

## Les nouvelles (1.4) entrées/sorties

Rémi Forax

---

# Plan

---

- Les Buffers
- Les canaux
- Les fichiers mappés
- Charset
- Encoder/Decoder

---

# Avant propos

---

- Depuis Java 1.4, [java.nio.\\*](#) NIO (*New Input Output*)
  - Gestion plus fine de la mémoire
  - Gestion plus performante des entrées-sorties
  - Gestion simplifiée des différents jeux de caractères
  - Interaction plus fine avec le système de fichiers
  - Utilisation d'entrées-sorties non bloquantes (*plus tard*)

---

# Avant propos

---

- Nouveau paquetages :
  - **Buffers** (tampons mémoire) `java.nio.*`
  - **Charsets** (jeux de caractères) `java.nio.charset.*`
  - **Channels** (canaux) `java.nio.channels.*`
- NIO 2 prévu pour la 1.5 puis 1.6 devrait être dans la 1.7

---

# Les buffers

---

- Utilisés comme tampons par les primitives d'entrées-sorties de [java.nio](#)
- Remplace les tableaux utilisés en [java.io](#)
- Zone de mémoire contiguë, permettant de stocker
  - une quantité de données fixée,
  - d'un type primitif donné
- Pas prévus pour accès concurrent
  - On évite de les partager

---

# Les buffers

---

- `java.nio.Buffer`
  - classe abstraite de tous les buffers
  - un ensemble de cases mémoire, avec 3 pointeurs
- Propriétés :
  - **position** : position dans le buffer
  - **capacity**: nombre de cases mémoire allouées
  - **limit** : position à partir de laquelle il ne faut plus lire/écrire
  - **mark** : position avant la position courante où l'on peut revenir

---

# Buffer de types primitifs

---

- Chaque type primitif possède une classe abstraite de buffer dédié
  - `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer` et `DoubleBuffer`
- Les classes `ByteBuffer` et `CharBuffer` fournissent un ensemble de méthodes et d'opérations plus riches que pour les autres types de buffers

---

# Création de Buffer

---

- Les `ByteBuffer` sont utilisés pour échanger des données en lecture ou écriture avec le système
- `AnyBuffer.allocate(int capacity)` renvoie un buffer dont les éléments sont stockés dans un tableau classique
  - Peu-couteux en allocation
  - Besoin d'un buffer intermédiaire en case de read/write ( le GC peut bouger les objets en mémoire )
- On parle de « managed buffer »



---

# ByteBuffer direct

---

- Solution: un ByteBuffer direct est géré par le GC mais sa zone mémoire est allouée dans un espace mémoire non géré par le GC
- `ByteBuffer.allocateDirect()` renvoie un buffer **direct**
  - Coûteux en allocation et désallocation
  - Mieux si la taille est une puissance de 2 (moins de fragmentation)
  - A réserver aux buffers d'entrées-sorties de taille et de durée de vie importantes
- `isDirect()` indique si le buffer est direct ou pas

---

# Créer un buffer sur un tableau

---

- Méthode statique `wrap()` dans chaque classe de buffer (pour chaque type primitif) pour envelopper un tableau
- `IntBuffer wrap(int[] tab, int offset, int length)` ou `IntBuffer wrap(int[] tab)`
  - Enveloppe la totalité du tableau: capacité vaut `tab.length`
  - Position du tampon produit est mise à `offset` (sinon `0`)
  - Limite est mise à `offset+length` (sinon `tab.length`)
  - Si un tampon est une enveloppe de tableau
    - `hasArray()` retourne `true`
    - `array()` retourne le tableau
    - `arrayOffset()` retourne le décalage du tampon par rapport au tableau

---

# Accès aux données

---

- Deux manières d'accéder aux éléments d'un buffer
  - Accès **aléatoire** (*absolute*)
    - Relativement à un indice (comme dans un tableau)
  - Accès **séquentiel** (*relative*)
    - Relativement à la position courante (comme un flot)
    - La position courante représente l'indice du prochain élément à lire ou à écrire

