1. NIO2 (new IO)

1.1. Introduction à nio , nio2

Dès l'origine du langage java , la classe File du package java.io était disponible pour accéder aux éléments du "file system" (fichier , répertoire, …) et pour effectuer des opérations dessus (tester l'existence d'un fichier , supprimer des fichiers , …) .

Bien qu'opérationnel , le package java.io (et la classe File) ont quelques limitations importantes :

- La classe File manque de fonctionnalité importante telle qu'une méthode "copy".
- Beaucoup de méthode retourne des booléens plutôt que des exceptions . Ce qui rend assez délicate l'analyse des problèmes (raison exacte de l'erreur?).
- Pas de bonne gestion des liens symboliques .
- Un ensemble très limité d'attributs sur les fichiers sont disponibles sur java.io.File.

Pour surmonter certaines lacunes , un nouveau package "java.nio" a été introduit dès java4 de façon à apporter quelques améliorations très techniques telles que les suivantes :

- *Channels et Selectors*: un "channel" est une abstraction d'une caractéristiques de bas niveau telle qu'un fichier mappé en mémoire .
- *Buffers*: Buffering pour toutes les classes wrapper/primitives (sauf pour Boolean).
- *Jeu de caractères*: **Charset** (*java.nio.charset*), avec encodeurs et décodeurs entre bytes[] et symboles "Unicode".

La version 7 de java a introduit un nouveau (gros) package "**java.nio.file"** (alias nio2) .

Ce package apporte à la fois de nouvelles fonctionnalités techniques telles que :

- gestion des liens symboliques
- bonne gestions des "Path" (aspect bien séparé).
- meilleur gestion des "File attributes"

et de nouvelles syntaxes (plus concises , plus "orienté objet") telles que les remontées d'exceptions.

Attention : Entre java1, java4 , java7 , des ajouts , mais pas de suppression (pour garder une compatibilité avec les anciennes versions).

Bien que la classe "java.io.File" n'est pas officiellement considérée comme obsolète / "deprecated" , il est très conseillé d'utiliser les nouvelles classes de nio2 (sachant qu'il existe des passerelles entre "java.io" et "java.nio.file") .

2. Principales classes et interfaces de nio2

NIO 2 (associé aux packages "**java.nio.file**" et "**java.nio.file.attribute**") repose sur plusieurs classes et interfaces dont les principales sont :

- **Path** : encapsule un chemin dans le système de fichiers
- **Files** : avec méthodes statiques pour manipuler les éléments du système de fichiers
- *FileSystemProvider* : fournisseur technique (implémentation) de FS
- **FileSystem** : encapsule un système de fichiers

- FileSystems : fabrique permettant entre autres de créer une instance de FileSystem
- **FileStorage** : système de stockage (ex : partition / volume logique , ….)

Quelques équivalences/transpositions de fonctionnalités basiques entre "java.io" et "nio2" :

Fonctionnalité	java.io	NIO 2
Encapsuler un chemin	java.io.File	java.nio.file.Path
Vérifier les permissions	File.canRead(), File.canCrite() et File.canExecute()	Files.isReadable(), Files.isWritable() et Files.isExecutable().
Vérifier le type d'élément	File.isDirectory(), File.isFile()	Files.isDirectory(Path, LinkOption...), Files.isRegularFile(Path, LinkOption...),
Taille d'un fichier	File.length()	Files.size(Path)
Obtenir ou modifier la date de dernière mise à jour	File.lastModified() , File.setLastModified(long)	Files.getLastModifiedTime(Path, LinkOption...), Files.setLastModifiedTime(Path, FileTime)
Modifier les attributs	File.setExecutable(), File.setReadable(), File.setReadOnly(), File.setWritable()	Files.setAttribute(Path, String, Object, LinkOption...)
Déplacer un fichier	File.renameTo()	Files.move()
Supprimer un fichier	File.delete()	Files.delete()
Créer un fichier	File.createNewFile()	Files.createFile()
Créer un fichier temporaire	File.createTempFile()	Files.createTempFile(Path, String, FileAttributes<?>), Files.createTempFile(Path, String, String, FileAttributes<?>)
Tester l'existence d'un fichier	File.exists	Files.exists() ou Files.notExists()
Obtenir le chemin absolu	File.getAbsolutePath() ou File.getAbsoluteFile()	Path.toAbsolutePath()
Chemin canonique (sans "." , "..")	File.getCanonicalPath() ou File.getCanonicalFile()	Path.toRealPath() ou Path.normalize()
Convertir en URI	File.toURI()	Path.toURI()
L'élément est-il caché	File.isHidden()	Files.isHidden()
Obtenir le contenu d'un répertoire	File.list() ou File.listFiles()	Path.newDirectoryStream()
Créer un répertoire	File.mkdir() ou File.mkdirs()	Path.createDirectory()
Obtenir le contenu du répertoire racine	File.listRoots()	FileSystem.getRootDirectories()
Place totale , libre , … sur FS	File.getTotalSpace() File.getFreeSpace()	FileStore.getTotalSpace() FileStore.getUnallocatedSpace()

