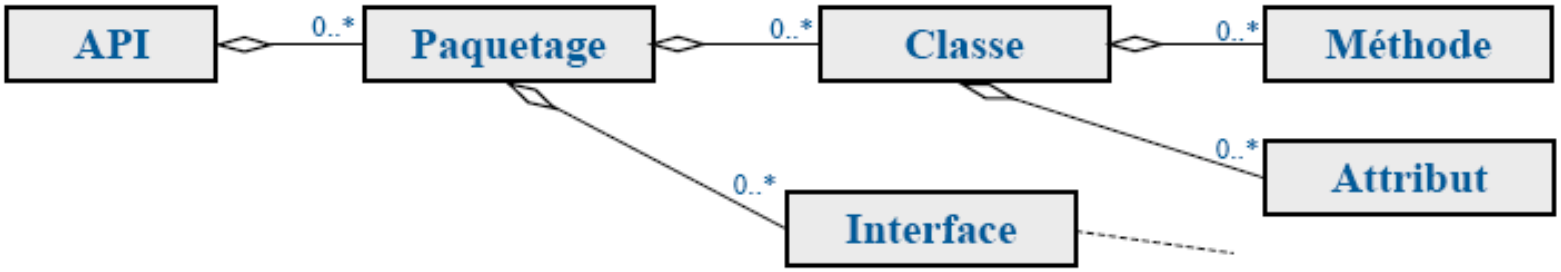
Les packages Java et E/S (Flux)

Les packages

- Le langage Java propose une définition très claire du mécanisme d'empaquetage qui permet de classer et de gérer les API externes
- Les API sont constituées :

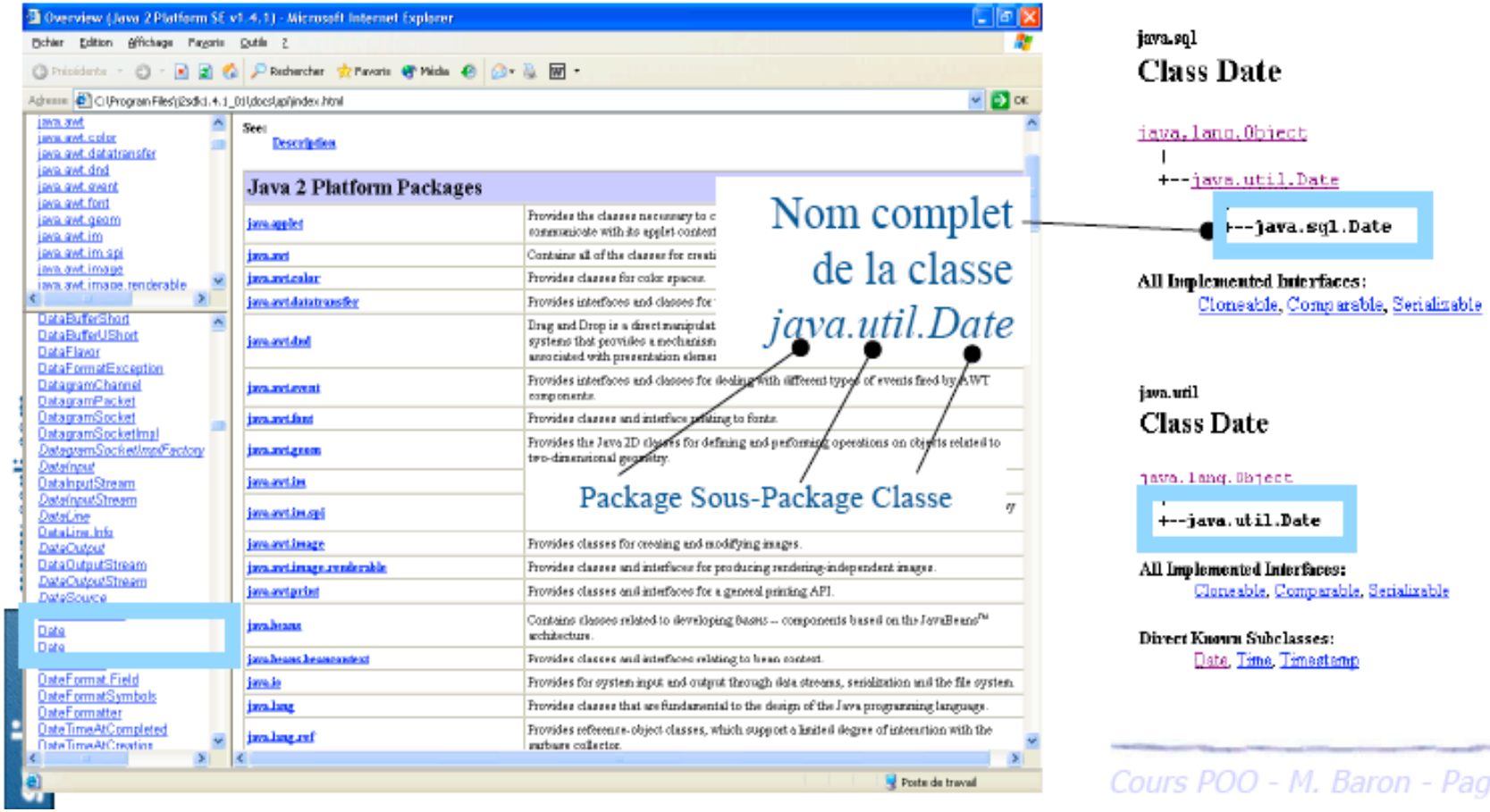


- Un package est donc un groupe de classes associées à une fonctionnalité
- Exemples de packages
  - *java.lang* : rassemble les classes de base Java (*Object*, *String*, *System*, …)
  - *java.util* : rassemble les classes utilitaires (*Collections*, *Date*, …)
  - *java.io* : lecture et écriture
  - …

www.serli.com
serli informatique

Les packages : ok, mais pourquoi faire?

- L'utilisation des packages permet de regrouper les classes afin d'organiser des libraires de classes Java
- Exemple : la classe *Date* est définie deux fois



Les packages : utilisation des classes

- Lorsque, dans un programme, il y a une référence à une classe, le compilateur la recherche dans le package par défaut (*java.lang*)
- Pour les autres, il est nécessaire de fournir explicitement l'information pour savoir où se trouve la classe :
  - Utilisation d'**import** (classe ou paquetage)

```
import mesclasses.Point;
import java.lang.String; // Ne sert à rien puisque par défaut
import java.io.ObjectOutput;
```

ou

```
import mesclasses.*;
import java.lang.*; // Ne sert à rien puisque par défaut
import java.io.*;
```

Ecriture très lourde préférer la solution avec le mot clé **import**

  - Nom du paquetage avec le nom de la classe

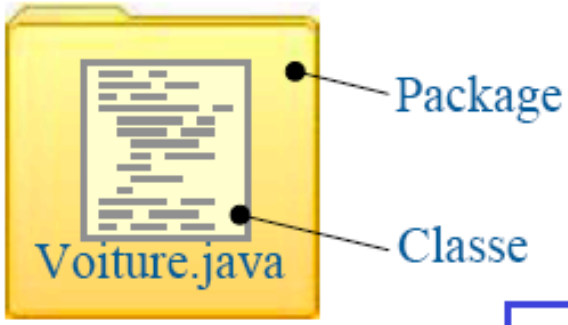
```
java.io.ObjectOuput toto = new java.io.ObjectOuput(...)
```

www.serli.com
serli informatique

Les packages : leur « existence » physique

- A chaque classe Java correspond un fichier
- A chaque package (sous-package) correspond un répertoire

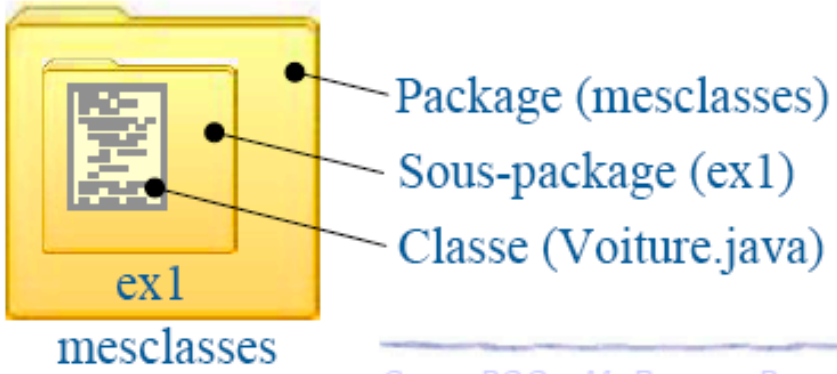
Exemple : *mesclassses.Voiture*



Le nom des packages est toujours écrit en minuscules

- Un package peut contenir
  - Des classes ou des interfaces
  - Un autre package (sous-package)

