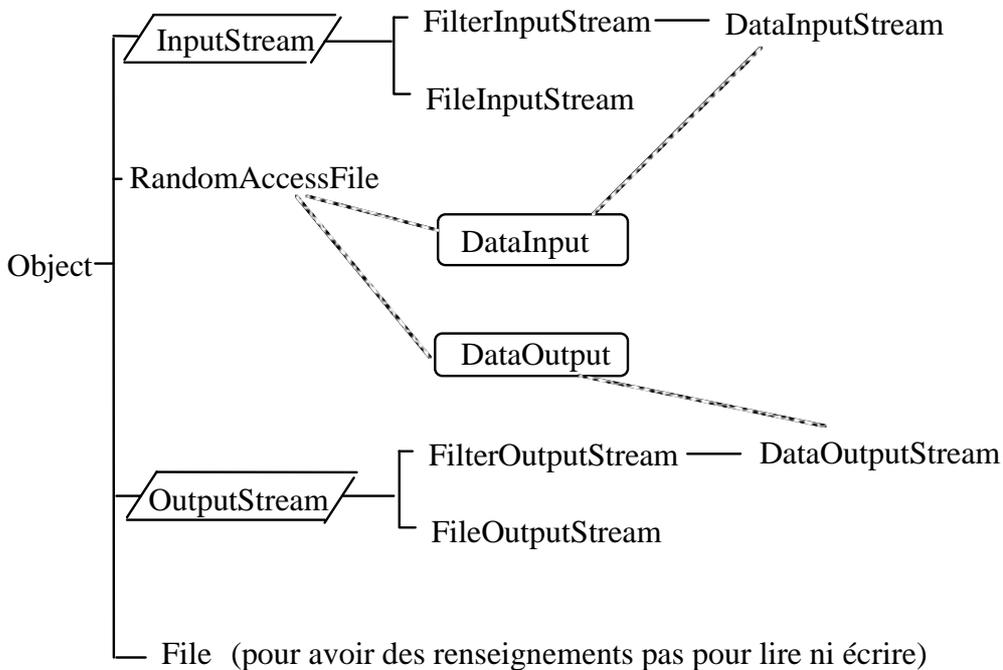
En Java 1.1 et suivant, les arborescences commençant par les classes `Reader` et `Writer` ont été ajoutées pour traiter les caractères.

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

Le début de cette arborescence :



Les classes de base ne permettent que de lire ou écrire des octets. D'autres classes XXXputStream (avec XXX = In ou Out) contiennent les méthodes pour lire ou écrire des type primitif de données ou des objets de classes.



Notion de filtre

La plupart des classes décrivent une fonction bien précise :

- lire, écrire des octets,
- lire, écrire des caractères,
- être lié à un fichier en lecture, être lié à un fichier en écriture,
- faire de la bufferisation, etc.

Mis bout à bout, on accomplit alors la fonction recherchée : écrire de manière bufferisée dans un fichier, etc.

Cette technique est dite la technique de filtre.

Syntaxiquement on utilise les constructeurs des classes par exemple on écrit :

```
BufferedReader in =  
    new BufferedReader(new FileReader("monFic.txt"));
```

On est alors prêt à lire de manière bufferisée dans le fichier `monFic.txt`.

L'intérêt de cette modularité est de pouvoir changer un des composants sans changer le reste du code. Par exemple (cf. chapitre programmation réseau) on écrira de manière similaire :

```
URL js = new URL("http://www.javasoft.com/");  
BufferedReader in =  
    new BufferedReader(  
        new InputStreamReader(js.openStream()));  
// suite du traitement d'un BufferedReader identique  
// a ci-dessus
```

et on est prêt à lire de manière bufferisée le contenu se trouvant à l'URL `http://www.javasoft.com/`.

L'astuce des entrées/sorties

La technique pour faire des entrées/sorties en Java est d'appliquer le slogan mai 68 (OK c'était il y a très longtemps) : « Prenons nos désirs pour des réalités ».

Si vous voulez lire une suite d'octets qui représente un double (c'est votre désir) il vous faudra utiliser une méthode `double readDouble()` (et il deviendra réalité). Il ne reste plus qu'à trouver une classe qui propose cette méthode.

Si vous voulez écrire une suite d'octets qui représente un objet (c'est votre désir) il vous faudra utiliser une méthode `void writeObject(Object)` (et il deviendra réalité). Il ne reste plus qu'à trouver une classe qui propose cette méthode.

Ces classes proposent des constructeurs qui ont comme argument, des références de classes mères des arborescences citées précédemment. Comme on peut mettre toute référence d'une classe dérivée de ces classes mère, il suffit de mettre la classe qui est associée au canal de communication dans lequel on veut lire ou écrire. Et voilà !!

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 (suite)

`FilterInputStream` et `FilterOutputStream` 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.

