Esial 1ère année

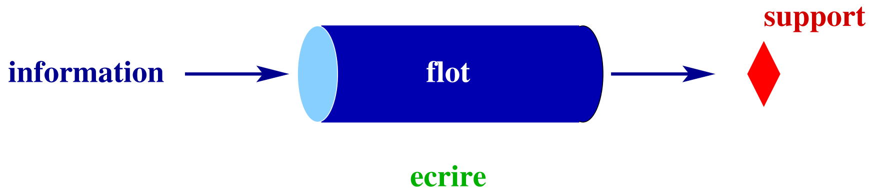
Martine Gautier Université H. Poincaré, Nancy
Martine.Gautier@loria.fr

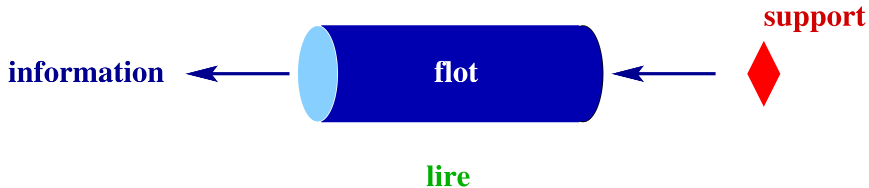
Année 2007-2008

Flots de données

Communication homme-machine

- Communication entre l'application et l'utilisateur : échange d'informations
  - *entrées* (input) = informations émanant de l'utilisateur, à destination de l'application
  - *sorties* (output) = informations émanant de l'application, à destination de l'utilisateur
- Package de gestion des entrées/sorties `java.io`
  $\leadsto$ Séparation entre l'information, appelée *flot* et le support


support
information
flot
ecrire


support
information
flot
lire

- *Flot de données*
  - octets (01001110, 11100010, *etc.*),
  - caractères ('A', '/', '1', *etc.*),
- *Support d'E/S*
  - entrée standard, sortie standard
  - tableau,
  - fichier
- Flots et supports différents se combinent.
  - avantage : le code est indépendant du périphérique utilisé
  - inconvénient : le code est plus compliqué à écrire

Une petite parenthèse : la classe `java.io.File`

- Une instance de `File` est un chemin d'accès dans un répertoire du système de fichiers.
  $\rightsquigarrow$ à ne pas confondre avec un flot

```
public class File {
  public File(String name) throws SecurityException { ... }
  public boolean exists() throws SecurityException { ... }
  public boolean canRead() throws SecurityException { ... }
  public boolean isDirectory() throws SecurityException { ... }
  public void delete() throws SecurityException { ... }
...
} // class File
```

Exemple

```
File chemin = new File("/home/etud/moi/java/essai.txt");
boolean estLa = chemin.exists();
boolean estRepertoire = chemin.isDirectory();
boolean aLire = chemin.canRead();
...
```

Déclenchement de l'exception `SecurityException` si l'accès est interdit
⇝ dans une applet
⇝ si un gestionnaire spécifique de sécurité est mis en place

Le package `java.io`

- Environ 80 classes, en dehors de la classe `File`.
  - une trentaine de classes instanciables
  - les autres : interfaces, classes abstraites et classes d'exceptions
- Etude exhaustive impossible et lassante ....
  - apprentissage des principes généraux, permettant d'utiliser n'importe quel flot
  - application aux flots de caractères avec un support de type fichier

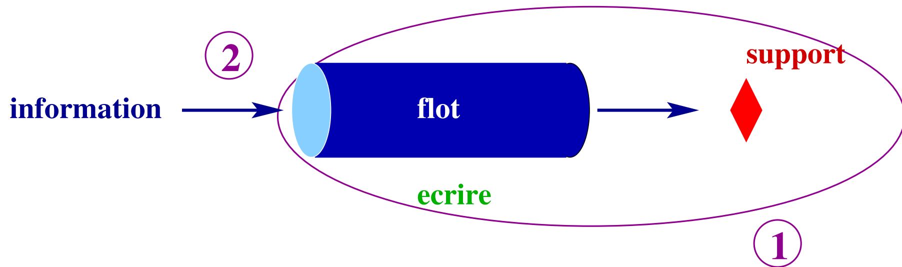
Structuration du package `java.io`

- Hiérarchie de classes héritant de `Writer` et `Reader`.
  - pour les entrées/sorties de caractères
- Hiérarchie de classes héritant de `OutputStream` et `InputStream`.
  - pour les entrées/sorties d'octets