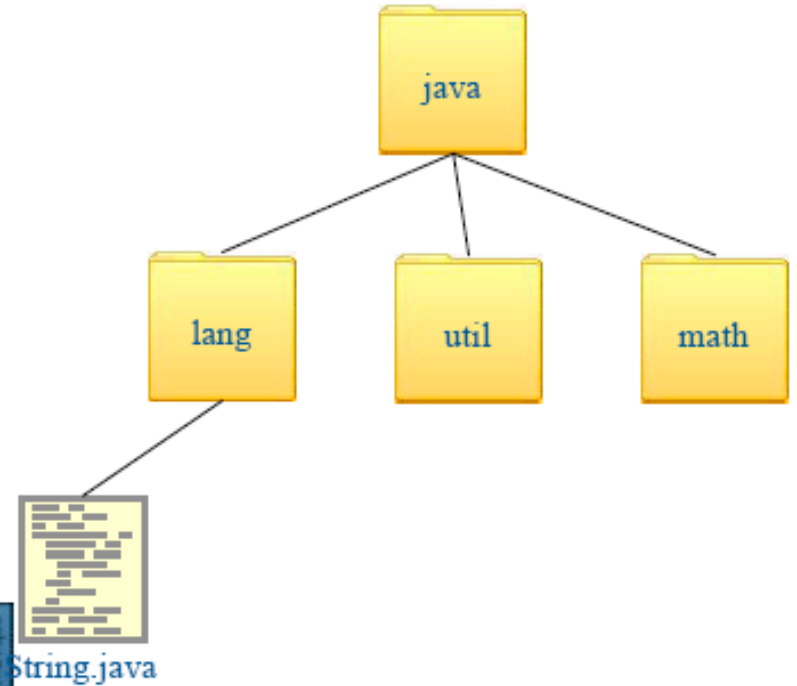
Exemple : *mesclassses.ex1.Voiture*



www.serli.com
Serli informatique

Les packages : hiérarchie de packages

- A une hiérarchie de packages correspond une hiérarchie de répertoires dont les noms coïncident avec les composants des noms de package
- Exemple : la classe *String*

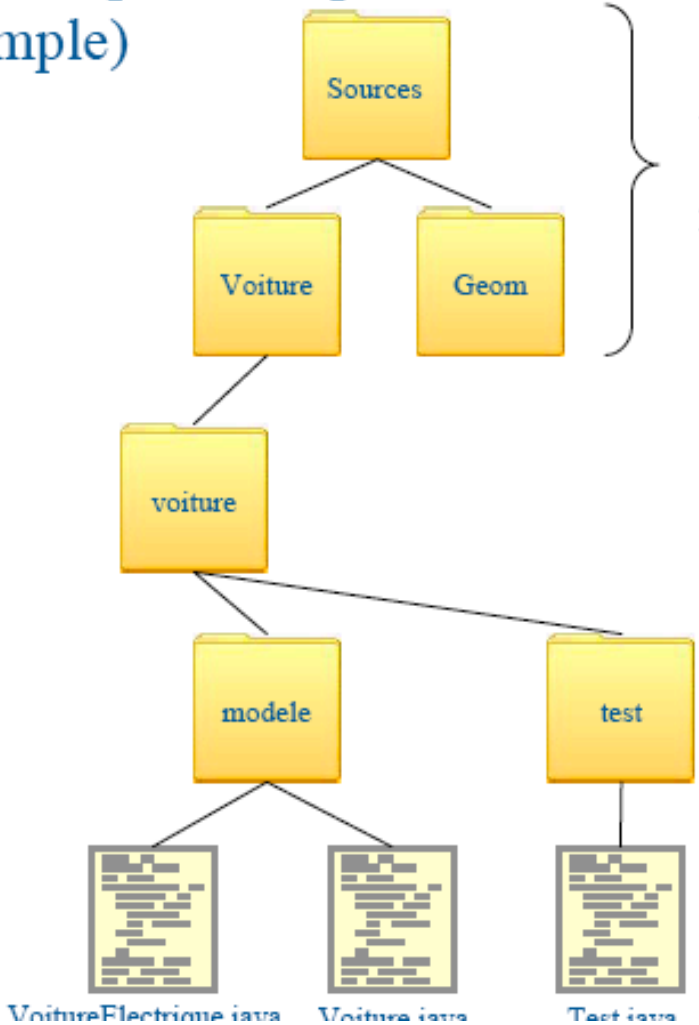


- Bibliothèque pure Java
- Les sources (*.java) se trouvent dans le répertoire *src* du répertoire Java
- Les bytecodes (*.class) se trouvent dans l'archive *rt.jar* du répertoire Java

www.serli.com
Serli informatique

Les packages : création et conseils

- Quand vous créer un projet (TP par exemple) nommer le package de plus haut (voiture par exemple) niveau au nom du projet (*Voiture* par exemple)



Vos répertoires de travail (Pas encore de notion de package)

Package voiture :

voiture.modele.VoitureElectrique
voiture.modele.Voiture
voiture.test.Test

Vos packages. A la racine de Voiture vous pouvez y placer des informations d'aides à l'utilisation de votre package. (Fichier de lancement par exemple)

Les packages : création et conseils

- Pour spécifier à une classe qu'elle appartient à une classe utiliser le mot clé **package**

Le mot clé package est toujours placé en première instruction d'une classe

```
package voiture.modele;
public class VoitureElectrique {
    ...
}
```

```
package voiture.modele;
public class Voiture {
    ...
}
```

```
package voiture.test;
import voiture.modele.VoitureElectrique;
import voiture.modele.Voiture;
import ...

public class Test1 {
    public static void main(String[] argv) {
        ...
    }
}
```

Ne confondez pas héritage et package. Pas le même chose. *VoitureElectrique* est dans le même package que *Voiture*

www.serli.com

Les packages : compilation et exécution

- Être placé dans la racine du répertoire Voiture

- La compilation doit prendre en compte les chemin des packages

```
javac voiture\modele\*.java voiture\test\*.java
```

- L'exécution se fait en indiquant la classe principale avec son chemin

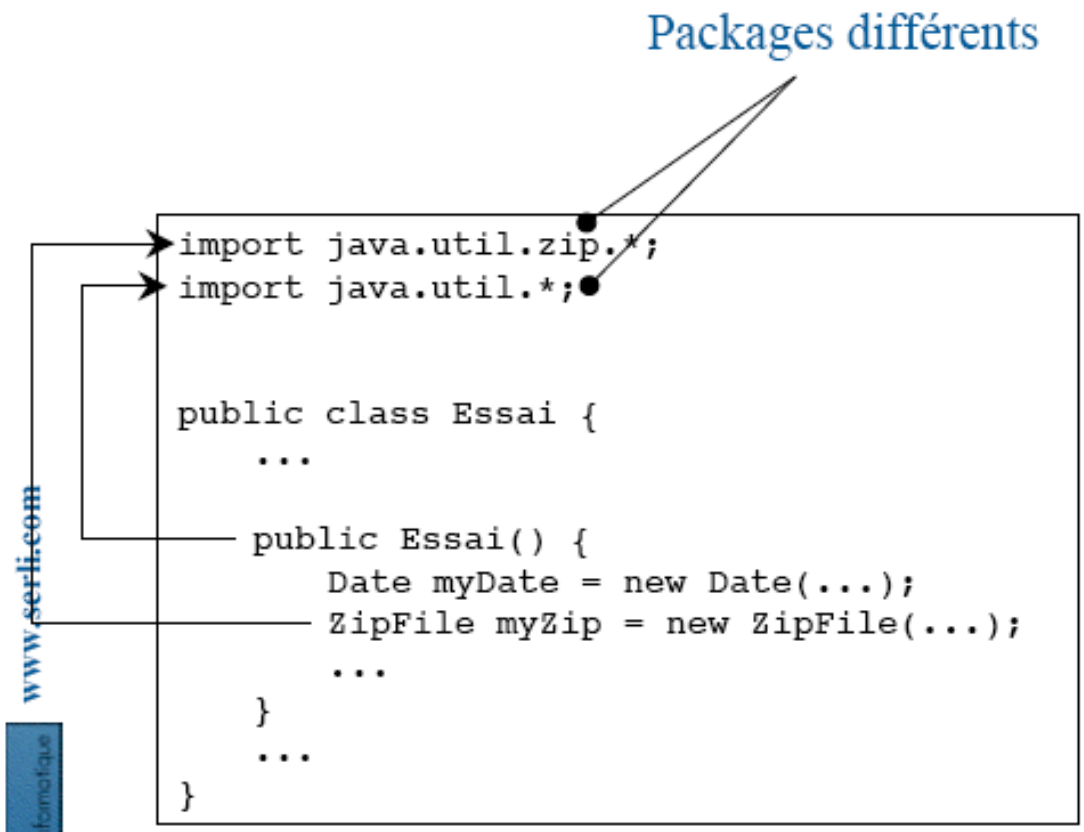
```
java voiture.test.Test
```

La séparation entre package, sous-packages et classes se fait à l'aide de point « . » et non de anti-slash « \ »

www.serli.com
serli informatique

Les packages : visibilité

- L'instruction import *nomPackage.** ne concerne que les classes du package indiqué. Elle ne s'applique pas aux classes des sous-classes



Essai utilise les classes *Date* du package *java.util* et *ZipFile* du package *java.util.zip*

www.serli.com

Les flux

- Pour obtenir des données, un programme ouvre un flux de données sur une source de données (fichier, clavier, mémoire, etc)
- De la même façon pour écrire des données dans un fichier, un programme ouvre un flux de données
- Java fournit un paquetage *java.io* qui permet de gérer les flux de données en entrée et en sortie, sous forme de caractères (exemple fichiers textes) ou sous forme binaire (octets, byte)

www.serli.com
serli informatique

Les flux

- En Java, le nombre de classes intervenant dans la manipulation des flux est important (plus de 50)
- Java fournit quatre hiérarchies de classes pour gérer les flux de données
  - Pour les flux binaires :
    - La classe *InputStream* et ses sous-classes pour lire des octets (*FileInputStream*)
    - La classe *OuputStream* et ses sous-classes pour écrire des octets (*FileOuputStream*)
  - Pour les flux de caractères :
    - La classe *Reader* et ses sous-classes pour lire des caractères (*BufferedReader, FileReader*)
    - La classe *Writer* et ses sous-classes (*BufferedWriter, FileWriter*)

www.serli.com
serli informatique

Les flux de caractères

- Exemple : écrire du texte dans un fichier

FileWriter hérite de Writer et
permet de manipuler un flux texte
Associé à un fichier

```
public class TestIO {
    public static void main(String[] argv) {
        FileWriter myFile = new FileWriter("a_ecrire.txt");

        myFile.write("Voilà ma première ligne dans un fichier");

        myFile.close();
    }
}
```

