CSC4509 — JAVA NIO

Éric Lallet
Eric.Lallet@telecom-sudparis.eu

Télécom SudParis

30 avril 2025

Java NIO vs Java IO

- Historique :
  - Java IO : depuis Java 1.0
  - Java NIO : depuis Java 1.4, version 2 depuis Java 1.7
- Principales différences :
  - Gestion du système de fichiers : liens symboliques, réelle portabilité entre différents SE, exploration des répertoires. . .(pas abordée dans ce cours).
  - Gestion des communications réseaux : gestion par bloc plutôt que par flux, possibilité de gérer des communications asynchrones sans boucle active. . ..
- Par conséquent :
  - Plus facile d'usage (pour l'accès au système de fichiers).
  - Plus portable.
  - Plus performant.

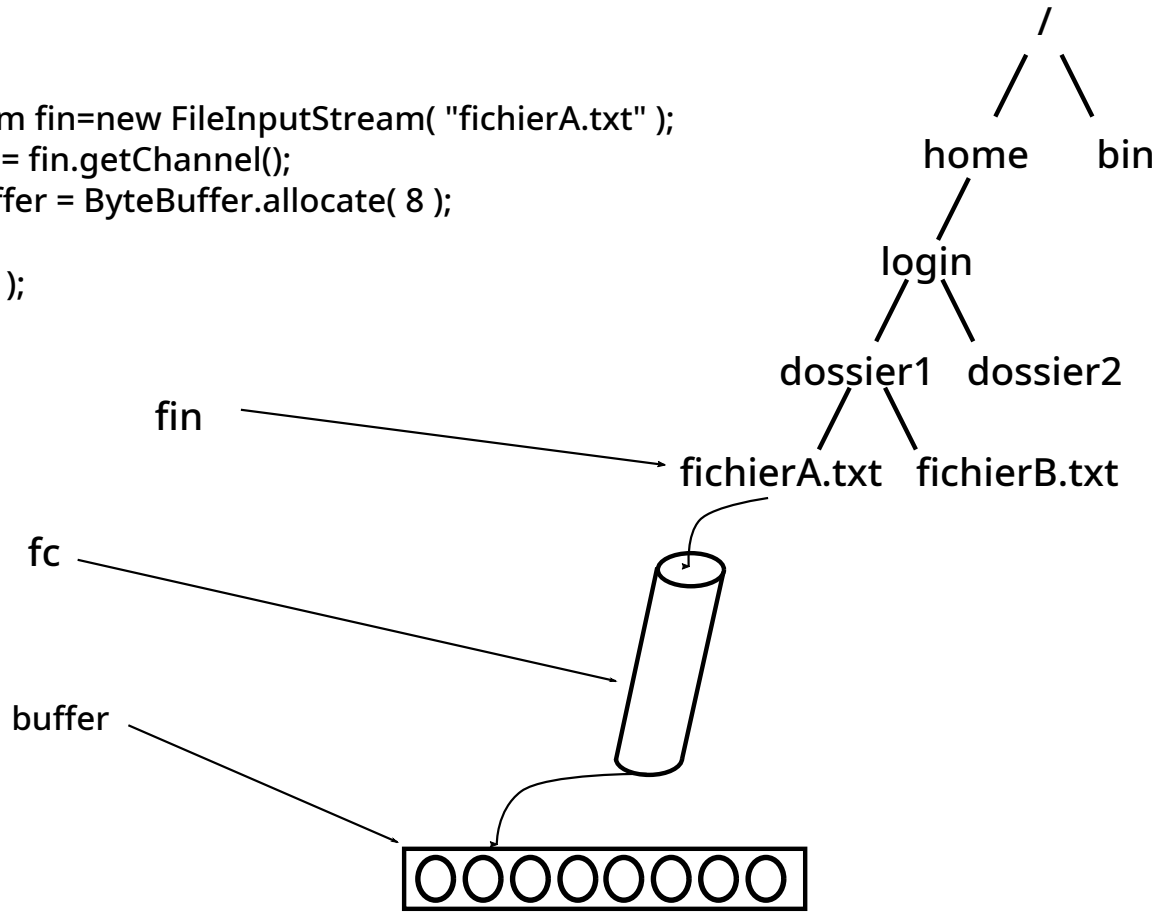
Buffeur et canal

- Ne plus écrire ou lire directement sur les flots (stream en anglais)
- Relier le flot à un buffeur par un canal.
- Faire les opérations de lectures et d'écritures par bloc sur un buffeur via ce canal.