3. Gestion des chemins (Path)

L'interface "java.nio.file.**Path**" est une représentation abstraite d'un chemin (relatif ou absolu) au sein d'un système de fichiers.

Un chemin peut référencer un fichier , un répertoire , un lien symbolique , un sous-chemin , …

Les instances de "Path" sont "immutables" et peuvent être utilisées dans un contexte multi-threads.

3.1. Obtention d'une instance de "Path"

Récupération d'un chemin en appelant explicitement **getPath**() sur une instance de "**FileSystem**" :

```
Path chemin = FileSystems.getDefault().getPath(
                    "C:/Users/powerUser/Temp/monfichier.txt");
```

équivalent indirect via la méthode (utilitaire / "helper") statique **Paths.get**() :

```
Path chemin = Paths.get("C:/Users/powerUser/Temp/monfichier.txt");
Path chemin2 = Paths.get(URI.create("file:///C:/Users/powerUser/Temp/monfichier.txt");
Path chemin3 = Paths.get(System.getProperty("java.io.tmpdir"),"monfichier.txt");
```

NB : bien que (sous windows) on puisse utiliser le séparateur "\\" (ex : C:\\RepXy\\monfichier.txt) , il vaut mieux utiliser le séparateur "/" (plus portable et plus simple) .

Passerelle "io / nio2" (depuis java 7) :
En partant d'une instance de l'ancienne classe **java.io.File** , on peut appeler la nouvelle méthode "**.toPath()**" pour récupérer une instance de "java.nio.file.Path".

3.2. Obtention des éléments d'un chemin

Méthode	Rôle
String **getFileName**()	Retourne le nom du dernier élément du chemin. Si le chemin concerne un fichier alors c'est le nom du fichier qui est retourné
Path **getName**(int **index**)	Retourne l'élément du chemin dont l'index est fourni en paramètre. Le premier élément (hors racine) possède l'index 0 et correspond à la première partie sous la racine (ex : "windows" sous racine "c:/").
int **getNameCount**()	Retourne le nombre d'éléments du chemin (hors racine)
Path **getParent**()	Retourne le chemin parent ou null s'il n'existe pas
Path **getRoot**()	Retourne la racine d'un chemin absolu (par exemple C:\ sous windows ou / sous Unix) ou null pour un chemin relatif
String **toString**()	Retourne le chemin sous la forme d'une chaîne de caractères
Path **subPath**(int	Retourne un sous-chemin (hors racine) [beginIndex,endIndex[

beginInclusiveIndex, int ***endExclusiveIndex***)	

Exemple avec un **chemin absolu** :	Exemple avec un **chemin relatif**:
path=C:\Windows\Fonts\arial.ttf toString() = C:\Windows\Fonts\arial.ttf getFileName() = arial.ttf getRoot() = C:\ getName(0) = Windows getNameCount() = 3 getParent() = C:\Windows\Fonts subpath(0,2) = Windows\Fonts	**path=Fonts\arial.ttf** toString() = Fonts\arial.ttf getFileName() = arial.ttf getRoot() = null getName(0) = Fonts getNameCount() = 2 getParent() = Fonts subpath(0,2) = Fonts\arial.ttf