Fermeture du flux
myFile vers le fichier
a_ecrire.txt

Ecriture d'une ligne de
texte dans le fichier
« a_ecrire.txt »

www.serli.com
serli informatique

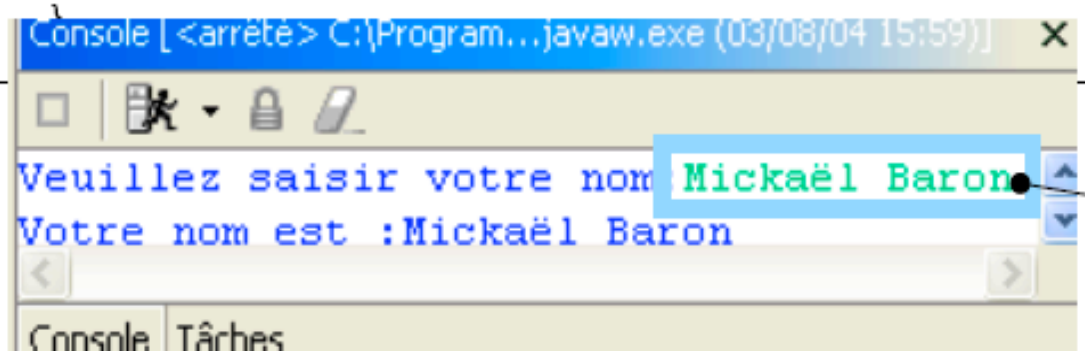
Les flux de caractères

- Exemple : lire l'entrée standard : enfin !!!

« Convertit » un objet de type *InputStream* en *Reader*

```
public class TestIO {
    public static void main(String[] argv) {
        System.out.println("Veuillez saisir votre nom :");

        String inputLine = " ";
        try {
            BufferedReader is = new BufferedReader(new InputStreamReader(System.in));
            String inputLine;
            inputLine = is.readLine();

            is.close();
        } catch (Exception e) {
            System.out.println("Intercepté : " + e);
        }

        if (inputLine != null)
            System.out.println("Votre nom est :" + inputLine);
    }
}
```

Lit la ligne jusqu'au prochain retour chariot



Chaîne saisie

www.serli.com

Les flux de caractères

- Exemple : copie de fichier en utilisant les caractères

FileReader et *FileWriter* héritent de *Reader* et *Writer* et permettent de manipuler un flux texte associé à un fichier texte

```
public class TestIO {
    public static void main(String[] argv) {
        FileReader in = new FileReader("a_lire.txt");
        FileWriter out = new FileWriter("a_ecrire.txt");
        int c;

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

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

Transfert de données jusqu'à ce que *in* ne fournisse plus rien

Fermeture des flux et par conséquent des fichiers respectifs

www.serli.com

Les flux binaires

- Exemple : copie de fichier en utilisant les binaires

Même raisonnement que pour les caractères sauf …

```
public class TestIO {
   public static void main(String[] argv) {
      FileInputStream in = new FileInputStream("a_lire.txt");
      FileOutputStream out = new FileOutputStream("a_ecrire.txt");
      int c;

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

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

www.serli.com
Serli informatique

Typologie des classes de java.io

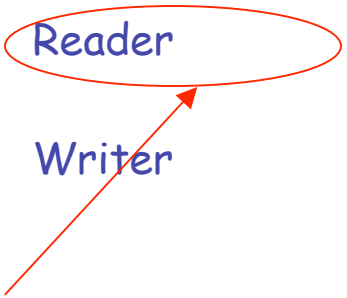
Le nom des classes se décompose en un préfixe et un suffixe.
Il y a quatre suffixes possibles en fonction du type de flux
(flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Typologie des classes de java.io

Le nom des classes se décompose en un préfixe et un suffixe.
Il y a quatre suffixes possibles en fonction du type de flux
(flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

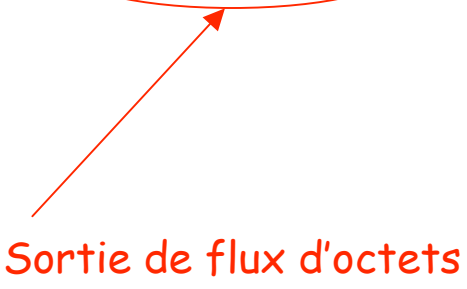


Entrée de flux de caractères

Typologie des classes de java.io

Le nom des classes se décompose en un préfixe et un suffixe.
Il y a quatre suffixes possibles en fonction du type de flux
(flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

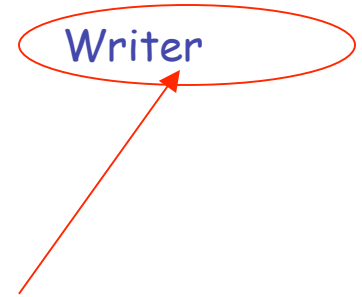


Sortie de flux d'octets

Typologie des classes de java.io

Le nom des classes se décompose en un préfixe et un suffixe.
Il y a quatre suffixes possibles en fonction du type de flux
(flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer



Sortie de flux de caractères

Typologie des classes de java.io

Le nom des classes se décompose en un préfixe et un suffixe.
Il y a quatre suffixes possibles en fonction du type de flux
(flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Il existe donc quatre hiérarchies de classes qui encapsulent des types de flux particuliers.
Ces classes peuvent être séparées en deux séries de deux catégories différentes :

- les classes de lecture et d'écriture
- les classes permettant la lecture/écriture de caractères ou d'octets.

Typologie des classes de java.io

Le nom des classes se décompose en un préfixe et un suffixe.
Il y a quatre suffixes possibles en fonction du type de flux
(flux d'octets ou de caractères) et du sens du flux (entrée ou sortie).

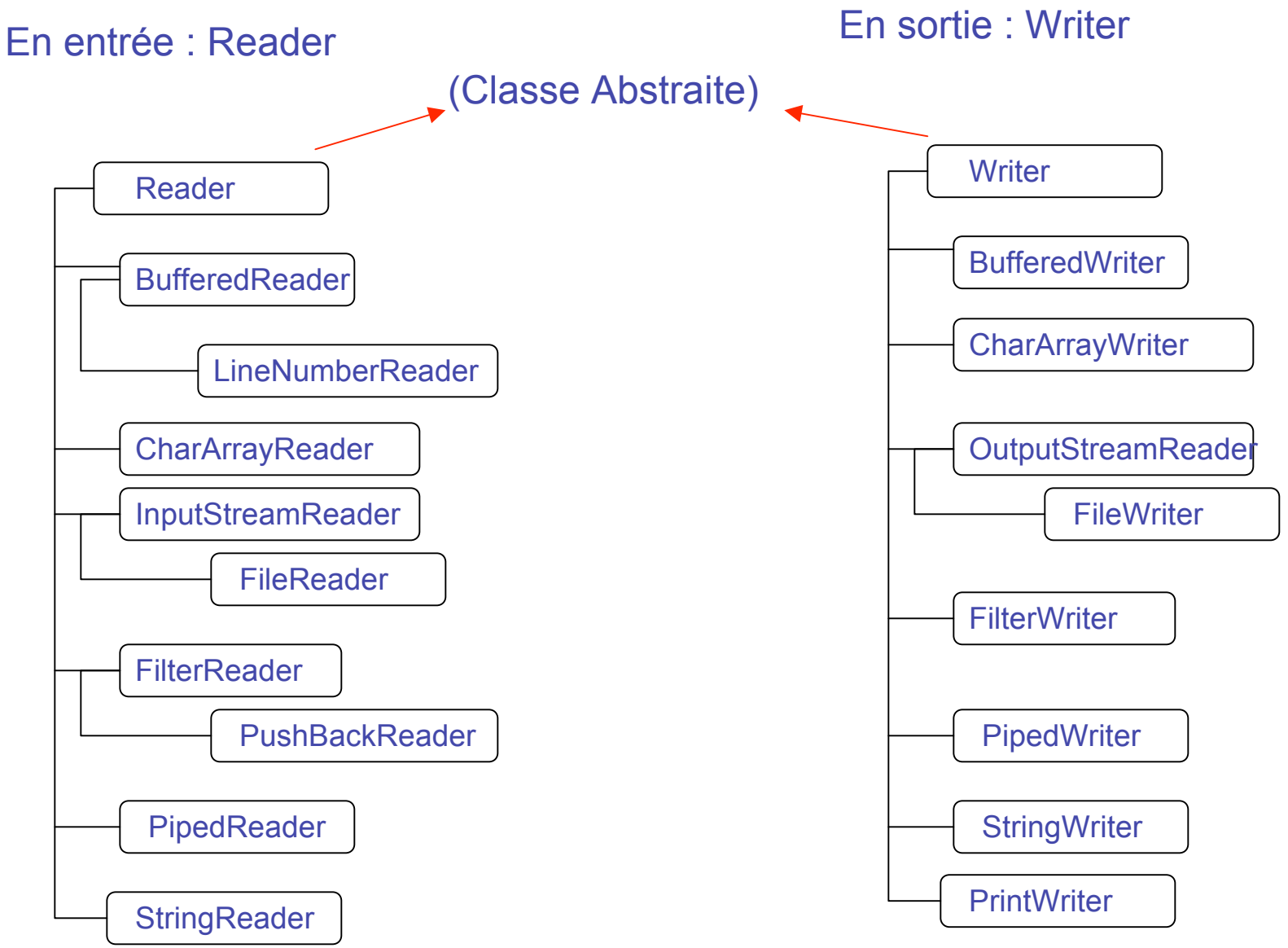
	Flux d'octets	Flux de caractères
Flux d'entrée	InputStream	Reader
Flux de sortie	OutputStream	Writer

Il existe donc quatre hiérarchies de classes qui encapsulent des types de flux particuliers.
Ces classes peuvent être séparéesen deux séries de deux catégories différentes :

- les classes de lecture et d'écriture
- les classes permettant la lecture/écriture de caractères ou d'octets.

