**Fichiers et Streams (package java.io)**

1. **I/O Streams**

**“I” signifie “input”**

Un programme ouvre un flux "stream" sur une source d'information (file, memory, socket) et lit l'information séquentiellement.

**“O” signifie “output”**

Un programme peut envoyer l'information à une destination externe en ouvrant un flux à une destination et écrivant l'information séquentiellement.

Le cycle d’utilisation de lecture ou écriture séquentielle d’un flot de données est le suivant :

1) Ouverture du flot.

2) Tant qu’il y a des données à lire (ou à écrire), on lit (ou on écrit) la donnée dans le flot.

3) Fermeture du flot.

**Il existe deux grandes catégories de fichiers : fichiers textes et fichiers binaires:**

* les fichiers texte (ils sont produits à l'aide de tout éditeur de textes standard comme notepad, textPad …).
* les fichiers binaires (tous les autres dont le contenu n'est pas directement lisible à l'aide d'un éditeur standard).

Dans chaque famille, il y a des flots concrets, et des filtres. Les filtres sont des flots ajoutant des fonctionnalités supplémentaires.

***Il y a quatre classes "mères", abstraites, pour traiter les flots de données :***

* les flots de caractères : **Reader/Writer**
  + fichiers de caractères : FileReader/FileWriter
  + formatage : PrintWriter
* les flots binaires (de bytes) : **InputStream/OutputStream**
  + fichiers binaires FileInputStream/FileOutputStream
  + formatage de données: DataInputStream/DataOutputStream
  + fichiers d'objets : "sérialisation"
  + fichiers séquentiels, flot à accès direct : RandomAccessFile.

**Lecture/écriture avec les flots qui utilisent les tampons**

* BufferedReader et BufferedWriter sont utilisés pour la lecture et écriture des caractères.
* BufferedInputStream et BufferedOutputStream sont utilisés pour lire et écrire des octets.
* Pour vider le tampon on utilise la méthode **flush()**. Mais certaines méthodes comme **println** effectuent **flush()** automatiquement.

1. **Flots de caractères**

Ils transportent des données sous forme de caractères : java les gèrent avec le format Unicode.

Les classes qui gèrent les flux de caractères (**FileReader** et **FileWriter**) héritent des classes abstraites **Reader** ou **Writer**.

1. **Lecture/écriture caractère par caractère**

**Classe Writer**

La classe Writer est la classe mère de toutes les classes qui gèrent des flux de caractères en écriture.

|  |  |
| --- | --- |
| **Méthodes** | **Rôles** |
| close() | ferme le flux et libère les ressources |
| write(int) | écrit le caractère en paramètre dans le flux. |
| write(char[ ]) | écrit le tableau de caractères dans le flux. |
| write(char[], int n, int m) | écrit une partie du tableau de caractères, le deuxième arg est l'indice du premier caractère dans le tableau à écrire et le dernier arg représente le nombre de caractères à écrire. |
| write(String) | écrit la chaîne de caractères dans le flux. |
| write(String, int, int) | écrit une portion d'une chaîne de caractères. Le deuxième arg est l'indice du premier caractère dans la chaîne à écrire et le dernier est le nombre de caractères à écrire. |

**Classe FileWriter**

Il faut instancier la classe **FileWriter** qui possède plusieurs constructeurs :

|  |  |
| --- | --- |
| **Constructeur** | **Rôle** |
| FileWriter(String nomfichier) | Si le nom du fichier n'existe pas le fichier sera créé. S'il existe et qu'il contient des données, celles ci seront écrasées. |
| FileWriter(File ptfichier) | Idem mais le fichier est précisé avec un objet de type **File**. |
| FileWriter(String nomfichier, boolean b) | Le booléen précise si les données seront ajoutées au fichier (true) ou écraseront les données existantes (false) |

**FileWriter F = new FileWriter(<*nom de fichier*>);**

**Exemple** : écriture d'un caractère dans un fichier

**public** **static** **void** main(String args []) **throws** IOException

{

**int** c=12;

//File f = new File("aaa.txt");

//f.createNewFile();

//FileWriter out = new FileWriter(f);

FileWriter out = **new** FileWriter("aaa.txt");

out.write(c);

out.flush();

out.close();