3.3. Manipulation , combinaison et conversions de chemins

Méthode	**Rôle**
Path **normalize**()	Normaliser (ou rendre "canonique") un chemin en supprimant les éléments non indispensables « . » et « .. » qu'il contient.
Path **relativize**(Path other)	Retourner le chemin relatif permettant d'aller du "path courant" vers celui fourni en paramètres .
Path **resolve**(Path relative)	Combiner deux chemins (courant + relatif_en_arg) pour former global
Path **toAbsolutePath**()	Retourne un **chemin absolu** (en ***tenant compte du répertoire courant du contexte d'exécution*** (idem à "pwd")).
Path **toRealPath**(LinkOption...)	Retourner le **chemin physique** du chemin notamment **en résolvant les liens symboliques selon les options fournies**. Peut lever une exception si le fichier au bout du chemin courant (absolu ou relatif) n'existe pas .
URI **toUri**()	Retourner le chemin sous la forme d'une URI (file:///)

Exemple1 :

```
path.toString()  = C:\Utilisateurs\..\Windows\.\Fonts\arial.ttf
path.normalize() = C:\Windows\Fonts\arial.ttf
```

```
path.toString()   = ..\..\Fonts\.\arial.ttf
path.normalize()  = ..\..\Fonts\arial.ttf      (seul le /./ inutile supprimé)
```

Exemple2 :

```
Path windowsFontPath = Paths.get("c:/Windows/Fonts");
Path usersPath = Paths.get("c:/Utilisateurs");
                                          //relative path from current to arg
Path relativePathFromUsersToWindowsFonts=
usersPath.relativize(windowsFontPath);
System.out.println("relative path=" + relativePathFromUsersToWindowsFonts) ;
```

```
relative path=..\Windows\Fonts
```

Exemple3 :

```
Path windowsPath = Paths.get("c:/Windows");
Path arialRelativePath = Paths.get("Fonts/arial.ttf");
Path globalPath = windowsPath.resolve(arialRelativePath);
System.out.println("globalPath="+globalPath);

globalPath=c:\Windows\Fonts\arial.ttf
```

```
Path windowsFontsPath = Paths.get("C:/Windows/Fonts");
System.out.println("uri=" + windowsFontsPath.toUri());   //java.net.URI

uri=file:///C:/Windows/Fonts/
```

```
Path relativePath = Paths.get(".");
Path absolutePath = relativePath.toAbsolutePath(); //selon rep courant (pwd)
System.out.println("absolutePath=" + absolutePath);

absolutePath=C:\Users\didier\workspace\test_j7_j8\.
```

Rappel : sous linux , un lien symbolique se construit via la commande
$ **ln -s /nom_du_dossier_source nom_du_lien**

Attention, sous windows , un lien symbolique ne doit pas être confondu avec un raccourci .
A l'époque de windows XP , un lien symbolique se construisait avec la commande "***junction.exe***" (à installer).
Depuis "Vista" , et encore aujourd'hui avec windows 7, 8 et 10, un lien symbolique se construit avec la commande mklink suivante :

MKLINK [[/D] | [/H] | [/J]] Lien Cible
/D : Crée un lien symbolique vers un répertoire. Par défaut, il s'agit d'un lien symbolique vers un fichier.
/H : Crée un lien réel à la place d'un lien symbolique.
/J : Crée une jonction de répertoires.
Lien : Spécifie le nom du nouveau lien symbolique.
Cible : Spécifie le chemin d'accès (relatif ou absolu) auquel le nouveau lien fait référence.