`DataOutputStream` 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 constructeurs 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()
```

créé 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

DataXXputStream

Ce sont ces classes qui possèdent des écritures ou lectures binaires 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()) {
            BufferedReader in = new
                BufferedReader(new
                    InputStreamReader(System.in));
            String response;
            if (!destination_file.canWrite())
                throw new FileCopyException("FileCopy:
vous n'avez pas les droits d'écriture 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 annulee.");
        } else /* if (destination_file.isFile()) */
            throw new FileCopyException("FileCopy: "
+ dest_name + " n'est pas un fichier");
        }
    }
```

```
    else /* if (destination_file.exists()) */ {
        // 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: " +
dest_name);
    } /* else if (destination_file.exists()) */
    // 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
finally {
    if (source != null)
        try { source.close(); } catch (IOException e) { }
    if (destination != null)
        try { destination.close(); } catch (IOException e) { }
```

```
    } // finally
} // (String source_name, String dest_name)

// 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"));
    } // if (dirname == null)
    return new File(dirname);
}

}

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

classe `RandomAccessFile`

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()`) : `RandomAccessFile` 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\n`.)

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)`

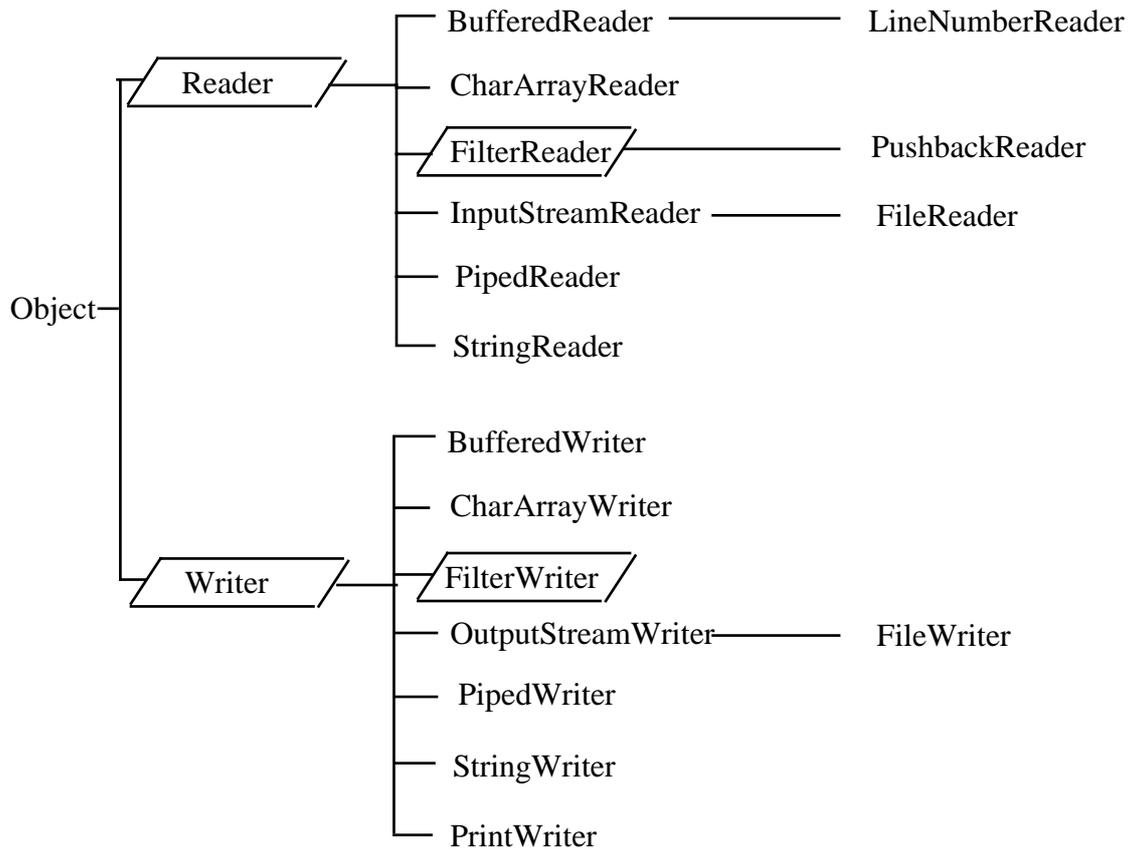
classe StringTokenizer

Exemple

```
int i;
FileInputFile fic = new FileInputFile("toto.txt");
StreamTokenizer tf = new StreamTokenizer(fic);
tf.eollsSignificant(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

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

!!

```
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 dans un fichier texte.

Manipuler les fichiers texte

`BufferedReader` est une classe possédant la méthode `readLine()` ainsi que le constructeur

`BufferedReader(Reader)`.

Pour lire une ligne d'un fichier texte, voici un code :

```
BufferedReader ibr = new BufferedReader(  
    new FileReader("monFic.txt"));  
String ligne = ibr.readLine();
```

Pour écrire on utilise plutôt la classe `PrintWriter` qui fournit les méthodes `print(...)`, ou `println(...)` surchargées avec différents types (`print(int)`, `print(double)`, ...) ainsi que le constructeur `PrintWriter(Writer)` et on peut écrire :

```
PrintWriter pw = new PrintWriter (  
    new FileWriter("monFic2.txt"));  
  
String line = "du texte";  
pw.println(line);
```

Pour les chaînes de caractères, on peut aussi utiliser les méthodes :

`void writeUTF(String)` de la classe `DataOutputStream` pour écrire,

`String readUTF()` de la classe `DataInputStream` pour lire.

Un mystère résolu : les accents dans une fenêtre dos

Sur windows, les fenêtres d'émulation MS-DOS utilisent le codepage Cp850. Pour afficher dans une telle fenêtre, il faut utiliser la constructeur à deux arguments de la classe `java.io.OutputStreamWriter` précisant dans son second argument l'encodage Cp850.

Voici un exemple complet :

```
import java.io.*;

public class DosString {
    public static void main(String args[]){
        String chaine = "caractères français : àùé";
        try {
            Writer w = new BufferedWriter(new
OutputStreamWriter(System.out, "Cp850"));
            w.write(chaine);
            w.flush();
            w.close();
        } catch (Exception e) {}
    }
}
```

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 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

écriture d'objet

```
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) { }
    }
}
```

Sérialisation (suite)

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();
        } catch (Exception e) { }
    }
}
```