- les sous classes de Reader : permettent la lecture sur des ensembles de caractères
- les sous classes de Writer : permettent l'écriture sur des ensembles de caractères
- les sous classes de InputStream : permettent la lecture sur des ensembles d'octets
- les sous classes de OutputStream: permettent l'écriture sur des ensembles d'octets

Les flux de caractères



La classe Reader

C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en lecture.

Méthodes

- *boolean markSupported()* : indique si le flux supporte la possibilité de marquer des positions
- *boolean ready():* indique si le flux est prêt à être lu
- *close()* : ferme le flux et libère les ressources qui lui étaient associées
- *int read()* : renvoie le caractère lu ou -1 si la fin du flux est atteinte.
- *int read(char[])* : lire plusieurs caractères et les mettre dans un tableau de caractères
- *int read(char[], int, int)* : lire plusieurs caractères. Elle attend en paramètre : un tableau de caractères qui contiendra les caractères lus, l'indice du premier élément du tableau qui recevra le premier caractère et le nombre de caractères à lire. Elle renvoie le nombre de caractères lus ou -1 si aucun caractère n'a été lu. La tableau de caractères contient les caractères lus.
- *Long skip(long)*: saute autant de caractères dans le flux que la valeur fournie en paramètre. Elle renvoie le nombre de caractères sautés.
- *mark()* :permet de marquer une position dans le flux
- *reset()* : retourne dans le flux à la dernière position marquée

La classe Writer

C'est une classe abstraite qui est la classe mère de toutes les classes qui gèrent des flux de caractères en sortie.

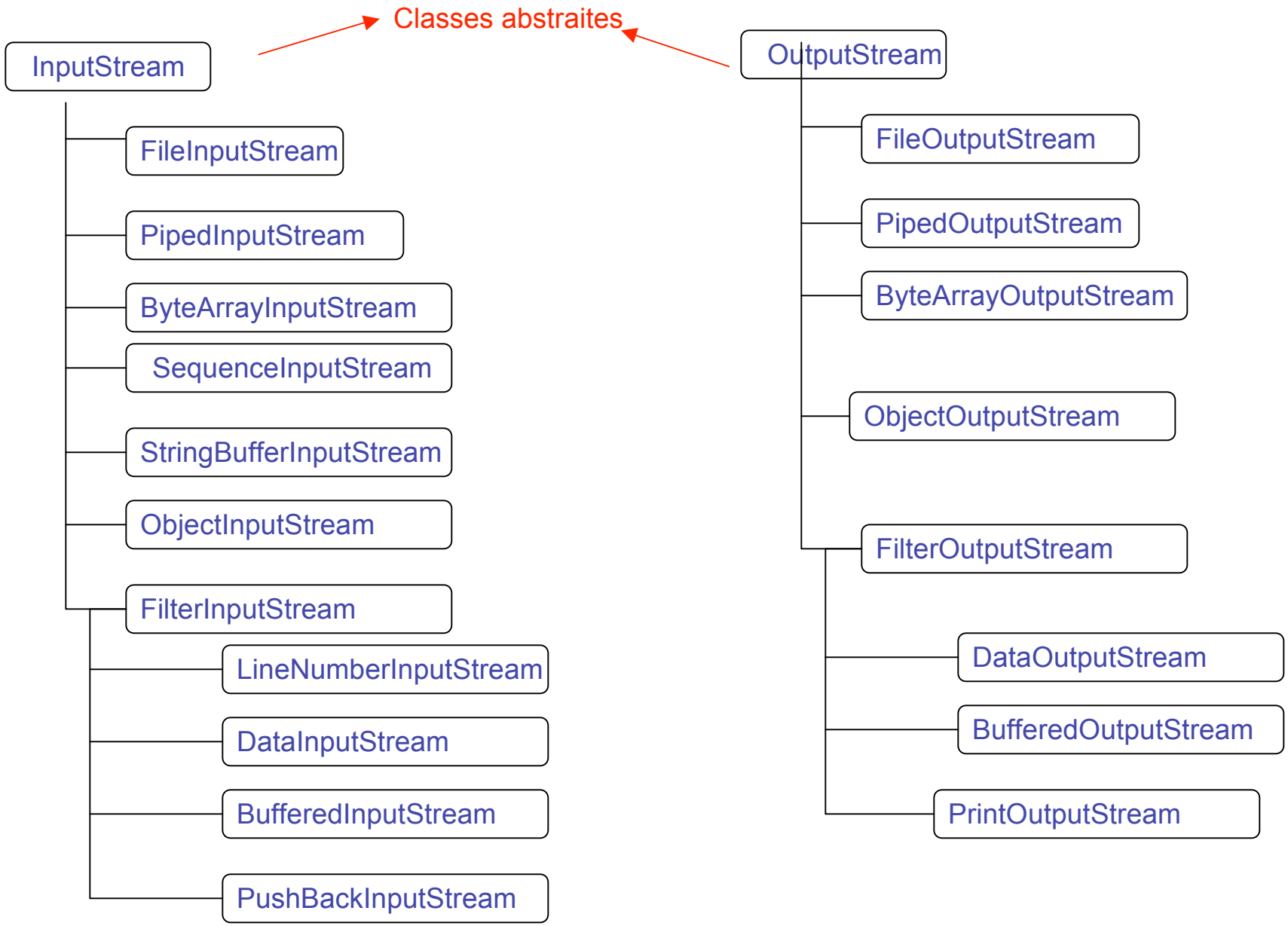
Méthodes

- *close()* : ferme le flux et libère les ressources qui lui étaient associées
- *write(int):* écrire le caractère en paramètre dans le flux.
- *write(char[]):* écrire le tableau de caractères en paramètre dans le flux.
- *write(char[], int, int)* : écrire plusieurs caractères. Elle attend en paramètres : un tableau de caractères, l'indice du premier caractère dans le tableau à écrire et le nombre de caractères à écrire.
- *write(String)* : écrire la chaîne de caractères en paramètre dans le flux
- *write(String, int, int)* : écrire une portion d'une chaîne de caractères. Elle attend en paramètre : une chaîne de caractères, l'indice du premier caractère dans la chaîne à écrire et le nombre de caractères à écrire.

Les flux d'octets

Ils transportent des données sous forme d'octets.
Les flux de ce type sont capables de traiter toutes les données.

Les classes qui gèrent les flux d'octets héritent d'une des deux classes abstraites InputStream ou OutputStream.
Il existe de nombreuses sous classes pour traiter les flux d'octets

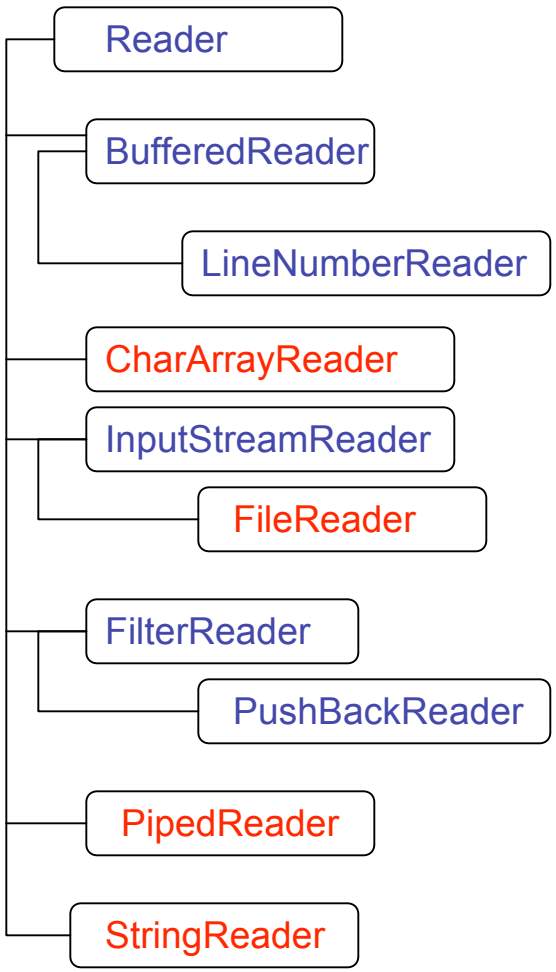

Classes abstraites
InputStream
FileInputStream
PipedInputStream
ByteArrayInputStream
SequenceInputStream
StringBufferInputStream
ObjectInputStream
FilterInputStream
LineNumberInputStream
DataInputStream
BufferedInputStream
PushBackInputStream
OutputStream
FileOutputStream
PipedOutputStream
ByteArrayOutputStream
ObjectOutputStream
FilterOutputStream
DataOutputStream
BufferedOutputStream
PrintOutputStream

Pour le préfixe, il faut distinguer les flux et les filtres.
Pour les flux, le préfixe contient la source/destination selon le sens du flux.