---

# Accès aux données

---

- Par ex. avec un ByteBuffer, l'accès **aléatoire** (*absolute*)
  - `byte get(int index)` donne l'élément à la position `index`
  - `ByteBuffer put(int index, byte value)` ajoute `value` à l'indice `index`, et renvoie le buffer modifié
    - Appel chaînable
    - Peuvent lever `IndexOutOfBoundsException`
- Accès **séquentiel** (*relative*)
  - `byte get()` resp. `ByteBuffer put(byte value)`
    - Donne la valeur (resp. met `value`) à la position courante
    - Incrémente la position
    - Peuvent lever des exceptions `BufferUnderflowException` ou `BufferOverflowException` respectivement

---

# Accès groupé aux données

---

- Les méthodes existent en version "groupé" (*bulk*)
  - Manipulent un tableau au lieu d'une variable
  - L'opération réussit où rien ne se passe
- ByteBuffer **get**(byte[] dest, int offset, int length) et ByteBuffer **get**(byte[] dest)
  - Tentent de lire **le nombre d'éléments spécifié, ou rien** si pas assez de choses à lire (BufferUnderflowException)

---

# Accès groupé aux données

---

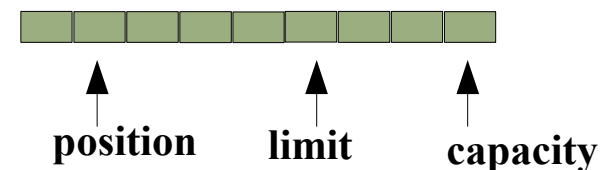
- ByteBuffer **put**(byte[] src, int offset, int length) et ByteBuffer **put**(byte[] src)
  - Tentent d'écrire **le nombre d'éléments spécifié, ou rien** si pas assez de place (BufferOverflowException)
- ByteBuffer **put**(ByteBuffer src)
  - Tente d'écrire le contenu de **src** (BufferOverflowException)

---

# Les méthodes d'un tampon

---

- **position**: indice du prochain élément accessible
  - consultable: `int position()`
  - modifiable: `Buffer position(int newPosition)`
- **capacity**: nombre d'éléments qui peuvent être contenus
  - fixée à la création du tampon
  - consultable par `int capacity()`
- **limit**: indice du premier élément ne devant pas être atteint
  - par défaut, égale à la capacité.
  - fixée par `Buffer limit(int newLimit)`
  - consultable par `int limit()`

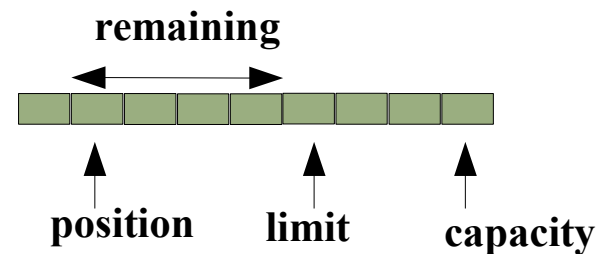


---

# Les méthodes d'un tampon

---

- **remaining:**
  - `int remaining()` donne le nombre d'éléments entre la position courante et la limite (limit-position)
  - `boolean hasRemaining()` vaut vrai si la position est strictement inférieure à limite



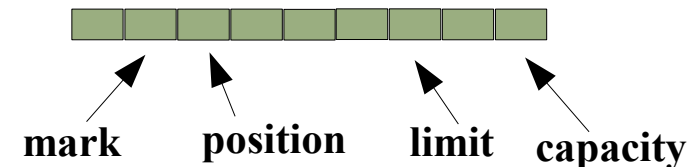


---

# La marque

---

- **Marque** (éventuelle): position dans le tampon
  - `Buffer mark()` place la marque à la position courante
  - `Buffer reset()` place la position à la marque  
ou lève `InvalidMarkException`
  - La marque est toujours inférieure ou égal à la position.  
Si la position ou la limite deviennent plus petite que la marque, la marque est effacée
  - `Buffer rewind()`  
met la **position** à 0 et supprime la **marque**





# Opération habituelle

- **Clear()**



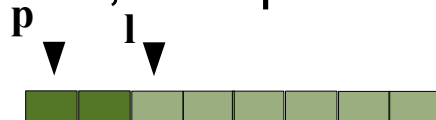
position à 0, limite à capacity, marque indéfinie

- **Rewind()**



position à 0, marque indéfinie

- **Flip()**



limite à position, position à 0, marque indéfinie.

- **compact()**



- Décale les éléments de la position courant à limite vers 0
- La position est après le dernier élément, la limite est à capacité et la marque effacée.