Utilisation d'un flot de données

Principe identique pour les entrées et les sorties et pour les différents supports

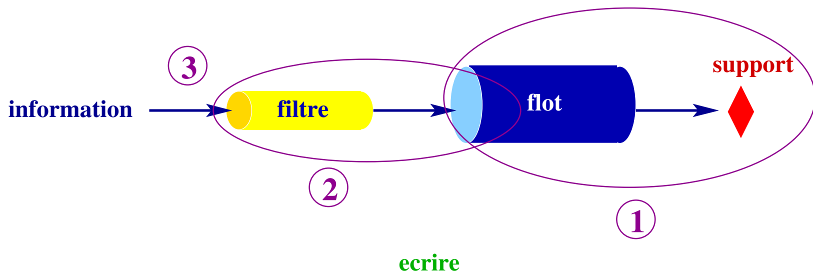
1. Créer un flot de données, càd un objet associé au support de l'information
2. Lire ou écrire sur le flot
3. Fermer le flot



Les flots de base peuvent être filtrés pour diverses raisons :

- amélioration de l'efficacité de la communication
- transformation de l'information

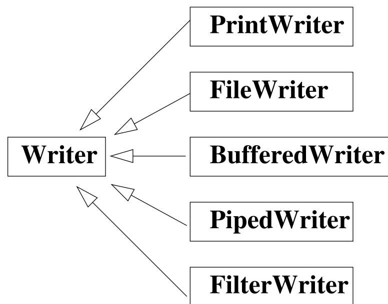
1. Créer un flot de données, càd un objet associé au support de l'information
2. Créer un filtre attaché au flot de données
3. Lire ou écrire sur le filtre
4. Fermer le filtre (qui ferme le flot)



Flot de sortie de caractères

Classe abstraite `Writer`

```
public abstract void write(String s) throws IOException;
    // écrit la chaîne sur le flot
public abstract void flush() throws IOException;
    // vide le flot
public abstract void close() throws IOException;
    // ferme le flot
```



Un exemple : écrire des caractères dans un fichier

`FileWriter` sous-classe de `Writer`

⇝ Implantation des méthodes `write`, `flush` et `close`

⇝ Constructeurs

```
public FileWriter(String nomFichier) throws IOException;
    // flot attaché à un fichier de nom donné
public FileWriter(File f) throws IOException;
    // flot attaché au fichier décrit par un chemin d'accès donné
```

```
import java.io.FileWriter;
...
FileWriter flot;
int x = 144;
String finDeLigne = System.getProperty("line.separator");
try {
    flot = new FileWriter("essai.txt");
    flot.write("Un nombre "+finDeLigne);
    flot.write(x + " ");
    flot.close();
} catch (IOException e) { ... }
```

Un autre exemple : bufferiser les informations, avec un filtre

- En utilisant `FileWriter`, on écrit physiquement dans le fichier à chaque instruction `write`.
  $\leadsto$ peu efficace, car écrire prend du temps
- Utiliser le filtre `BufferedWriter`
  - accumule les informations dans un tampon (buffer)
  - vide le tampon sur le flot lorsqu'il est plein
- Méthodes de `BufferedWriter`
  - méthodes `write`, `flush` et `close`
  - constructeurs de même nature que ceux de `FileWriter`
  - méthode `newLine()` pour écrire une fin de ligne

```
import java.io.FileWriter;
import java.io.BufferedWriter;
...
FileWriter flot;
BufferedWriter flotFiltre;
int x = 566;
try {
    flot = new FileWriter("essai.txt");
    flotFiltre = new BufferedWriter(flot);
    flotFiltre.write("Un nombre ");
    flotFiltre.newLine();
    flotFiltre.write(x + " ");
    flotFiltre.close();
} catch (IOException e) { ... }
```