Buffeur et canal

```
FileInputStream fin=new FileInputStream( "fichierA.txt" );
FileChannel fc = fin.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 8 );

fc.read( buffer );
```



Les buffeurs Java NIO

Java NIO fournit 7 types de buffeurs :

- *ByteBuffer*
- *CharBuffer*
- *ShortBuffer*
- *IntBuffer*
- *LongBuffer*
- *FloatBuffer*
- *DoubleBuffer*

Création du buffeur par la méthode de classe `allocate()` :

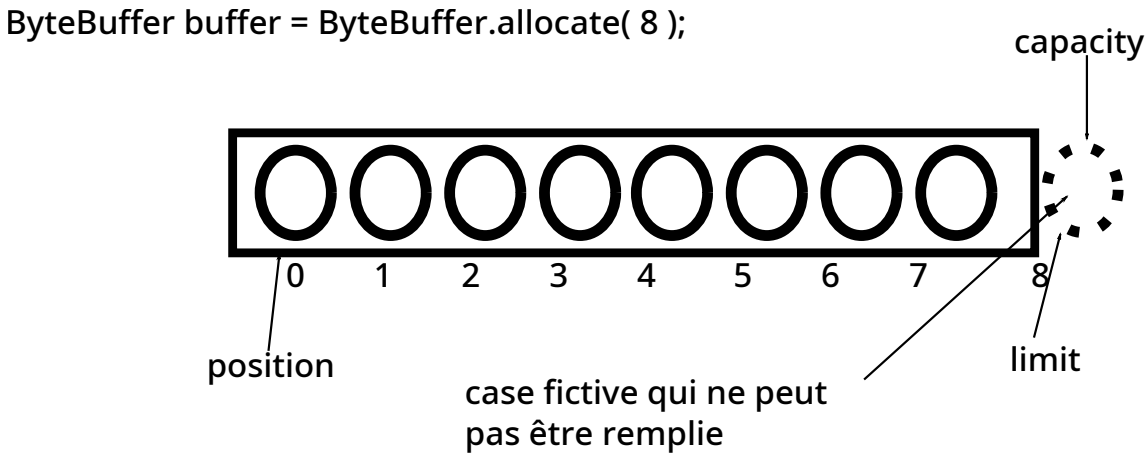
```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

Les buffeurs Java NIO : les curseurs

L'état d'un buffeur est caractérisé par trois valeurs qui servent de curseurs. La signification de ces curseurs va dépendre de l'opération que l'on veut faire avec ce buffeur (consultation ou modification).

- *position*
- *limit*
- *capacity*

Buffeur avec ses curseurs dans leur état initial :



Les buffeurs Java NIO : les curseurs pour une modification

Pour une modification (remplissage du buffeur) :

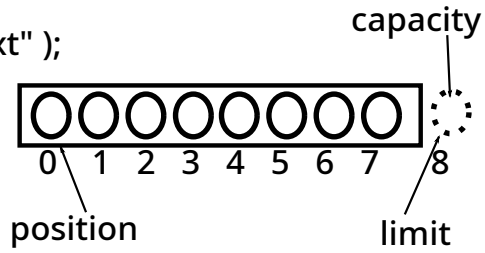
position : Avant la modification, *position* indique la case à partir de laquelle le buffeur va être rempli. Après la modification, *position* indique la première case qui n'a pas encore été remplie.

limit : indique la limite de remplissage du buffeur (la case numérotée par *limit* ne sera pas remplie).

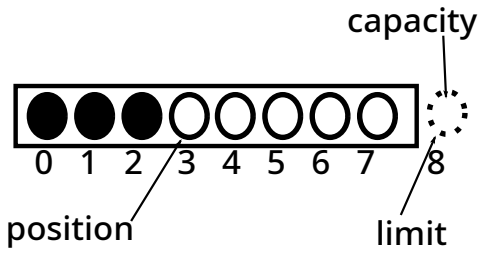
capacity : indique la valeur maximale de *limit*. *limit* peut être changé arbitrairement à tout moment, mais ne peut pas dépasser *capacity*.

Les buffeurs Java NIO : les curseurs pour la modification

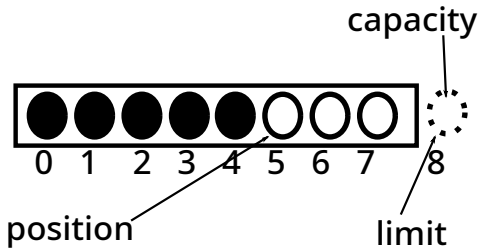
```
FileInputStream fin=new FileInputStream( "fichierA.txt" );
FileChannel fc = fin.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 8 );
```