---

# Egalité, comparable

---

- Deux tampons sont égaux au sens de equals() si
  - ils ont le même type d'éléments,
  - ils ont le même nombre d'éléments restants et
  - les deux séquences d'éléments restants (remaining()), sont égales élément par élément  
(indépendamment de leurs positions)
- Un *AnyBuffer* implante Comparable<AnyBuffer>
  - compareTo() compare les séquences d'éléments restants (au sens de remaining()) de manière lexicographique (le prochain, puis le suivant, etc.)

- `duplicate()` retourne un buffer **partagé**
  - toute modification de données de l'un est vue dans l'autre
  - chaque tampon possède ses propres propriétés (position, limite, marque), les propriétés du nouveau sont initialisées à partir de l'ancien.
- `slice()` retourne un buffer **partagé** ne permettant de "**voir**" que ce qui reste à lire dans le tampon de départ
  - sa capacité est égale au `remaining()` du tampon de départ
  - La position du nouveau est 0 et la marque n'est plus définie

---

# Vues read-only

---

- Sur n'importe quel buffer, `asReadOnlyBuffer()` retourne un **vue** en **lecture seule**
  - Les méthodes comme `put()` lèvent l'exception `ReadOnlyException`
  - Peut être testé avec `isReadOnly()`

---

# ByteBuffer et vues d'un autre type

---

- Possibilité de créer des "**vues**" d'un ByteBuffer **comme** s'il s'agissait d'un buffer d'un autre type
  - `asCharBuffer()`, `asShortBuffer()`, `asIntBuffer()`,  
`asLongBuffer()`, `asFloatBuffer()`, `asDoubleBuffer()`
  - Seule façon d'obtenir des *IntBuffer* direct  
`ByteBuffer.allocateDirect(8192).asIntBuffer();`

---

# Buffer de caractères

---

- Les buffers de caractères `CharBuffer` implantent l'interface `CharSequence`
  - Permet de rechercher des motifs (expressions régulières) directement sur les tampons:  
classe `Pattern`: `matcher()`, `matches()`, `split()`
    - Pour toute recherche, penser à revenir au début du tampon, par exemple avec `flip()`, car seuls les caractères restants à lire sont pris en compte
  - `toString()` retourne la chaîne entre la position courante et la limite
  - `wrap()` peut accepter (en plus d'un `char[]`), n'importe quel `CharSequence` : `String`, `StringBuffer` ou `CharBuffer`
    - Dans ces derniers cas, le tampon est en ***lecture seule***.

---

# Buffer de caractères

---

- **CharBuffer** implantent les interfaces :
  - **Appendable**: un truc auquel on peut ajouter des **char**
    - Soit un caractère tout seul: **Appendable append(char c)**
    - Soit tout ou partie d'une **CharSequence** :  
**Appendable append(CharSequence csq)** et  
**Appendable append(CharSequence csq, int start, int end)**
  - **Readable**: un truc dans lequel on peut lire des **char**
    - **int read(CharBuffer cb)**
    - Retourne le nombre de caractères lus et placés dans le **CharBuffer** cb, ou -1 si le **Readable** n'a plus rien à lire



---

# FileChannel

---

- Un canal **FileChannel** reflète un flot
  - Classe abstraite: instances obtenues par les méthodes **getChannel()** de
    - **FileInputStream**, **FileOutputStream** ou **RandomAccessFile**
    - Ne supporte que les méthodes correspondantes, sinon **NonWritableChannelException** ou **NonReadableChannelException**
  - Le flot et le canal sont liés et se reflètent leurs états respectifs
  - Méthodes **read()** et **write()** conformes aux flots
    - **read()** retourne -1 quand la fin de fichier est atteinte
    - Lecture et écriture supportent la concurrence

---

# FileChannel & Buffer-cache

---

- L'écriture des données n'est pas immédiate
  - Méthode `force()` pour écriture des données sur le fichier
- Permet la copie par bloc
  - Lit d'un `ReadableByteChannel` vers le canal courant
    - long `transferFrom`(`ReadableByteChannel src`, long position, long count)
  - Écrit sur un `WritableByteChannel`, les données du canal courant
    - long `transferTo`(long position, long count, `WritableByteChannel target`)

---

# Fichiers mappés en mémoire

---

- Un `MappedByteBuffer`
  - étend `ByteBuffer`
  - permet de voir un fichier comme un buffer dans lequel on peut lire ou écrire
    - Opérations beaucoup plus rapides mais
    - Coût pour réaliser le mapping (surtout sous Windows)
- Le fichier mappé reste valide jusqu'à ce que le `MappedByteBuffer` soit garbage-collecté (aie !)