}

**La classe Reader**

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

|  |  |
| --- | --- |
| **Méthodes** | **Rôles** |
| 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[]) | lit plusieurs caractères et les met dans un tableau de caractères, renvoie le nbre de caractères lus. |
| int read(char[], int, int) | lit des caractères. Le 2eme arg est l'indice du premier élément du tableau qui recevra le premier caractère et le dernier est le nombre de caractères à lire. renvoie le nombre de caractères lus ou -1 si aucun caractère n'a été lu. Le 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. |

* **Classe FileReader** (*classe pour lire des fichiers de caractères)*

**Exemple**

**public** **static** **void** main(String args []) **throws** IOException {

**int** c;

FileReader in = **new** FileReader("aaa.txt");

//while ((c = in.read()) != -1)

c = in.read();

System.*out*.println(c);

in.close();

}

1. **Lecture / écriture ligne par ligne**

Les classes **BufferedReader** et **BufferedWriter** (ou **PrintWriter**) permettent lire et écrire un fichier texte ligne par ligne au lieu de caractère par caractère.

|  |  |
| --- | --- |
| **Constructeur** | **Rôle** |
| BufferedWriter(Writer) | le paramètre fourni doit correspondre au flux dans lequel les données sont écrites. |
| BufferedWriter(Writer, int) | l'entier en paramètre précise la taille du buffer. Il doit être positif sinon une exception IllegalArgumentException est levée. |

BufferedWriter fichier = new BufferedWriter( new FileWriter(nom\_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.

|  |  |
| --- | --- |
| **Méthode** | **Rôle** |
| flush() | vide le tampon en écrivant les données dans le flux. |
| newLine() | écrit un séparateur de ligne dans le flux |

**Algorithme**

* Instancier un FileWriter :

**FileWriter FW = new FileWriter(<*nom de fichier*>);**

* Instancier un BufferedWriter pour ce FileWriter

**BufferedWriter BW = new BufferedWriter(FW);**

* Tant qu'il y a des lignes à écrire dans le fichier

**BW.write(<chaîne à écrire>);**

**BW.newLine();**

fin tant que

Fermer le BufferedWriter

* **BW.close();**

**Lecture ligne par ligne dans un fichier texte**

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 lever l'exception **IOException**.

**Flux de caractères tamponnés en lecture avec un fichier**

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

|  |  |
| --- | --- |
| **Constructeur** | **Rôle** |
| BufferedReader(Reader) | le paramètre fourni doit correspondre au flux à lire. |
| 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. |

BufferedReader F = new BufferedReader(new FileReader( nom\_fichier)).

La classe BufferedReader définit une méthode supplémentaire pour la lecture, readLine() qui permet de 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.

**Algorithme**

* Instancier un Filereader

Filereader **FR** = new Filereader(<*nom de fichier*>);

* Instancier un BufferedReader pour ce FileReader

BufferedReader **BR** = new BufferedReader(**FR**);

* Sting S=null; // pour récupérer chaque ligne du fichier
* Tant que la fin de fichier n'est pas atteinte :
  + **S = BR**.readLine();
  + if (S!=null)

traiter S;

fin tant que

* Fermer le BufferedReader

**BR**.close();

**Exemple : copier le contenu d'un fichier dans un autre**

String s;

BufferedWriter out = new BufferedWriter(new FileWriter("fichier1.txt"));

s=texte1.getText();

out.write(s);

in = new BufferedReader(new FileReader("fichier1.txt"));

while ((String s = in.readLine()) != null) {

System.out.println(s);

}

1. **Formatage de textes : Classe PrintWriter**

Cette classe permet d'écrire des données formatées dans un flux. Elle ajoute à un flux la possibilité de faire des écritures (sous forme de texte) des types primitifs java, et des chaînes de caractères.

La principale raison d'utiliser PrintWriter est d'accéder aux méthodes printXXX telles que println (). Vous pouvez essentiellement utiliser un PrintWriter pour écrire dans un fichier, comme vous utiliseriez System.out pour écrire dans la console.