```
fc.read( buffer ); // première lecture de 3 octets
```




```
fc.read( buffer ); // seconde lecture de 2 octets
```



Les buffeurs Java NIO : les curseurs pour une consultation

Pour une consultation (utilisation des données du buffeur) :

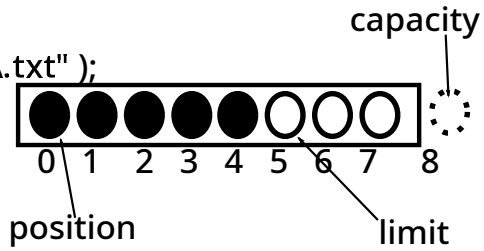
position : Avant la consultation, indique la case à partir de laquelle les données vont être prises dans le buffeur. Après la consultation, indique la case de la première donnée qui n'a pas encore été extraite.

limit : indique la limite des cases qui contiennent des données à extraire (la case numérotée par *limit* ne contient pas de données significatives).

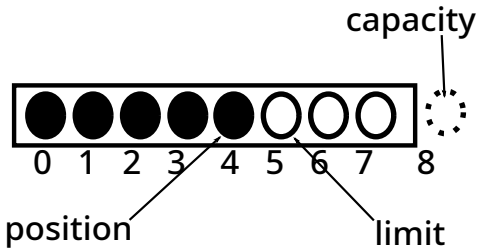
capacity : indique la valeur maximale de *limit*. Il n'est pas pertinent de changer la valeur de *limit*, puisque les cases au-delà de *limit* ne contiennent pas de données significatives.

Les buffeurs Java NIO : les curseurs pour une consultation

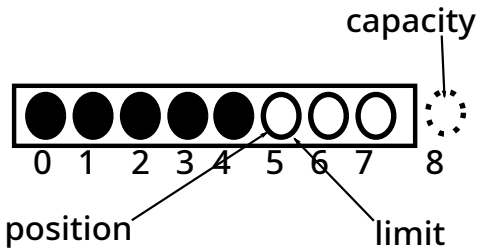
```
ByteBuffer buffer = ByteBuffer.allocate( 8 );
// ici des instuctions qui ont rempli le buffeur et
// positionné les curseurs.
FileOutputStream fout=new FileOutputStream( "fichierA.txt" );
FileChannel fcout = fout.getChannel();
```