---

# FileChannel.map()

---

- Le `MappedByteBuffer` est obtenu par `map()` sur un `FileChannel` avec arguments:
  - `FileChannel.MapMode`
    - `READ_ONLY`, fichier ne peut pas être modifié via ce tampon
    - `READ_WRITE`, le fichier est modifié (avec un délai, cf. `force()`)
    - `PRIVATE` crée une copie privée du fichier. Aucune modification répercutée sur le fichier réel
  - `long position`
  - `long taille` (`FileChannel.size()` permet d'avoir sa taille)

---

# Les jeux de caractères

---

- `java.nio.charset.Charset` : représente une association entre
  - un jeu de caractères (sur un ou plusieurs octets)
  - et le codage Unicode "interne" à Java sur 2 octets
- Liste de jeux de caractères officiels gérée par IANA:  
<http://www.iana.org/assignments/character-sets>
- Référencé par un nom (***canonique***, US-ASCII, ou ***alias*** ASCII)
- `CharsetEncoder` : encodeur
- `CharsetDecoder` : décodeur

---

# Classe Charset

---

- Jeux de caractères disponibles sur la plateforme
  - `Charset.availableCharsets()` retourne une `SortedMap` associant les noms aux `Charset`
  - La plateforme Java requiert au minimum:  
`US-ASCII`, `ISO-8859-1`, `UTF-8`, `UTF-16BE`, `UTF-16LE`, `UTF-16`
  - `Charset.isSupported(String csName)` vrai si la JVM supporte le jeu de caractères dont le nom est passé en argument
  - `Charset.forName(String csName)` retourne le `Charset`
  - Pour un `Charset` donné, `name()` donne le nom canonique et `aliases()` donne les alias
  - `contains()` teste si un jeu de caractères en contient un autre

---

# Encoder/Decoder

---

- **CharsetEncoder** : encodeur
  - Transforme une séquence de caractères Unicode codés sur 2 octets en une séquence d'octets représentant ces caractères, mais utilisant un autre jeu de caractères.
    - CharBuffer vers ByteBuffer
- **CharsetDecoder** : décodeur
  - À partir d'une séquence d'octets représentant des caractères dans un jeu de caractères donné, produit une suite de caractères Unicode représentés sur deux octets.
    - ByteBuffer vers CharBuffer

---

# CharsetEncoder

---

- Création: `charset.newEncoder()`
- Méthode `encode()` la plus simple
  - Accepte un `CharBuffer` et encode son contenu (*remaining*) dans un `ByteBuffer` alloué pour l'occasion
    - `IllegalStateException` si opération de codage déjà en cours
    - Ou bien `CharacterCodingException` qui peut être:
      - `MalformedInputException` si valeur d'entrée incorrecte
      - `UnmappableCharacterException` si caractère d'entrée n'a pas de codage dans le jeu de caractères de destination
  - Racourci:  
`Charset.forName("ASCII").encode("texte à coder");`



---

# Gestion des problèmes de codage

---

- On peut spécifier un comportement en indiquant une constante de type `CodingErrorAction`
  - `IGNORE` permet d'ignorer simplement le problème
  - `REPLACE` permet de remplacer le caractère non valide ou non codable par une séquence d'octets (par défaut '?')
    - `byte[] replacement()` permet de la consulter
    - `replaceWith(byte[])` permet d'en spécifier une nouvelle
  - `REPORT` provoque la levée d'exception (par défaut) `MalformedInputException` ou `UnmappableCharacterException`
- Editable avec les méthodes `malformedInputAction()/onMalformedInput()` et `unmappableCharacterAction()/onUnmappableCharacter()`