Alors que BufferedWriter est un moyen efficace d’écrire dans un fichier (ou autre), car il tamponne les caractères dans la mémoire Java avant (de passer en C ) pour effectuer l’écriture dans le fichier.

|  |  |
| --- | --- |
| **Constructeur** | **Rôle** |
| PrintWriter(Writer) | Le paramètre fourni précise le flux. Le tampon est automatiquement vidé. |
| PrintWriter(Writer, boolean) | Le booléen permet de préciser si le tampon doit être 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 vidé |
| PrintWriter( String nomfich) | Crée un Printwriter associé au fichier de nomfich. |

PrintWriter F = new PrintWriter( new FileWriter("fichier.txt"));

* 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. Cette classe définit plusieurs méthodes pour envoyer des données formatées dans le flux (print, println…)

**Ecriture**

public static void main(String args[]) throws IOException {

String nomfich;

int n;

Scanner sc = new Scanner(System.in);

PrintWriter sortie = new PrintWriter(new FileWriter("toto.txt"));

do {

System.out.print("donner un entier:");

n = sc.nextInt();

if (n != 0) {

sortie.print(n);

sortie.println(" a pour carre"+n\*n);

}

} while (n != 0);

sortie.close();

System.out.println("\*\*\*\*\*\*\* Fin de création \*\*\*\*\*\*\*\*");

}}

**Lecture**

public class Lecftxt1 {

public static void main(String args[]) throws IOException {

String ligne;

BufferedReader entree= new BufferedReader (new FileReader("toto.txt"));

do {

ligne = entree.readLine();

if (ligne != null) System.out.println(ligne);

} while (ligne != null);

entree.close();

}}

1. **Les flux 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**.

* Les classes **FileInputStream** et **FileOutputStream** offrent les primitives de lecture/écriture de byte.
* Les classes **DataInputStream** et **DataOutputStream** permettent la lecture/écriture des types primitifs (int, double, boolean, char, ...) dans leur représentation en byte.

1. **Les flux d'octets avec un fichier**

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

* **Ecriture sur un fichier**

Il faut instancier la classe **FileOutputStream**.

|  |  |
| --- | --- |
| **Constructeur** | **Rôle** |
| FileOutputStream(String) | Si le fichier précisé n'existe pas, il sera créé. S'il existe et qu'il contient des données elles seront écrasées. |
| FileOutputStream(String, boolean) | Le booléen précise si les données seront ajoutées au fichier (true) ou écraseront les données existantes (false) |

**FileOuputStream F = new FileOutputStream("fichier");**

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

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

write(byte[]) : écrit plusieurs octets.

write(byte[], int, int) : Cette méthode écrit une partie du tableau d'octets. Le 2eme arg est l'indice du premier élément du tableau et le dernier est le nombre d'octets à écrire.

**Exemple : écriture des octets dans un fichier**

**public** **static** **void** main(String args []) **throws** IOException {

FileOutputStream out = **new** FileOutputStream("fichier");

**int** c;

byte T []={'a', '1'};

out.write(T);

c = 65;

out.write(c);

}

**Lecture dans un fichier binaire : La classe FileInputStream**

Il existe plusieurs méthodes pour 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();

}

**int read(byte[], int, int)** : Cette méthode lit plusieurs octets. Elle attend en paramètre un tableau d'octets qui recevra 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 lu.

**Exemple**

**try** {

FileInputStream f = **new** FileInputStream("xxx");

**try** {

**for**( **int** a = f.read(); a!=-1; a = f.read())

System.out.print((**char**)a);

}**finally** { f.close();}

} **catch** (FileNotFoundException e) {

*// . . .*

} **catch** (IOException e) {

*// . . .*

}

Ou :

**try** {

FileInputStream f = **new** FileInputStream("xxx");

**try** {

**byte** [] t = **new byte**[100];

int a = f.read( t);

while(a!=-1) {

**for**( **int** i = 0; i< a; ++i) System.out.print((**char**)t[i]);

a = f.read( t);

}

}**finally**{f.close();}

} **catch** (FileNotFoundException e) {

*// . . .*

} **catch** (IOException e) {

*// . . .*

}

1. **Flux d'octets bufférisés avec un fichier : Flots de données**

* **Ecriture**

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

**Algorithme**

* Instancier un **FileOutputStream** :

FileOutputStream F = new FileOutputStream (<fichier à créer>);

* Instancier un **DataOutputStream** pour ce flux fichier :

DataOutputStream D = new DataOutputStream (F);