Exemple (à lancer avec des droits "administrateur"):
cd c:\tmp\bb
mklink /J *lien_vers_aa c:\tmp\aa*
jonction créée pour lien_vers_aa <==> c:\tmp\aa

```
Path path = Paths.get("C:/tmp/bb/raccourci_vers_aa/f1.txt");
→ java.nio.file.NoSuchFileException: C:\tmp\bb\raccourci_vers_aa\f1.txt
```

```
Path path = Paths.get("C:/tmp/bb/lien_vers_aa/f1.txt");
try {
   System.out.println("chemin indirect sans suivre résolution lien symbolique="
                 + path.toRealPath(LinkOption.NOFOLLOW_LINKS));
   System.out.println("chemin direct suivant résolution lien symbolique="
                + path.toRealPath());
    } catch (IOException e) {
           e.printStackTrace();
      }
chemin indirect sans suivre résol lien symbolique=C:\tmp\bb\lien_vers_aa\f1.txt
chemin direct suivant résolution lien symbolique=C:\tmp\aa\f1.txt
```

3.4. Comparaisons de chemins

La méthodes **.equals()** permet de tester si deux chemins ont des valeurs identiques.

L'interface **Path** hérite de **Comparable**. Les chemins peuvent donc être automatiquement **triés**.

L'interface Path comporte en outre les méthodes de comparaison spécifiques suivantes :

Méthode	Rôle
int **compareTo**(Path other) (*Comparable*<Path>)	Comparer le chemin avec celui fourni en paramètre et retourne 0 si identiques , <0 avant , >0 si après
boolean **endsWith**(Path other) boolean **endsWith**(*String* other)	Comparer la fin du chemin avec celui fourni en paramètre
boolean **startsWith**(Path other) boolean **startsWith**(*String* other)	Comparer le début du chemin avec celui fourni en paramètre

3.5. interface PathMatcher avec méthode matches(path) et "glob"

Un "**glob**" est une expression basée sur certains méta-caractères qui seront comparés à des parties de "path" .

```
Path path = Paths.get("C:/tmp/aa/f1.txt");
PathMatcher txtMatcher = FileSystems.getDefault().getPathMatcher("glob:*.txt");
if (txtMatcher.matches(path.getFileName())) {
        System.out.println(path + " reference un fichier texte");
        }
```

Motif (dans glob)	Rôle associé / correspondance
*	Aucun ou plusieurs caractères
**	Aucun ou plusieurs sous-répertoires
?	Un caractère quelconque
{}	Un ensemble de motifs exemple : {htm, html}
[]	Un ensemble de caractères. Exemple : [A-Z] : toutes les lettres majuscules [0-9] : tous les chiffres [a-z,A-Z] : toutes les lettres indépendamment de la casse Chaque élément de l'ensemble est séparé par un caractère virgule Le caractère - permet de définir une plage de caractères

	A l'intérieur des crochets, les caractères *, ? et / ne sont pas interprétés
\	Il permet d'échapper des caractères pour éviter qu'ils ne soient interprétés. Il sert notamment à échapper le caractère \ lui-même
Les autres caractères	Ils se représentent eux-mêmes sans être interprétés

Quelques exemples :

Glob	*Correspondances*
***.html**	tous les fichiers ayant l'extension .html
???	trois caractères quelconques
[0-9]	tous les fichiers qui contiennent au moins un chiffre
***.{htm, html}**	tous les fichiers dont l'extension est htm ou html
Test*.java	tous les fichiers dont le nom commence par un Test et possède une extension .java

NB : L'interface **PathMatcher** (existant depuis java 7) comporte une unique méthode *"Boolean **matches**(Path path)* " . Elle peut donc être utilisée au sein de "lambda expression " depuis java 8 .

Exemple (fonctionnant depuis java8) :

…

….

4. Classe utilitaire Files *("helper" avec 50 méthodes statiques)*

4.1. Vérifications sur fichiers ou répertoires

Méthode	Rôle
boolean **exists**(Path) boolean **notExists**(Path)	vérifier l'existence sur le système de fichiers de l'élément dont le chemin est encapsulé dans le paramètre de type Path fourni
boolean **isReadable**(Path path)	peut être lu (droits en lecture) ?
boolean **isWritable**(Path path)	peut être modifié (droits en écriture) ?
boolean **isHidden**(Path path)	est caché ?
boolean **isExecutable**(Path path)	est exécutable ?
boolean **isRegularFile**(Path path)	est un fichier ?
boolean **isDirectory**(Path path)	est un répertoire ?
boolean **isSymbolicLink**(Path path)	est un lien symbolique ?
String **probeContentType**(Path path)	retourne type MIME d'un fichier (ex: "text/plain") (ou null

	si indéterminé). S'appuie par défaut sur l'OS sous jacent.