Un autre exemple : transformer les informations, avec un filtre

- En utilisant `FileWriter`, on ne peut écrire que des `String`.
  ⇝ l'écriture des informations numériques nécessite une conversion préalable
- Utiliser le filtre `PrintWriter`
  - prend en charge la conversion des informations numériques
- Méthodes de `PrintWriter`
  - méthodes `flush` et `close`
  - constructeurs de même nature que ceux de `FileWriter`
  - méthodes surchargées `print(...)` et `println(...)` pour écrire une information numérique
    ⇝ `print(int i)`, `print(double d)`, etc.

```
import java.io.FileWriter;
import java.io.PrintWriter;
...
FileWriter flot;
PrintWriter flotFiltre;
int x = 5730;
try {
    flot = new FileWriter("essai.txt");
    flotFiltre = new PrintWriter(flot);
    flotFiltre.println("Un nombre ");
    flotFiltre.print(x);
    flotFiltre.close();
} catch (IOException e) { ... }
```

Un dernier exemple : transformer et bufferiser les informations, avec deux filtres

```
 import java.io.*;
...
FileWriter flot;
PrintWriter flotFiltre;
int x = 78;
try {
    flot = new FileWriter("essai.txt");
    flotFiltre = new PrintWriter(new BufferedWriter(flot));
    flotFiltre.println("Un nombre ");
    flotFiltre.print(x);
    flotFiltre.close();
} catch (IOException e) { ... }
```

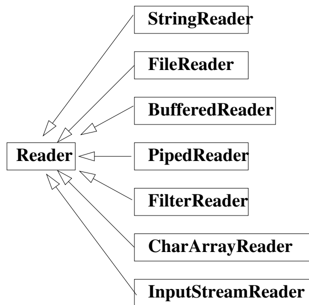
Le cas particulier de la sortie standard

- Sortie standard = flot particulier d'octets
  ⇝ on y accède par le champ statique `System.out` de type `PrintStream`
- Attention ce champ n'est pas de type `PrintWriter` comme on pourrait s'y attendre
  ⇝ flot d'octets au fonctionnement identique à `PrintWriter`
  ⇝ classe obsolète depuis la version 1.1
  ⇝ conservé pour des raisons de compatibilité de versions

Flot d'entrée de caractères

Classe abstraite `Reader`

```
public abstract int read() throws IOException ;
    // lit un caractère sur le flot, -1 si flot est terminé
public abstract void close() throws IOException ;
    // ferme le flot
```



Un exemple : bufferiser les informations, avec un filtre

- En utilisant simplement `FileReader`, on lit physiquement dans le fichier à chaque instruction `read`.
  $\rightsquigarrow$ même fonctionnement que `FileWriter`
  $\rightsquigarrow$ peu efficace, car lire prend du temps
- Utiliser le filtre `BufferedReader`
  - lit un tampon complet
  - distille les informations au fur et à mesure des lectures demandées
- Méthodes de `BufferedReader`
  - méthodes `read` et `close`
  - constructeurs analogues à ceux de `FileWriter`
  - méthode `readLine()` pour lire une ligne complète (`null` à la fin)

```
import java.io.FileReader;
import java.io.BufferedReader;
...
FileReader flot;
BufferedReader flotFiltre;
try {
    flot = new FileReader("essai.txt");
    flotFiltre = new BufferedReader(flot);
    String ligne = flotFiltre.readLine();
    while (ligne != null) {
        ligne = flotFiltre.readLine();
        ...
    } // while
} catch (IOException e) { ... }
```

Le cas particulier de l'entrée standard

Accessible par le champ statique `System.in` de type `InputStream`.