* Transférer le tableau dans le fichier :

for (i=0 ; i<T.length ; i++ ) D.writeFloat(T[i]);

* Fermer le flux :

F.close();

* **Lecture**

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

**InputStream (available, read, skip, close).**

|  |
| --- |
| **void** read (**byte** [ ] tab) |
| **void** read (**byte** [ ] tab, **int** d, **int** l) |

Toutes ces méthodes peuvent lever une exception EOFException quand on essaie de lire au-delà de la fin du flux, ou une exception IOException.

|  |  |
| --- | --- |
| **void** readFully (**byte** [ ] tab, **int** d, **int** N) | Lit N octets depuis le flux d’entrée, et les range dans tab à partir de d.  Si la fin de fichier est détectée une exception   EOFException est levée. |
| **boolean readBoolean ()** | Lit un octet et retourne *true* si cet octet est différent de 0, et *false* sinon. |
| **byte readByte ()** | ….. |

|  |  |
| --- | --- |
| 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 dans le flux |

**Algorithme**

* Instancier un FileInputStream :

FileInputStream **FI** = new FileInputStream (<fichier à lire>);

* Instancier un DataInputStream pour ce flux fichier :

DataInputStream **DI** = new DataInputStream (**FI**);

* Récupérer la taille en octets du fichier à lire :

int Taille = **DI**.available();

* Calculer la longueur du tableau pour récupérer les données :

int Longueur = Taille / <taille du type>

Longueur = Taille / 4 ... si les données sont des int

Longueur = Taille / 8 ... si les données sont des long

* Préparer le tableau :
* **long** T []= new **long**[Longueur];
* Lire le tableau dans le fichier :

for (i=0 ; i<T.length ; i++ )T[i] = **DI**.readLong();

* Fermer le flux :

**FI**.close();

**Exemple**

public static void main(String[] argv) throws IOException {

**try** {

DataOutputStream dos = **new** DataOutputStream(

**new** FileOutputStream ("fich"));

do{

f=sc.nextInt();

dos.writeInt(f);

}while (…);

} **catch** (FileNotFoundException e) {

...

} **catch** (IOException e) {

...

}

public static void main(String[] argv) throws IOException {

int i;

FileInputStream FI = new FileInputStream (dataFile);

DataInputStream DI = new DataInputStream (FI);

int Taille = DI.available();

int Longueur = Taille / 4;

int[] T= new int[Longueur];

for (i=0 ; i<T.length ; i++ )

T[i] = DI.readInt();

FI.close();

for (i=0 ; i<T.length ; i++ )

System.out.print(T[i]+" ");

}}

**Les flux tamponnés**

**Classe BufferedOutputStream**

La classe BufferedOutputStream permet de représenter un flux avec tampon, qui permet d'obtenir une écriture plus rapide sur un flux.

Constructeurs :

|  |  |
| --- | --- |
| BufferedOutputStream( OutputStream o) | Crée un *BufferedOutputStream* avec un tampon de taille 8192. |
| BufferedOutputStream( OutputStream o, **int** taille) | Crée un *BufferedOutputStream* avec un tampon de taille *taille*. |

Exemple

tempsAvant = System.currentTimeMillis();

**try** {

DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream ("fich"));

**long** tempsAvant = System.currentTimeMillis();

**for**(**int** i = 0; i < 5000; ++i)

dos.writeInt(i);

dos.close();

**long** tempsApres = System.currentTimeMillis();

System.out.println((tempsApres-tempsAvant)/1000.0+"s ") ;

DataOutputStream bdos = **new** DataOutputStream(**new** BufferedOutputStream(**new** FileOutputStream ("x")));

tempsAvant = System.currentTimeMillis();

**for**( **int** i = 0; i < 5000; ++i) bdos.writeInt(i);

bdos.close();

tempsApres = System.currentTimeMillis();

System.out.println((tempsApres-tempsAvant)/1000.0+"s ") ;

} **catch** (FileNotFoundException e) {

...

} **catch** (IOException e) {

...

}

**La classe BufferedInputStream**

*BufferedInputStream* permet de lire sur un flux de données en utilisant une zone mémoire tampon (*buffer*). Cette classe se charge de lire sur le flux de données un grand nombre d'octets qu'elle garde dans un tampon.