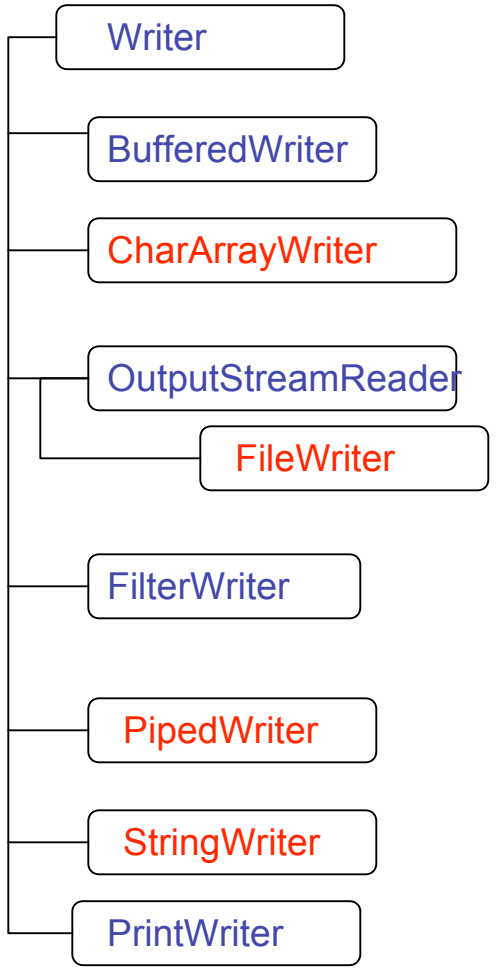
Préfixe du flux	source ou destination du flux
ByteArray	tableau d'octets en mémoire
CharArray	tableau de caractères en mémoire
File	fichier
Object	objet
Pipe	pipeline entre deux threads
String	chaîne de caractères

Les flux de caractères

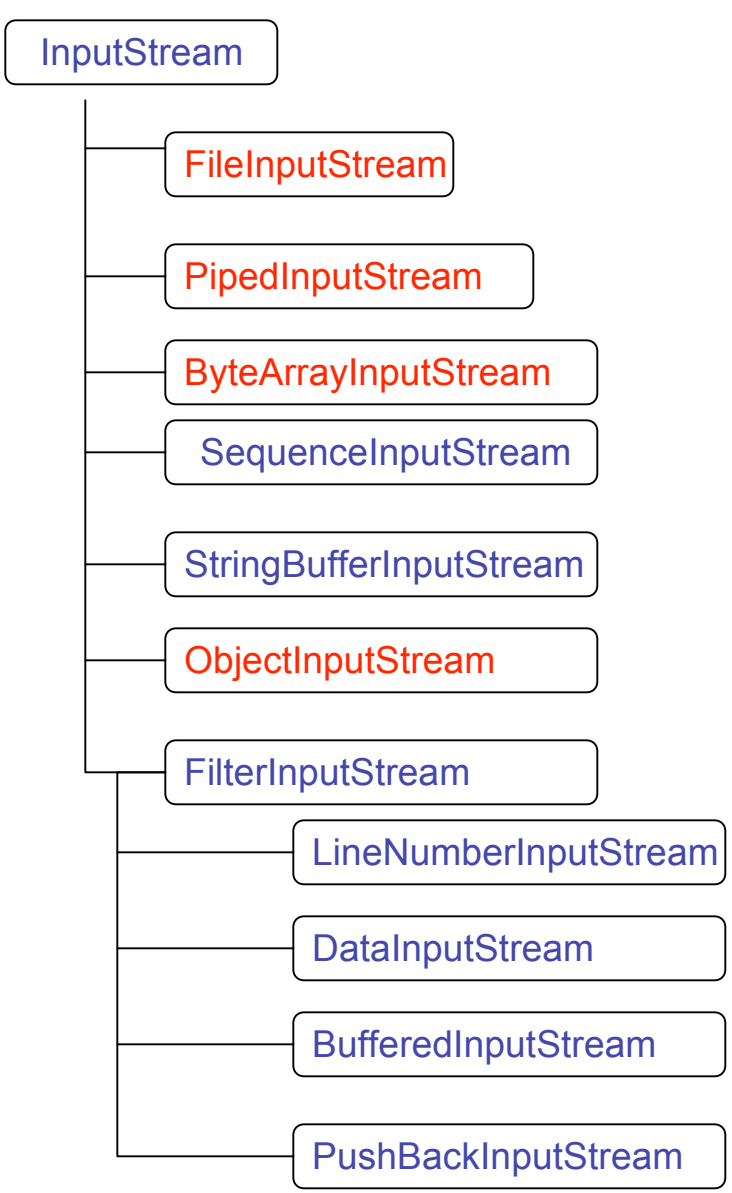
En entrée : Reader

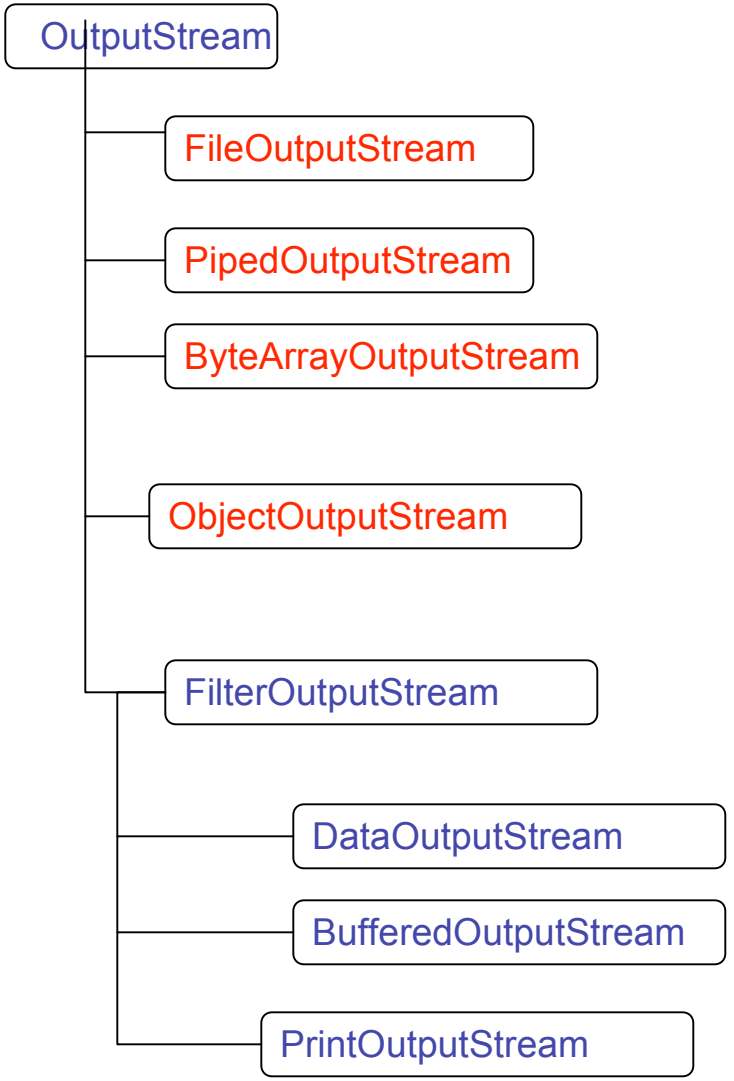


En sortie : Writer



Les flux d'octets



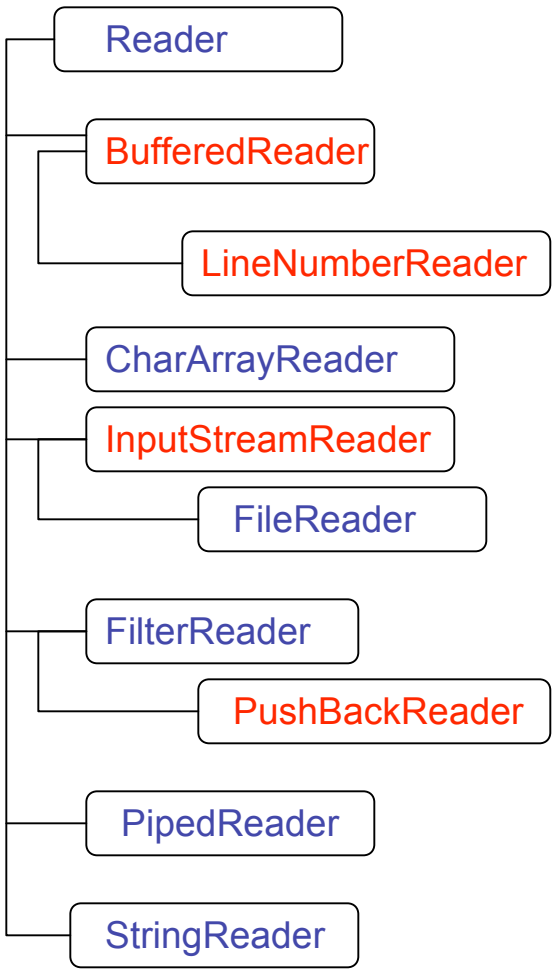


Pour les filtres, le préfixe contient le type de traitement qu'il effectue.
Les filtres n'existent pas obligatoirement pour des flux en entrée et en sortie