Exemple:

```
Path path = Paths.get("C:/tmp/aa/f1.txt");
if(Files.isRegularFile(path)){
                System.out.println(path + " est un chemin vers un fichier");
}
```

4.2. Création d'un fichier ou d'un répertoire

Méthode	Rôle ou bien exemple
Path **createFile**(Path path, FileAttribute<?>... attrs)	Créer un fichier dont le chemin est encapsulé par l'instance de type Path fournie en paramètre
Path **createDirectory**(Path dir, FileAttribute<?>... attrs)	Path monRepertoire = Paths.get("C:/temp/mon_repertoire"); **Files.createDirectory**(monRepertoire);
Path **createDirectories**(Path dir, FileAttribute<?>... attrs)	Créer d'un seul coup plusieurs niveaux de sous répertoire selon le chemin exprimé "C:/temp/**niveau1/niveau2/niv3**"
Path **createTempDirectory**(Path dir, String ***prefix***,FileAttribute<?>... attrs) Path **createTempDirectory**(String ***prefix***, FileAttribute<?>... attrs)	Créer un répertoire temporaire de nom "prefixe indiqué" + numéro_calculé_par_syst (ex : ***rep***_1245643) au sein du répertoire indiqué ou (à défaut) au sein du répertoire système (par défaut) prévu pour les temporaires.
Path **createTempFile**(Path dir, String ***prefix***, String ***suffix***, FileAttribute<?>... attrs) Path **createTempFile**(String ***prefix***, String ***suffix***, FileAttribute<?>... attrs)	Créer un fichier temporaire de nom "prefixe indiqué" + numéro_calculé_par_syst + "suffixe indiqué" (ex : **fic**_1245643**.txt**) au sein du répertoire indiqué ou (à défaut) au sein du répertoire système (par défaut) prévu pour les temporaires. Le suffix peut éventuellement être à null .

Les **attributs** (de type *FileAttribute*) sont **facultatifs** (des valeurs par défaut existent).

Les attributs possibles seront étudiés ultérieurement .

NB : **createDirectory**() créér un seul niveau de sous répertoire à la fois (*contrairement à createDirectories*) .

Si le répertoire existe déjà , l'exception FileAlreadyExists est remontée.

Si le répertoire parent n'existe pas , NoSuchFileException est remontée.

4.3. Copie d'un fichier ou d'un répertoire

Méthode	Rôle / fonctionnalité

Path **copy**(Path **source**, Path **target**, CopyOption... options)	Copier un élément avec les options précisées (StandardCopyOption.REPLACE_EXISTING , StandardCopyOption.COPY_ATTRIBUTES)
long **copy**(***InputStream*** **in**, Path **target**, CopyOption... options)	Copier tous les octets d'un flux de type InputStream vers un fichier
long **copy**(Path **source**, ***OutputStream*** **out**)	Copier tous les octets d'un fichier dans un flux de type OutputStream

NB : L'option pointue LinkOption.NOFOLLOW_LINKS permet de recopier si besoin un lien symbolique au bout du path indiqué (plutôt que de suivre le lien).

NB2: il est possible d'utiliser la méthode copy() sur un répertoire . Cependant, le répertoire sera créé sans que les fichiers contenus et .. ne soient eux aussi copiés .

Quoi que contienne le répertoire, la méthode copy ne créé qu'un répertoire vide. Pour copier le contenu du répertoire, il faut parcourir son contenu et copier chacun des éléments un par un.

4.4. Déplacement et suppression d'un fichier ou d'un répertoire

Méthode	Rôle
move(Path source, Path target, CopyOption... options)	Déplacer ou renommer un élément avec les options précisées (StandardCopyOption.REPLACE_EXISTING , StandardCopyOption.ATOMIC_MOVE)
void **delete**(Path path)	Supprimer un élément du système de fichiers (avec exception s'il n'existe pas ou si répertoire pas vide)
boolean **deleteIfExist**(Path path)	Supprimer un élément du système de fichiers s'il existe (sans exception s'il existe pas)

Si un déplacement efficace/performant demandé en mode "ATOMIC_MOVE" est impossible (par exemple déplacement de "c:/repXy" vers "d:/repXy") , une exception est alors remontée. On peut alors éventuellement ré-essayer sans l'option .