Tous les appels à la méthode *read()* ultérieurs sur une instance de cette classe renvoient des données provenant de ce tampon tant qu'il y reste des octets à lire; le tampon est rechargé à partir du fichier quand on a épuisé toutes les informations qui s'y trouvent.

L'utilisation de cette classe accélère la lecture des fichiers.

Constructeurs :

|  |  |
| --- | --- |
| BufferedInputStream (InputStream in) | Le tampon a la taille 8192 par défaut. |
| BufferedInputStream(InputStream in, **int** taille) | Le tampon a la taille *taille* |

Comparer les temps d'exécution des deux séquences suivantes :

InputStream entree = **new** FileInputStream("x");

**for**( **int** c = entree.read(); c !=-1; c = entree.read()) ... ;

entree.close();

InputStream entree = **new** BufferedInputStream(

**new** FileInputStream("x"), 4096);

**for**(**int** c = entree.read(); c !=-1; c = entree.read()) ... ;

entree.close();

1. **Fichiers à accès direct : classe** **RamdonAccessFile**

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 contient des enregistrements de taille fixe. 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**.

**Constructeur** :

RandomAccessFile(nom\_fichier, "mode");

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.

Les constructeurs de cette classe 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 et de nombreuses méthodes **readXXX**() pour lire des données primitives dans le fichier.

**Navigation dans le fichier**

Pour naviguer dans le fichier, la classe utilise un pointeur qui indique la position dans le fichier.

La méthode **getFilePointer**() permet de connaître la position du pointeur

La méthode **seek**() permet de déplacer le pointeur.

**Exemple**

public static void main(String[] args) {

try {

RandomAccessFile Fichier = new RandomAccessFile("fichier.dat", "rw");

for (int i = 0; i < 10; i++)

{

Fichier.writeInt(i\*10 + i);

}

Fichier.close();

} catch (Exception e) { e.printStackTrace(); }

public static void main(String[] args) {

try {

RandomAccessFile Fichier = new RandomAccessFile("fichier.dat", "rw");

//Fichier.seek(2\*4); // position trois, pour les entiers

System.out.println(Fichier.readInt());

Fichier.close();

} catch (Exception e) { e.printStackTrace(); }

* **Sortie formatée**

Utiliser la classe **PrintWriter** pour les sorties formatées de caractères. La classe possède les méthodes **print** et **println**. Pour le formatage à la C utiliser la méthode format System.out.format("The square root of %d is %f", i, r);

* **Lecture à partir de la console**

Pour lire à partir de la console on peut utiliser le flot System.in, mais c’est plus facile avec la classe java.io.**Console**

Console c = System.console();

Les méthodes utiles : Console.readLine et Console.readPassword.

1. **La classe File**

La classe java.io.**File** contient les méthodes permettant de manipuler le type de fichier, l’existence de fichier, les droits d’accès etc.

Une instance de la classe **File** est une représentation logique d'un fichier ou d'un répertoire.

**Principales méthodes**

|  |  |
| --- | --- |
| **Méthode** | **Rôle** |
| 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, File) | création d'un nouveau fichier temporaire. Les trois arguments sont le préfixe, le suffixe du fichier et le répertoire. |
| boolean delete() | détruire le fichier ou le répertoire. Le booléen indique le succès de l'opération |
| deleteOnExit() | demande la suppression du fichier à l'arrêt de la JVM |
| boolean exists() | indique si le fichier existe physiquement |
| 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 répertoires 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 |

public static void main(String args[]) {

String nomFichier = "rep";

File fichier = new File(nomFichier);

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());

fichier.deleteOnExit();

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]);

} }

1. **Sérialisation d’objets**

La clé d'écriture d'un objet est de représenter son état en une forme sérialisée suffisante pour le reconstruire. Ainsi l'opération lecture/écriture des objets est appelée sérialisation d'objets. Elle permet de rendre l'objet persistant, il est mis sous une forme sous laquelle il pourra être reconstitué.

•*Sérialiser un objet* revient à le transformer en une série d’octets.

•Les données deviennent persistantes.

•Pour qu'un objet puisse être sérialisé sa classe doit implémenter l'interface **Serializable** ou hériter d'une classe qui implémente l'interface **Sérializable**. Si l'on tente de sérialiser un objet qui n'implémente pas l'interface **Serializable**, une exception **NotSerializableException** est levée.