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
...
InputStreamReader flot;
BufferedReader flotFiltre;
try {
    flot = new InputStreamReader(System.in);
    flotFiltre = new BufferedReader(flot);
    ...
} catch (IOException e) { ... }
```

Un filtre évolué : la classe `StreamTokenizer`

- Une instance de cette classe est une forme simplifiée d'analyseur lexical.
  $\leadsto$ à utiliser pour filtrer un flot d'entrée de caractères
- Le filtre ignore les espaces et les fins de ligne.
- Il peut aussi ignorer les commentaires : `/* ... */` et `// ....`
- Les caractères lus dans le flot sont regroupés en unités lexicales.
  - nombre : chiffre + { . chiffre * }
  - mot : lettre { lettre | chiffre | . } *
- Les règles de construction des unités lexicales peuvent être modulées.

- Constructeur `StreamTokenizer(Reader r)`
- Ne pas ignorer les fins de lignes : méthode `eolIsSignificant(boolean)`
- Ignorer les commentaires : méthodes `slashStar(boolean)` et `slashSlash(boolean)`
- Consulter la prochaine unité lexicale : méthode `int nextToken()`
- Identifier sa nature en comparant le résultat de `nextToken()` avec
  - champ statique `int TT_NUMBER`
  - champ statique `int TT_WORD`
  - champ statique `int TT_EOF`
- Consulter la valeur de la dernière unité lexicale
  - champ `double nval` pour une unité lexicale numérique
  - champ `String sval` pour un mot

```
import java.io.*;
BufferedReader flot; StreamTokenizer flotFiltre;
try {
   flot = new BufferedReader(new FileReader("essai.txt"));
   flotFiltre = new StreamTokenizer(flot);
   flotFiltre.eolIsSignificant(true);
   int lu = flotFiltre.nextToken();
   while (lu!=StreamTokenizer.TT_EOF) {
     if (lu==StreamTokenizer.TT_NUMBER) {
        double numLu = flotFiltre.nval;
     ....
     } else {
        String chLue = flotFiltre.sval;
     ....
     }
      int lu = flotFiltre.nextToken();
   } // while
} catch (IOException e) { ... }
```

Des filtres très évolués : les classes `ObjectOutputStream` et `ObjectInputStream`

- L'utilisation des flots de caractères peut s'avérer très lourde à mettre en œuvre lorsque les informations sont des objets à part entière.
- Le filtre `ObjectOutputStream` transforme une information complexe (un objet) en une suite d'octets (et l'inverse pour `ObjectInputStream`).
- Contraintes
  - La classe de déclaration de l'objet doit être publique.
  - Elle doit implanter l'interface `Serializable`.
  - Pour un filtrage en entrée, elle doit proposer un constructeur accessible, sans paramètre.
  - Tous les champs sont de type primitif ou de type `Serializable`.
  - Un champ déclaré `transient` ne transite pas sur le flot.
- Filtres à attacher aux flots d'octets `FileOutputStream` et `FileInputStream`

```
 import java.io.*;
...
ObjectOutputStream flot;
Polygone pol = new Polygone(..);
try {
    flot = new ObjectOutputStream(new
                    FileOutputStream("polygone.txt"));
    flot.writeObject(pol);
    flot.writeObject(new Point(1., 2.));

    flot = new ObjectInputStream(new
                    FileInputStream("polygone.txt"));
    Polygone pol = (Polygone)(flot.readObject());
    Point pt = (Point)(flot.readObject());
} catch (IOException e) { ... }
```

L'utilisation de ces filtres peut déclencher des exceptions

- `InvalidClassException`
  - Il est impossible de reconstituer dans le flot d'entrée une instance de la classe demandée.
  - La classe n'a pas de constructeur sans paramètre accessible.
- `NotSerializableException`
  - La classe n'implante pas l'interface `Serializable`.
- `IOException`
  - Problème avec le support de l'E/S