---

# Exemple de codage vers ASCII

---

```
Charset ascii = Charset.forName("ASCII");
CharsetEncoder versASCII = ascii.newEncoder();
CharBuffer cb = CharBuffer.wrap("accentués et J€€");
// (a)
// versASCII.replaceWith(new byte[]{'$'});
// versASCII.onUnmappableCharacter(
//         CodingErrorAction.REPLACE);
try {
    ByteBuffer bb = versASCII.encode(cb);
// UnmappableCharacterException si (a) en commentaire
    while (bb.hasRemaining()) {
        System.out.print(((char)bb.get()));
    } // affiche "accentu$s et J$$" en décommentant (a)
} catch(CharacterCodingException cce) {
```

---

# Méthode `encode()` plus complète

---

- `CoderResult encode(CharBuffer in, ByteBuffer out, boolean endOfInput)`
  - Encode au plus `in.remaining()` caractères de `in`
  - Écrit au plus `out.remaining()` octets dans `out`
  - Fait évoluer les positions des deux buffers
  - Retourne un objet `CoderResult` représentant le résultat de l'opération d'encodage. Ce résultat peut être testé:
    - `isUnderflow()` vrai si pas assez de caractères dans `in`
    - `isOverflow()` vrai si pas assez de place dans `out`
    - `isError()` vrai si erreur produite (*malformed* ou *unmappable*)
    - `isMalformed()` si un caractère mal formé a été rencontré
    - `isUnmappable()` si caractère pas codable dans le jeu de sortie

---

# Méthode `encode()` (suite)

---

- Le 3<sup>e</sup> argument booléen `endOfInput`
  - Comme on encode un buffer et pas un flot, il faut signaler la fin à encode
  - S'il est à `false`, il indique que d'autres caractères doivent encore être décodés (tous appels sauf dernier)
    - L'état interne du codeur peut les attendre
  - Il doit être à `true` lors du dernier appel à cette fonction

---

# Méthode `encode()` (fin)

---

- L'encodage est terminé lorsque
  - Le dernier appel à `encode()`, avec `endOfInput` à `true`, a renvoyé un `CoderResult` tel que
    - le buffer `in` à `hasRemaining` à `false`
    - `isUnderflow()` est `true` (plus rien à lire)
  - Il faut faire `flush()` pour terminer le codage (purge des états internes)
    - On a des états internes car un caractère peut être codé sur plusieurs chars (encoder) et qu'un caractère peut être codé sur plusieurs bytes (decoder)

---

# Principe d'utilisation pour codage

---

1. (Optionel) Remettre à jour l'encodeur avec `reset()`
  - Purge des états internes
2. Appeler la méthode `encode()` zéro fois ou plus
  - Tant que de nouvelles entrées peuvent être disponibles
  - En passant le troisième argument à `false`, en remplissant le buffer d'entrée et en vidant le buffer de sortie à chaque fois
  - Cette méthode ne retourne que lorsqu'il n'y a plus rien à lire, plus de place pour écrire ou qu'en cas de pbme de codage
3. Appeler la méthode `encode()` une dernière fois
  - En passant le troisième argument à `true`
4. Appeler la méthode `flush()`
  - Purger les états internes de l'encodeur dans le buffer de sortie

---

# Méthodes de l'encodeur

---

- Dimensionner les buffers d'octets/caractères
  - `float averageBytesPerChar()` : # moyen d'octet par `char`
  - `float maxBytesPerChar()` : pire des cas
- Assurer qu'une séquence de remplacement est correcte
  - `boolean isLegalReplacement(byte[] repl)`
- Savoir si on est capable d'encoder un ou plusieurs `char`
  - En effet, certains caractères sont "couplés" (*surrogate*)
    - `boolean canEncode(char c)`
    - `boolean canEncode(CharSequence cs)`
  - Attention, ces méthodes peuvent changer l'état interne de l'encodeur (ne pas les appeler si encodage en cours)

---

# Le décodage

---

- Principe semblable à celui du codage
  - Instance d'une sous-classe de la classe abstraite `CharsetDecoder`, créer par `Charset.newDecoder()`
  - `maxCharsPerByte()` et `averageCharsPerByte()`
- Méthodes `decode()`
  - Lit un tampon d'octets et produit un tampon de caractères
  - Version complète avec `ByteBuffer` d'entrée, `CharBuffer` de sortie et paramètre booléen `endOfInput` retournant un `CoderResult`
- Les caractères à produire en cas de problème de décodage sont fournis par `replacement()` et `replaceWith()` qui manipulent des `String` au lieu de `byte[]`