•Ces octets peuvent être stockés ou transmis.

•L’objet peut être reconstruit à partir de ces octets.

Deux classes permettent de lire et écrire des objets, java.io.**ObjectInputStream** et java.io.**ObjectOutputStream**.

La sérialisation utilise l'interface **Serializable** et les deux classes **ObjectOutputStream** et **ObjectInputStream**.



**Ecriture**

•créer un OutputStream

•l'envelopper (wrapper) dans un ObjectOutputStream

•appeler writeObject()

**Lecture :**

•créer un InputStream

•l'envelopper dans un ObjectInputStream

•appeler readObject()

•rend une référence sur un objet : faire un cast

Tout attribut est sérialisé si :

* Il est de type primitif (int, char, boolean, ….) ou c'est une référence dont le type est un type **sérialisable** (dans ce cas l'objet référencé est sérialisé)
* Il n'est pas déclaré **static**.
* Il n'est pas hérité d'une classe mère sauf si celle-ci est elle-même **sérialisable**.
* **L'interface Serializable**

Cette interface ne définit aucune méthode mais permet simplement de savoir qu'une classe peut être sérialisée.

* **ObjectOutputStream**

public class point\_fichiers{

public static void main(String [] args) throws Exception {

point p = new point(20,200);

FileOutputStream fichier = new FileOutputStream("fich\_point.dat");

ObjectOutputStream sortie = new ObjectOutputStream(fichier);

sortie.writeObject(p);

sortie.close();

}}

* **ObjectInputStream**

public class point\_fichiers\_L{

public static void main(String [] args) throws Exception {

FileInputStream fichier = new FileInputStream("fich\_point.dat");

ObjectInputStream entree = new ObjectInputStream(fichier);

point p = (point) entree.readObject();

entree.close();

p.affiche();

}}

**Le mot clé transient**

Le mot clé transient permet d'interdire la sérialisation d'un attribut d'une classe. Il est utilisé pour les données "sensibles" telles que les mots de passe.

En effet, le contenu des attributs est visible dans le flux où l'objet est sérialisé. Pour tout objet ayant accès au flux, il est possible de voir le contenu des attributs même s'ils sont private. Ceci peut poser des problèmes de sécurité surtout si les données sont sensibles.

Java introduit le mot clé transient qui précise que l'attribut qu'il qualifie ne doit pas être inclus dans un processus de sérialisation et donc de désérialisation.

**Exemple :** private transient String codeSecret;

Lors de la désérialisation, les champs transient sont initialisés avec la valeur null. Ceci peut poser des problèmes de type NullPointerException.

**La sérialisation et l'héritage**

Comment se comporte la sérialisation face à l'héritage.

* **Si la classe hérite d'une classe sérialisable**

Elle se comporte comme si elle avait implémenté elle-même l'interface *Serializable*. Les attributs de la classe mère sont sérialisés selon l'implémentation de la classe dérivée.

* **Si la classe implémente l'interface Serializable et hérite d'une classe non sérialisable.**

Deux points fondamentaux à savoir :

* Les attributs hérités ne sont pas sérialisés.
* Il est nécessaire que la classe étendue possède un constructeur par défaut accessible, sinon une *InvalidClassException* est levée à l'exécution.

**Exemple**

class point implements Serializable{

private int x ;

private int y ;

public point( int x , int y ){

this.x = x ;

this.y = y ;

}

public void affiche(){

System.out.println("x= "+ x + " et y= "+y ) ;

}

public int getX(){

return x ;

}}

* **Sérialisation**

public class point\_fichiers{

public static void main(String [] args) throws Exception {

point a = new point(20,200);

FileOutputStream fichier = new FileOutputStream("fich\_point.dat");

ObjectOutputStream sortie = new ObjectOutputStream(fichier);

sortie.writeObject(a);

System.out.println("fichier rempli");

sortie.close();

}}

* **Désérialisation**

public class point\_fichiers\_L{

public static void main(String [] args) throws Exception {

FileInputStream fichier = new FileInputStream("fich\_point.dat");

ObjectInputStream ois = new ObjectInputStream(fichier);

point p = (point) ois.readObject();

//traiter l'objet p

ois.close();

p.affiche();

}}