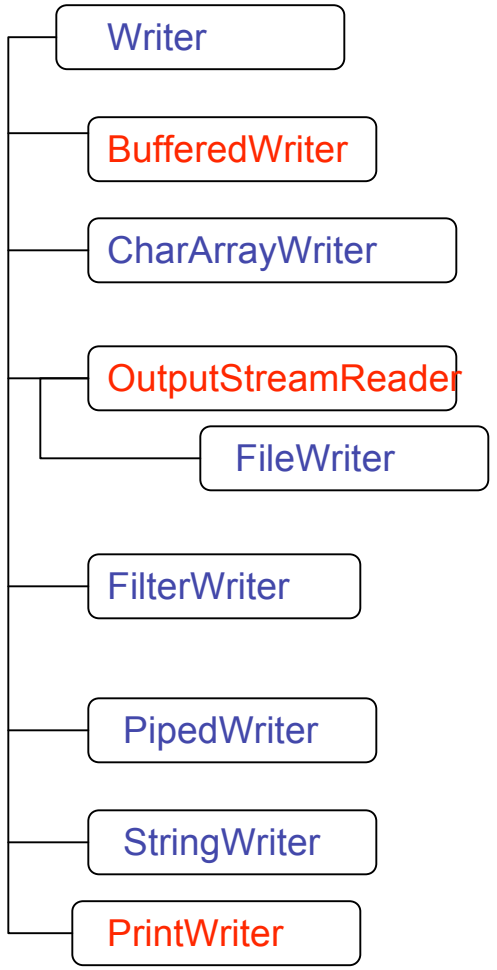
Type de traitement	Préfixe de la classe	En entrée	En sortie
Mise en tampon	Buffered	Oui	Oui
Concaténation de flux	Sequence	Oui (octets)	Non
Conversion de données	Data	Oui (octets)	Oui (octets)
Numérotation des lignes	LineNumber	Oui (caractères)	Non
Lecture avec remise dans le flux des données	PushBack	Oui	Non
Impression	Print	Non	Oui
Sérialisation	Object	Oui (octets)	Oui (octets)
Conversion octets/caractères	InputStream /OutputStream	Oui (octets)	Oui (octets)

Les flux de caractères

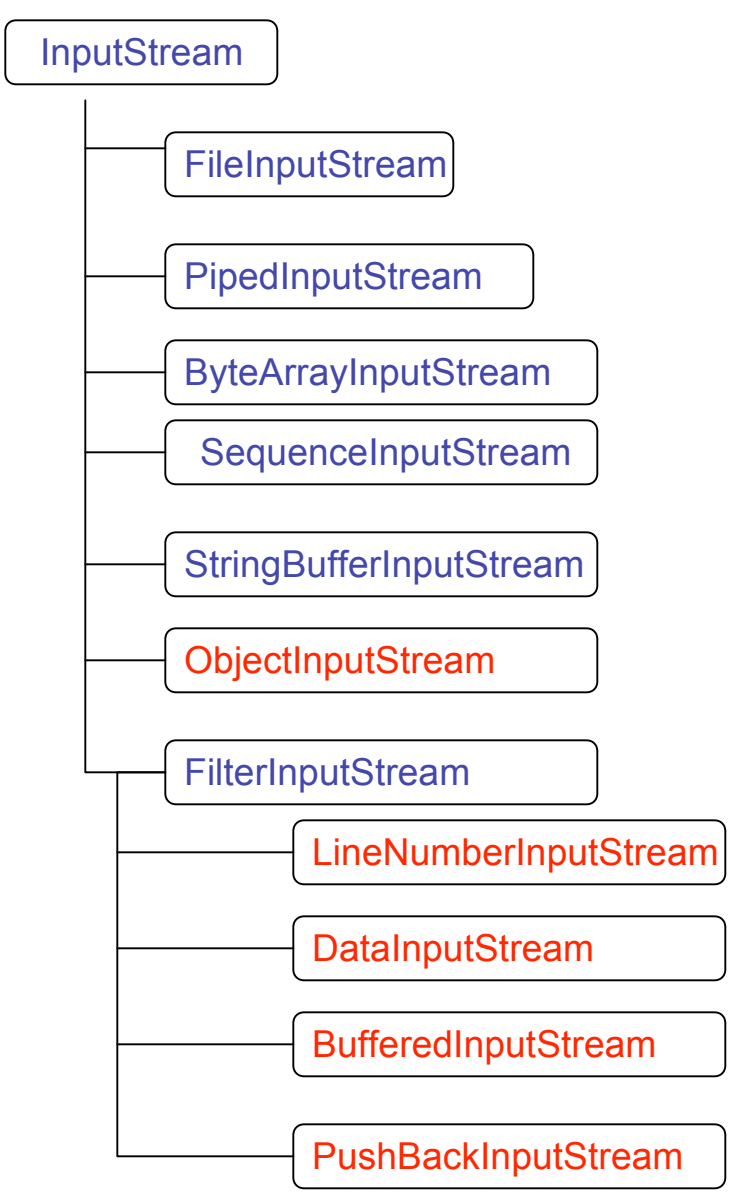
En entrée : Reader

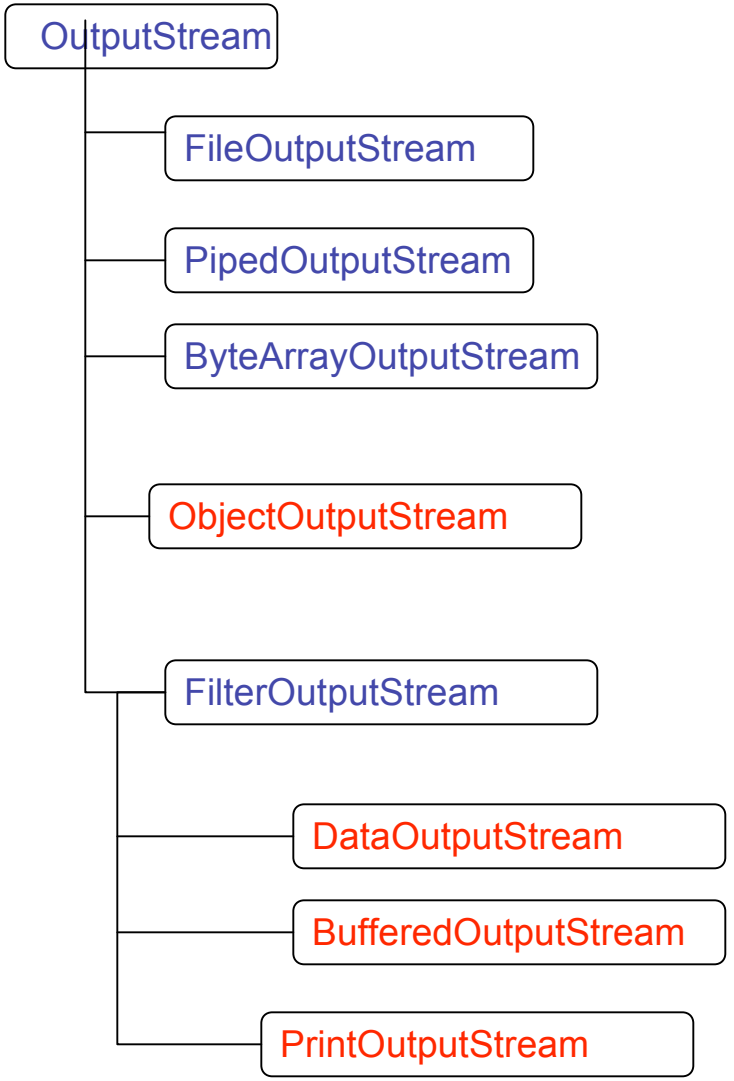


En sortie : Writer



Les flux d'octets





• Buffered : ce type de filtre permet de mettre les données du flux dans un tampon. Il peut être utilisé en entrée et en sortie

• Sequence : ce filtre permet de fusionner plusieurs flux.

• Data : ce type de flux permet de traiter les octets sous forme de type de données

• LineNumber : ce filtre permet de numéroter les lignes contenues dans le flux

• PushBack : ce filtre permet de remettre des données lues dans le flux

• Print : ce filtre permet de réaliser des impressions formatées

• Object : ce filtre est utilisé par la sérialisation

• InputStream / OuputStream : ce filtre permet de convertir des octets en caractères

Les flux de caractères avec un fichier

Les classes FileReader et FileWriter permettent de gérer des flux de caractères avec des fichiers

Les flux de caractères en lecture sur un fichier

Il faut instancier un objet de la classe *FileReader*.
Cette classe hérite de la classe *InputStreamReader* et possède plusieurs constructeurs qui peuvent tous lever une exception de type *FileNotFoundException*

Constructeur

- FileReader(String)
  Créer un flux caractère en lecture vers le fichier dont le nom est précisé en paramètre.

Exemple :

```
FileReader fichier = new FileReader(«monfichier.txt»);
```

- FileReader(File)
  Idem mais le fichier est précisé avec un objet de type File

Les flux de caractères avec un fichier

Il existe plusieurs méthodes de la classe *FileReader* qui permettent de lire un ou plusieurs caractères dans le flux.
Toutes ces méthodes sont héritées de la classe *Reader* et peuvent toutes lever l'exception *IOException*.

Une fois les traitements sur le flux terminés,
il faut libérer les ressources qui lui sont allouées en utilisant la méthode *close()*.

Les flux de caractères en écriture sur un fichier

Il faut instancier un objet de la classe *FileWriter* qui hérite de la classe *OuputStreamWriter*.
Cette classe possède plusieurs constructeurs :

Constructeurs

- FileWriter(String)

  Si le nom du fichier précisé n'existe pas alors le fichier sera créé.
  Si il existe et qu'il contient des données celles ci seront écrasées.

  ```
  Ex:  FileWriter fichier = new FileWriter («monfichier.dat»);
  ```

- FileWriter(File)

  Idem mais le fichier est précisé avec un objet de la classe File

- FileWriter(String, boolean)

  Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true)
  ou écraseront les données existantes (valeur false)

Les flux de caractères en écriture sur un fichier

Il existe plusieurs méthodes de la classe *FileWriter* héritées de la classe *Writer* qui permettent d'écrire un ou plusieurs caractères dans le flux.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode *close()*

Les flux de caractères bufferisés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble de caractères représentant une ligne plutôt que de traiter les données caractères par caractères.
Le nombre d'opérations est ainsi réduit.