```
fcout.write(buffer); //première écriture de 4 octets
```




```
fcout.write(buffer);// seconde écriture de 1 octet
```



Les buffeurs Java NIO : les méthodes gérant les curseurs

Les classes Buffer fournissent des méthodes pour la gestions des curseurs.
Les deux Principales :

`Buffer clear()` : remet les curseurs dans leur état initial : position va en 0, et *limit* en *capacity*. Donc remet en état un buffeur (dont on n'a plus besoin du contenu) pour un nouveau remplissage.

Buffer flip() : met *limit* en *position*, et ensuite *position* en 0. Donc prépare un buffeur pour une extraction des données après un remplissage.

Les buffeurs Java NIO : les méthodes gérant les curseurs

Consultation et changement manuel :

`int position()` : retourne la valeur de *position*.

`Buffer position(int newPos)` : change la valeur de *position* à *newPos* (doit être compris entre 0 et *limit*).

`int limit()` : retourne la valeur de *limit*.

`Buffer limit(int newLimit)` : change la valeur de *limit* à *newLimit* (doit être compris entre 0 et *capacity*).

`int capacity()` : retourne la valeur de *capacity*.

Les buffeurs Java NIO : les méthodes gérant les curseurs

Quelques autes méthode :

`Buffer reset()` : annule les derniers changements de *position* depuis le dernier `mark()` (donc, annule les derniers ajouts ou extractions de données).

`Buffer rewind()` : met *position* à 0 (donc, permet d'extraire une nouvelle fois les mêmes données).

`int remaining()` : retourne la différence entre *limit* et *position* (donc, soit la capacité restante pour les prochaines lectures, soit les données restantes pour les prochaines écritures).

`boolean hasRemaining()` : retourne *false* si *limit* vaut *position*, *true* sinon (donc, retourne *true* si on peut encore écrire dans le buffeur en cas de remplissage, ou s'il y a encore des données à extraire du buffeur, en cas de consultation).

Les buffeurs Java NIO : d'autres méthodes

Quelques méthodes pour extraire des données du buffer :
Méthodes relatives (curseurs utilisés et modifiés) :

- `byte get()`
- `ByteBuffer get(byte dst[])`
- `ByteBuffer get(byte dst[], int offset, int length)`
- `byte getByte(), char getChar(), int getInt()...`

Méthodes absolues (curseurs ni utilisés, ni modifiés) :

- `byte get(int index )`
- `byte getByte(int index), char getChar(int index), int getInt(int index)...`

Les buffeurs Java NIO : d'autres méthodes

Quelques méthodes pour placer des données dans le buffeur :
Méthodes relatives (curseurs utilisés et modifiés) :

- `ByteBuffer put(byte b)`
- `ByteBuffer put(byte src[])`
- `ByteBuffer put(byte src[], int offset, int length)`
- `ByteBuffer put(ByteBuffer src)`
- `ByteBuffer putChar(char value)`, `ByteBuffer putInt(int value)...`

Méthodes absolues (curseurs ni utilisés, ni modifiés) :

- `ByteBuffer put(int index, byte b);`
- `ByteBuffer putChar(int index, char value)`, `ByteBuffer putInt(int index, int value)...`

Les canaux

- Le canal fait le lien entre le flot et le buffeur.
- Choisir la classe de canal adaptée à la classe de Stream.
- Création du canal par une méthode d'instance du Stream utilisé ou par une méthode de classe de la classe Channels

Exemple :

```
FileInputStream fin=new FileInputStream("ficA");
FileChannel fcin=fin.getChannel();

WritableByteChannel wbout = Channels.newChannel((OutputStream) System.out);
```

Les canaux : les méthodes principales

Principales méthodes de la classe *FileChannel* :

`int read(ByteBuffer dst)` : lit des données du fichier pour les placer dans dst. Retourne le nombre d'octets lus.

`int write(ByteBuffer src)` : écrit les données de src dans le fichier. Retourne le nombre d'octets écrits.

`void close()` : ferme le canal.

Les canaux : exemple d'usage

```
int res;
try (FileInputStream fin=new FileInputStream("ficA");
     FileOutputStream fout=new FileOutputStream("ficB");
     FileChannel fcin=fin.getChannel();
     FileChannel fcout=fout.getChannel();) {
       ByteBuffer buffer=ByteBuffer.allocate(1024);
       do {
          buffer.clear();
          res=fcin.read(buffer);
          buffer.flip();
          fcout.write(buffer);
      } while (res == 1024);
} catch (IOException ioe) {
        System.err.println(ioe);
}
```