Des exceptions peuvent potentiellement remonter si un répertoire à déplacer n'est pas vide et que certains fichiers contenus sont en cours d'utilisation.

5. Parcours des éléments d'un répertoire

La solution de parcours proposée par NIO2 est plus performante que java.io.File.**list**(...) .

La méthode **newDirectoryStream**() de la classe utilitaire **Files** attend en paramètre un objet de type Path qui correspond au répertoire à parcourir et permet d'obtenir une instance de "**stream**" de type **DirectoryStream<Path>** (à parcourir avec un itérateur ou autre) .

Attention: il est très important d'invoquer la méthode **close**() de l'instance de type **DirectoryStream** pour libérer les ressources utilisées.

Exemple :

```
Path tmpAaPath = Paths.get("C:/tmp/aa");
DirectoryStream<Path> stream = null;
try {
     stream=Files.newDirectoryStream(tmpAaPath);
     Iterator<Path> iterator = stream.iterator();
     while(iterator.hasNext()) {
          Path p = iterator.next();
          System.out.println(p);
          }
     }
     catch(IOException ex){   ex.printStackTrace();
     }
     finally {
          try {stream.close();
          } catch (IOException e) {e.printStackTrace();
            }
     }
```

Exemple amélioré et simplifié :

```
Path tmpAaPath = Paths.get("C:/tmp/aa");

//NB1 : le second paramètre facultatif de Files.newDirectoryStream()
// sert à filtrer les éléments à parcourir (sans besoin de préfixe glob:)

//NB2: L'interface  DirectoryStream implemente hérite de Closable et
//le try(avec_ressource_implementant_interface Closable)
//sera automatiquement associé à un finally implicite déclenchant  .close()

try (DirectoryStream<Path> stream = Files.newDirectoryStream(tmpAaPath,"*.txt")){
 for(Path p : stream){
        System.out.println(p);
        }
}
catch(IOException ex){
        ex.printStackTrace();
}
```

On peut également paramétrer et utiliser un filtre spécifique lors du parcours :

```
Path tmpAaPath = Paths.get("C:/tmp/aa");
DirectoryStream.Filter<Path> littleSizeFilter = new   DirectoryStream.Filter<Path>() {
  public static final long MEGABYTE = 1024*1024;
   @Override
  public boolean accept(Path element) throws IOException {
                return (Files.size(element) <= MEGABYTE );
        }
 };//fin de classe anonyme imbriquée implémentant DirectoryStream.Filter<Path>

try (DirectoryStream<Path> stream = Files.newDirectoryStream(tmpAaPath,littleSizeFilter)){
        for(Path p : stream){
                System.out.println(p);
```

```
        }
}
```

NB : ce code (java7) pourra être amélioré/simplifié via une lambda expression de java8 .

6. Parcours d'une hiérarchie de répertoires (visiteur)

La méthode **Files.walkFileTree()** permet de parcourir une (sous-)arborescence de répertoires en utilisant le **design pattern "visiteur"**. Ce type de parcours peut être utilisé pour rechercher, copier, déplacer, supprimer, ... des éléments de la hiérarchie parcourue.

Il faut écrire une classe qui implémente l'interface **java.nio.file.FileVisitor<T>**. Cette interface définit des méthodes qui seront des callbacks appelées lors du parcours de la hiérarchie.