Les classes *BufferedReader* et *BufferedWriter* permettent de gérer des flux de caractères tamponnés (bufferisés) avec des fichiers

Les flux de caractères bufferisés avec un fichier.

Pour améliorer les performances des flux sur un fichier, la mise en tampon des données lues ou écrites permet de traiter un ensemble de caractères représentant une ligne plutôt que de traiter les données caractères par caractères.
Le nombre d'opérations est ainsi réduit.

Les classes BufferedReader et BufferedWriter permettent de gérer des flux de caractères tamponnés avec des fichiers

Les flux de caractères bufferisés en lecture avec un fichier

Il faut instancier un objet de la classe BufferedReader. Cette classe possède plusieurs constructeurs qui peuvent tous lever une exception de type FileNotFoundException:

Constructeurs

BufferedReader(Reader) le paramètre fourni doit correspondre au flux à lire.

Ex:

```
BufferedReader fichier  =
    new BufferedReader(new FileReader("monfichier.txt")
```

BufferedReader(Reader, int) l'entier en paramètre permet de préciser la taille du buffer. Il doit être positif sinon une exception de type *IllegalArgumentException* est levée.

Il existe plusieurs méthodes de la classe *BufferedReader* héritées de la classe *Reader* qui permettent de lire un ou plusieurs caractères dans le flux.
Toutes ces méthodes peuvent lever une exception de type IOException.
Elle définit une méthode supplémentaire pour la lecture :

Méthodes

- String readLine()
  lire une ligne de caractères dans le flux. Une ligne est une suite de caractères qui se termine par un retour chariot '\r' ou un saut de ligne '\n' ou les deux.

- La classe BufferedReader possède plusieurs méthodes pour gérer le flux hérité de la classe Reader.

- Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Exemple

```
import java.io.*;

public class TestBufferedReader {
  protected String source;

  public TestBufferedReader(String source) {
    this.source = source;
    lecture();
  }

  public static void main(String args[]) {
    new TestBufferedReader("source.txt");
  }

  private void lecture() {
    try {
      String ligne ;
      BufferedReader fichier = new BufferedReader(new FileReader(source));

      while ((ligne = fichier.readLine()) != null) {
          System.out.println(ligne);
      }

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Les flux de caractères bufferisés en écriture avec un fichier

Il faut instancier un objet de la classe *BufferedWriter*.
Cette classe possède plusieurs constructeurs :

Constructeurs

- BufferedWriter(Writer)

  le paramètre fourni doit correspondre au flux dans lequel les données sont écrites.

  Exemple

```
BufferedWriter fichier =
        new BufferedWriter(new FileWriter(«monfichier.txt»))
```

- BufferedWriter(Writer, int)

  l'entier en paramètre permet de préciser la taille du buffer.
  Il doit être positif sinon une exception IllegalArgumentException est levée.

Les flux de caractères bufferisés en écriture avec un fichier

Il existe plusieurs méthodes de la classe *BufferedWriter* héritées de la classe *Writer* qui permettent de lire un ou plusieurs caractères dans le flux.

La classe BufferedWriter possède plusieurs méthodes pour gérer le flux :

Méthode	Rôle
• flush()	vide le tampon en écrivant les données dans le flux.
• newLine()	écrire un séparateur de ligne dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode *close()*.

Exemple

```
import java.io.*;
import java.util.*;

public class TestBufferedWriter {
  protected String destination;

  public TestBufferedWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestBufferedWriter("print.txt");
  }
  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      BufferedWriter fichier = new BufferedWriter(new FileWriter(destination));

      fichier.write("bonjour tout le monde");
      fichier.newLine();
      fichier.write("Nous sommes le "+ new Date());
      fichier.write(", le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La classe PrintWriter

Cette classe permet d'écrire dans un flux des données formatées.
Cette classe possède plusieurs constructeurs :

Constructeurs

- PrintWriter(Writer)

  Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.

  Exemple:

```
PrintWriter fichier =
        new PrintWriter( new FileWriter(«monfichier.txt »))
```

- PrintWriter(Writer, boolean)

  Le booléen permet de préciser si le tampon doit être automatiquement vidé

- PrintWriter(OutputStream)

  Le paramètre fourni précise le flux. Le tampon est automatiquement vidé.

- PrintWriter(OutputStream, boolean)

  Le booléen permet de préciser si le tampon doit être automatiquement vidé

Il existe de nombreuses méthodes de la classe *PrintWriter* qui permettent d'écrire un ou plusieurs caractères dans le flux en les formatant. Les méthodes *write()* sont héritées de la classe *Writer*.

Elle définit plusieurs méthodes pour envoyer des données formatées dans le flux :

- *print(...)*
  Plusieurs méthodes *print* acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux

- println()
  Cette méthode permet de terminer la ligne courante dans le flux en y écrivant un saut de ligne.

- println (...)
  Plusieurs méthodes *println* acceptent des données de différents types pour les convertir en caractères et les écrire dans le flux avec une fin de ligne.

- La classe *PrintWriter* possède plusieurs méthodes pour gérer le flux :
  - *flush()* Vide le tampon en écrivant les données dans le flux.es dans le flux

Exemple :

```
import java.io.*;
import java.util.*;

public class TestPrintWriter {
  protected String destination;

  public TestPrintWriter(String destination) {
    this.destination = destination;
    traitement();
  }

  public static void main(String args[]) {
    new TestPrintWriter("print.txt");
  }

  private void traitement() {
    try {
      String ligne ;
      int nombre = 123;
      PrintWriter fichier = new PrintWriter(new FileWriter(destination));

      fichier.println("bonjour tout le monde");
      fichier.println("Nous sommes le "+ new Date());
      fichier.println("le nombre magique est " + nombre);

      fichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Les flux d'octets avec un fichier

Les classes FileInputStream et FileOutputStream permettent de gérer des flux d'octets avec des fichiers.

Les flux d'octets en lecture sur un fichier

Il faut instancier un objet de la classe FileInputStream. Cette classe possède plusieurs constructeurs qui peuvent tous lever l'exception FileNotFoundException:

Constructeurs

- FileInputStream(String)

  Ouvre un flux en lecture sur le fichier dont le nom est donné en paramètre

  Exemple :

```
FileInputStream fichier = new FileInputStream(«monfichier.dat»);
```

- FileInputStream(File)

  Idem mais le fichier est précisé avec un objet de type F

Les flux d'octets avec un fichier

Les classes FileInputStream et FileOutputStream permettent de gérer des flux d'octets avec des fichiers.

Les flux d'octets en lecture sur un fichier

Il faut instancier un objet de la classe FileInputStream. Cette classe possède plusieurs constructeurs qui peuvent tous lever l'exception FileNotFoundException:

Constructeurs

- FileInputStream(String)

  Ouvre un flux en lecture sur le fichier dont le nom est donné en paramètre

  Exemple

```
FileInputStream fichier = new FileInputStream(«monfichier.dat»);
```

- FileInputStream(File)

  Idem mais le fichier est précisé avec un objet de type F

Les flux d'octets avec un fichier

Les flux d'octets en lecture sur un fichier

Il existe plusieurs méthodes de la classe FileInputStream qui permettent de lire un ou plusieurs octets dans le flux. Toutes ces méthodes peuvent lever l'exception IOException.

• int read()

Cette méthode envoie la valeur de l'octet lu ou -1 si la fin du flux est atteinte.

Exemple

```
int octet = 0;
while (octet != 1 ) {
   octet = fichier.read();
}
fichier.read();
}
```

• int read(byte[], int, int)

Cette méthode lit plusieurs octets. Elle attend en paramètre :
un tableau d'octets qui contiendra les octets lus, l'indice du premier élément du tableau qui recevra le premier octet et le nombre d'octets à lire.

Elle renvoie le nombre d'octets lus ou -1 si aucun octet n'a été lus. Le tableau d'octets contient les octets lus.

Les flux d'octets avec un fichier

Les flux d'octets en lecture sur un fichier

La classe *FileInputStream* possède plusieurs méthodes pour gérer le flux :

Méthode

- long skip(long)
  saute autant d'octets dans le flux que la valeur fournie en paramètre.
  Elle renvoie le nombre d'octets sautés.
- close()
  ferme le flux et libère les ressources qui lui étaient associées
- int available()
  retourne le nombre d'octets qu'il est encore possible de lire dans le flux

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().

Les flux d'octets en écriture sur un fichier

Il faut instancier un objet de la classe *FileOutputStream*
Cette classe possède plusieurs constructeurs :

Constructeurs

- FileOutputStream(String)

  Si le fichier précisé n'existe pas, il sera créé. Si il existe et qu'il contient des données celles ci seront écrasées.

  Exemple :