Méthode	**Rôle / fonctionnalité**
FileVisitResult **postVisitDirectory**(T dir, IOException exc)	Le parcours sort d'un répertoire qui vient d'être parcouru ou une exception est survenue durant le parcours
FileVisitResult **preVisitDirectory**(T dir, BasicFileAttributes attrs)	Le parcours rencontre un répertoire, cette méthode est invoquée avant de parcourir son contenu
FileVisitResult **visitFile**(T file, BasicFileAttributes attrs)	Le parcours rencontre un fichier
FileVisitResult **visitFileFailed**(T file, IOException exc)	La visite d'un des fichiers durant le parcours n'est pas possible et une exception a été levée

Les méthodes de l'interface FileVisitor renvoient toutes une valeur qui appartient à l'énumération FileVisitResult. Cette valeur permet de contrôler le processus de parcours de l'arborescence :

- CONTINUE : poursuite du parcours
- TERMINATE : arrêt immédiat du parcours
- SKIP_SUBTREE : inhibe le parcours de la sous-arborescence.
- SKIP_SIBLING : inhibe le parcours des répertoires frères.

Exemple(s) à ajouter ici plus tard .

7. FileSystem (par défaut et "personnalisé")

7.1. Fabrique FileSystems

FileSystems est une **fabrique** (avec **méthodes statiques**) pour obtenir des objets *FileSystem*.

.getDefault() renvoie l'instance de type FileSystem qui encapsule le F.S. de la JVM.

.getFileSystem(fsUri) renvoie un FileSystem selon l'URI est fourni en paramètre.

.newFileSystem() surchargée permet de créer une instance spécifique de type FileSystem (cas

pointu)

7.2. FileSystem

String separator = FileSystems.getDefault().**getSeparator**(); // / sous linux ou \ sous windows

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
  for (Path name: dirs) {
    System.err.println(name);   // C:\  , D:\ , ...
  }
```

7.3. FileSystem spécifique

...

8. Lecture et écriture dans un fichier

8.1. Vue d'ensemble

Principaux apports de NIO et NIO2 :

IO	NIO	NIO2
Depuis Java 1.0 et 1.1	Depuis **Java 1.4** (JSR 151)	Depuis **Java 7** (JSR 203)
Synchrone bloquant	Synchrone non bloquant	Asynchrone non bloquant
File **InputStream** **OutputStream** Reader (Java 1.1) Writer (Java 1.1) Socket RandomAccessFile	**FileChannel** SocketChannel ServerSocketChannel (Charset, Selector, ByteBuffer)	Path **AsynchronousFileChannel** **AsynchronousByteChannel** AsynchronousSocketChannel AsynchronousServerSocketChannel SeekableByteChannel

8.2. Options sur l'ouverture d'un fichier

L'énumération **StandardOpenOption** implémente l'interface *OpenOption* et définit les options d'ouverture standard d'un fichier :

Valeur	Signification

APPEND	Si le fichier est ouvert en écriture alors les données sont ajoutées au fichier. Cette option doit être utilisée avec les options CREATE ou WRITE
CREATE	Créer un nouveau fichier s'il n'existe pas sinon le fichier est ouvert
CREATE_NEW	Créer un nouveau fichier : si le fichier existe déjà alors une exception est levée
DELETE_ON_CLOSE	Supprimer le fichier lorsque son flux associé est fermé : cette option est utile pour des fichiers temporaires
DSYNC	Demander l'écriture synchronisée des données dans le système de stockage sous-jacent (pas d'utilisation des tampons du système) ???
READ	Ouvrir le fichier en lecture
SPARSE	Indiquer au système que le fichier est clairsemé ce qui peut lui permettre de réaliser certaines optimisations si l'option est supportée par le système de fichiers (c'est notamment le cas avec NTFS)
SYNC	Demander l'écriture synchronisée des données et des métadonnées dans le système de stockage sous-jacent
TRUNCATE_EXISTING	Si le fichier existe et qu'il est ouvert en écriture alors il est vidé. Cette option doit être utilisée avec l'option WRITE
WRITE	Ouvrir le fichier en écriture

8.3. Lecture de l'intégralité d'un fichier / Files.readAllLines()

Lecture (en boucle) de toutes les lignes d'un fichier texte :

```
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
…
List<String> lignes =  Files.readAllLines(
                FileSystems.getDefault().getPath("c:/tmp/aa/f1.txt"), StandardCharsets.UTF_8);
for (String ligne : lignes)
                System.out.println(ligne);
```

Lecture d'un bloc de tout un (petit) fichier binaire :

```
byte[] binaryContent = Files.readAllBytes(binaryFilePath);
```