```
FileOuputStream fichier =
        new FileOutputStream(«monfichier.dat»)
```

- FileOutputStream(String, boolean)

  Le booléen permet de préciser si les données seront ajoutées au fichier (valeur true) ou écraseront les données existantes (valeur false)

Les flux d'octets en écriture sur un fichier

Il existe plusieurs méthodes de la classe *FileOutputStream* qui permettent de lire un ou plusieurs octets dans le flux.

- write(int) : Cette méthode écrit l'octet en paramètre dans le flux.

- write(byte[]) : Cette méthode écrit plusieurs octets.
  Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire : tous les éléments du tableau sont écrits.

- write(byte[], int, int) : Cette méthode écrit plusieurs octets.
  Elle attend en paramètre : un tableau d'octets qui contient les octets à écrire, l'indice du premier éléments du tableau d'octets à écrire et le nombre d'octets à écrire.

Une fois les traitements sur le flux terminés, il faut libérer les ressources qui lui sont allouées en utilisant la méthode close().méthode close()

Exemple :

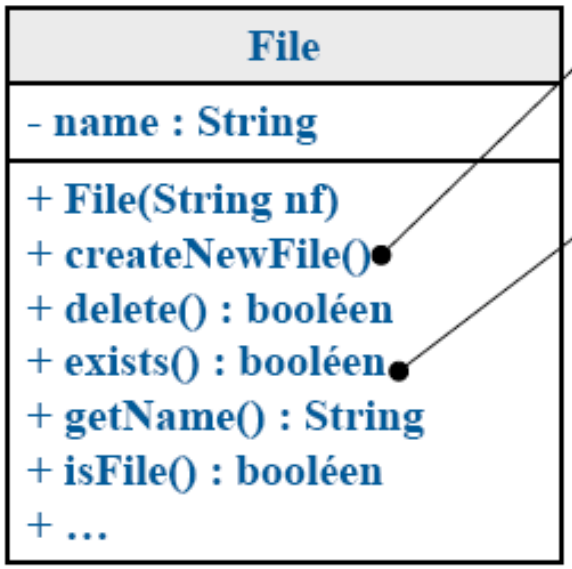
```
import java.io.*;

public class CopieFichier {
  protected String source;
  protected String destination;

  public CopieFichier(String source, String destination) {
    this.source = source;
    this.destination = destination;
    copie();
  }

  public static void main(String args[]) {
    new CopieFichier("source.txt","copie.txt");
  }

  private void copie() {
    try {
      FileInputStream fis = new FileInputStream(source);
      FileOutputStream fos = new FileOutputStream(destination);
      while(fis.available() > 0) fos.write(fis.read());
      fis.close();
      fos.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

La classe File

- Java dispose d'une classe *File* qui offre des fonctionnalités de gestion de fichiers
- La création d'un objet de type *File*

Attention : ne pas confondre la création de l'objet avec la création du fichier physique

```
File monFichier = new File("truc.dat");
```



Création du fichier portant le nom de *name*

Vérifie si le fichier existe physiquement

```
File monFichier = new File("c:\toto.txt");
if (monFichier.exists()) {
    monFichier.delete();
} else {
    monFichier.createNewFile();
}
```

www.serli.com
serli informatique

La classe File

- Les fichiers et les répertoires sont encapsulés dans la classe File du package java.io.

- Il n'existe pas de classe pour traiter les répertoires car ils sont considérés comme des fichiers.

- Une instance de la classe File est une représentation logique d'un fichier ou d'un répertoire qui peut ne pas exister physiquement sur le disque.

- Si le fichier ou le répertoire existe, de nombreuses méthodes de la classe File permettent d'obtenir des informations sur le fichier.

- Sinon plusieurs méthodes permettent de créer des fichiers ou des répertoires.

La classe File

Méthodes

- boolean canRead() — indique si le fichier peut être lu
- boolean canWrite() — indique si le fichier peut être modifié
- boolean createNewFile() — création d'un nouveau fichier vide

- File createTempFile(String, String)
  création d'un nouveau fichier dans le répertoire par défaut des fichiers temporaires.
  Les deux arguments sont le préfixe et le suffixe du fichier.

- File createTempFile(String, String, File)
  création d'un nouveau fichier temporaire.
  Les trois arguments sont le préfixe et le suffixe du fichier et le répertoire.

- boolean delete()
  détruire le fichier ou le repertoire. Le booléen indique le succés de l'opération
- deleteOnExit()
  demande la suppression du fichier à l'arrêt de la JVM
- boolean exists()
  indique si le fichier existe physiquement

La classe File

Méthodes

- String getAbsolutePath() renvoie le chemin absolu du fichier
- String getPath renvoie le chemin du fichier
- boolean isAbsolute() indique si le chemin est absolu
- boolean isDirectory() indique si le fichier est un répertoire
- boolean isFile() indique si l'objet représente un fichier
- long length() renvoie la longueur du fichier
- String[] list() renvoie la liste des fichiers et répertoire contenu dans le répertoire
- boolean mkdir() création du répertoire
- boolean mkdirs() création du répertoire avec création des répertoires manquants dans l'arborescence du chemin
- boolean renameTo() renommer le fichier

La classe File

Méthodes

Depuis la version 1.2 du J.D.K., de nombreuses fonctionnalités ont été ajoutées à cette classe

• la création de fichiers temporaires (createNewFile, createTempFile, deleteOnExit)

• la gestion des attributs "caché" et "lecture seule" (isHidden, isReadOnly)

• des méthodes qui renvoient des objets de type File au lieu de type String (getParentFile, getAbsoluteFile, getCanonicalFile, listFiles)

• une méthode qui renvoie le fichier sous forme d'URL (toURL)

Exemple

```
import java.io.*;

public class TestFile {
  protected String nomFichier;
  protected File fichier;

  public TestFile(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+"n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droite d'ecriture : "+fichier.canWrite());

    if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      String fichiers[] = fichier.list();
      for(int i = 0; i < fichiers.length; i++) System.out.println("  "+fichiers[i]);
    }
  }
}
```

Exemple:

```
import java.io.*;

public class TestFile_12 {
  protected String nomFichier;
  protected File fichier;

  public TestFile_12(String nomFichier) {
    this.nomFichier = nomFichier;
    fichier = new File(nomFichier);
    traitement();
  }

  public static void main(String args[]) {
    new TestFile_12(args[0]);
  }

  private void traitement() {

    if (!fichier.exists()) {
      System.out.println("le fichier "+nomFichier+"n'existe pas");
      System.exit(1);
    }

    System.out.println(" Nom du fichier    : "+fichier.getName());
    System.out.println(" Chemin du fichier : "+fichier.getPath());
    System.out.println(" Chemin absolu     : "+fichier.getAbsolutePath());
    System.out.println(" Droit de lecture  : "+fichier.canRead());
    System.out.println(" Droite d'ecriture : "+fichier.canWrite());
```

```
if (fichier.isDirectory() ) {
      System.out.println(" contenu du repertoire ");
      File fichiers[] = fichier.listFiles();
      for(int i = 0; i < fichiers.length; i++) {

         if (fichiers[i].isDirectory())
           System.out.println("  ["+fichiers[i].getName()+"]");
         else
           System.out.println("  "+fichiers[i].getName());
      }
    }
  }
}
```

Les fichiers à accès direct

- Les fichiers à accès direct permettent un accès rapide à un enregistrement contenu dans un fichier.

- Le plus simple pour utiliser un tel type de fichier est qu'il contienne des enregistrements de taille fixe, mais ce n'est pas obligatoire.

- Il est possible dans un tel type de fichier de mettre à jour directement un de ces enregistrements.

La classe *RamdonAccessFile* encapsule les opérations de lecture/écriture d'un tel fichier.
Elle implémente les interfaces *DataInput* et *DataOutput.*

- Elle possède deux constructeurs qui attendent en paramètre le fichier à utiliser (sous la forme d'un nom de fichier ou d'un objet de type File qui encapsule le fichier) et le mode d'accès. Le mode est une chaîne de caractères qui doit être égal à «r» ou «rw» selon que le mode soit lecture seule ou lecture/écriture.

Constructeurs

- *RamdonAccessFile (String NameFile, String mode)*

  Exemple:

  RamdonAccessFile F= new RamdonAccessFile(« Toto.dat », « r »)

- *RamdonAccessFile (File file, String mode)*

Les fichiers à accès direct

Ces deux constructeurs peuvent lever les exceptions suivantes :

- FileNotFoundException si le fichier n'est pas trouvé
- IllegalArgumentException si le mode n'est pas «r» ou «rw»
- SecurityException si le gestionnaire de sécurité empêche l'accès aux fichiers dans le mode précisé

La classe RandomAccessFile possède de nombreuses méthodes

writeXXX() pour écrire des types primitifs dans le fichier

Les fichiers à accès direct

Exemple :

```
package com.moi.test;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier =
                new RandomAccessFile("monfichier.dat", "rw");
      for (int i = 0; i < 10; i++) {
        monFichier.writeInt(i * 100);
      }
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Les fichiers à accès direct

Cette classe possède aussi de nombreuses classes
readXXX() pour lire des données primitives dans le fichier.

Exemple

```
package com.moi.test;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier = new RandomAccessFile("monfichier.dat", "rw");
      for (int i = 0; i < 10; i++) {
         System.out.println(monFichier.readInt());
      }
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résultat :

```
0
100
200
300
400
500
600
700
800
900
```

Les fichiers à accès direct

Pour naviguer dans le fichier, la classe utilise un pointeur qui indique la position dans le fichier ou les opérations de lecture ou de mise à jour doivent être effectuées.

- La méthode getFilePointer() permet de connaître la position de ce pointeur et la méthode seek() permet de le déplacer.

- La méthode seek() attendant en paramètre un entier long qui représente la position, dans le fichier, précisée en octets. La première position commence à zéro.

Les fichiers à accès direct

Exemple : lecture de la sixème données

```
package com.moi.test;

import java.io.RandomAccessFile;

public class TestRandomAccesFile {
  public static void main(String[] args) {
    try {
      RandomAccessFile monFichier
                = new RandomAccessFile("monfichier.dat", "rw");
//5 représente le sixième enregistement puisque le premier commence à 0
//4 est la taille des données puisqu'elles sont des entiers de type int
//(codé sur 4 octets)
      monFichier.seek(5*4);
      System.out.println(monFichier.readInt());
      monFichier.close();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Résutlat :

```
50
```