

Applications réseaux

Nouvelles entrées-sorties « java.nio »
Canaux et mode non-bloquant

Etienne Duris
Arnaud Carayol

Les nouvelles entrées-sorties en Java

- Depuis SDK 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*)
- Nouveaux concepts dans les entrées-sorties en Java
 - **Buffers** (tampons mémoire) [java.nio.*](#)
 - **Charsets** (jeux de caractères) [java.nio.charset.*](#)
 - **Channels** (canaux) [java.nio.channels.*](#)

Les tampons mémoire (*buffers*)

- Utilisés par les primitives d'entrées-sorties de [java.nio](#)
 - Remplace les tableaux utilisés en [java.io](#)
 - IO « classiques » : ex. `InputStream : read(byte[])`
 - NIO : ex. `Channel : read(ByteBuffer)`
 - Buffer = zone de mémoire contiguë, permettant de stocker
 - une quantité de données fixée,
 - d'un type primitif donné
 - A priori pas prévus pour un accès concurrent
 - il faudra les protéger en cas de besoin...
- Représentés par des classes abstraites
 - Permet de dédier l'implémentation native à la plate-forme d'accueil

Classes abstraites des tampons

- Classe abstraite `Buffer`
 - factorise les opérations indépendantes du type primitif concerné
- Classe abstraite `ByteBuffer`
 - fournit un ensemble de méthodes et d'opérations plus riche que pour les autres types de buffer
- Classes abstraites dédiées à des types primitifs
 - `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer` et `DoubleBuffer`

Allocation et accès aux tampons

- Allocation de buffer d'un type primitif donné ([prim](#)):
 - méthodes statiques dans les classes [PrimBuffer](#)
 - [PrimBuffer](#) `allocate(int capacity)`
- 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 aléatoire vs séquentiel

- Si PrimBuffer est un buffer d'éléments de type prim :
- Accès **aléatoire** (*absolute*)
 - prim **get**(int index) donne l'élément à la position index
 - primBuffer **put**(int index, prim value) ajoute value à l'indice index, et renvoie le buffer modifié
 - comme `StringBuilder.append()`
 - Peuvent lever `IndexOutOfBoundsException`
- Accès **séquentiel** (*relative*)
 - Prim **get**() resp. primBuffer **put**(prim value)
 - Donne la valeur (resp. place value) à la position courante
 - Peuvent lever des exceptions `BufferUnderflowException` ou `BufferOverflowException`

Les attributs et méthodes d'un tampon

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

Les attributs et méthodes d'un tampon

- **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 à la position.
 - Si la position ou la limite deviennent plus petite que la marque, la marque est effacée
- Invariant :
$$0 \leq \text{marque} \leq \text{position} \leq \text{limite} \leq \text{capacité}$$
- `Buffer rewind()`
 - met la **position** à 0 et supprime la **marque**

Les attributs et méthodes d'un tampon

- Quand la position courante vaut la limite
 - Un appel à `get()` provoque `BufferUnderflowException`
 - Un appel à `put()` provoque `BufferOverflowException`
- Pour éviter ça:
 - `int remaining()` donne le nombre d'éléments entre la position courante et la limite
 - `boolean hasRemaining()` vaut vrai si la position est strictement inférieure à limite

Les attributs et méthodes "en gros"

- Les méthodes existent en version "groupées" (**bulk**)
 - Manipulent un tableau au lieu d'une variable
- primBuffer **get**(prim[] dest, int offset, int length) et primBuffer **get**(prim[] dest)
 - Tentent de lire **le nombre d'éléments spécifié, ou rien** si pas assez de choses à lire ([BufferUnderflowException](#))
- primBuffer **put**(prim[] src, int offset, int length) et primBuffer **put**(prim[] src)
 - Tentent d'écrire **le nombre d'éléments spécifié, ou rien** si pas assez de place ([BufferOverflowException](#))
- primBuffer **put**(primBuffer src)
 - Tente d'écrire le contenu de src ([BufferOverflowException](#))

Exemple de tampon aléatoire

```
final static int SIZE=32;
public static void main(String[] args) {
    // Accès aléatoire (mode tableau)
    IntBuffer iba = IntBuffer.allocate(SIZE);
    for (int i=0; i<SIZE; i++) {
        iba.put(i, 2*i); // Met la valeur 2*i à l'indice i
    }

    for (int i=0; i<SIZE; i++) {
        System.out.println(iba.get(i));
    } // affiche: 0 2 4 ... 62

    System.out.println(iba.get(2));           // affiche 4
    System.out.println(iba.get(31));          // affiche 62
    System.out.println(iba.get(32));
        // lève IndexOutOfBoundsException
    }
}
```

Exemple de tampon séquentiel

```
// Accès séquentiel (mode flot)
IntBuffer ibs = IntBuffer.allocate(SIZE);
// capacity=32 limit=32 position=0 remaining=32
for (int i=0; i<SIZE; i++) {
    ibs.put(2*i);
}
// capacity=32 limit=32 position=32 remaining=0

ibs.rewind(); // remet la position courante à 0
for (int i=0; i<SIZE; i++) {
    System.out.println(ibs.get());
} // affiche: 0 2 4 ... 62

ibs.rewind().limit(2);
for (int i=0; i<SIZE; i++) {
    System.out.println(ibs.get());
} // affiche: 0, 2, puis lève BufferUnderflowException
```

Exemple d'accès "groupé" (*bulk*)

```
char[] t1 = {'a', 'b', 'c', 'd', 'e', 'f'};
// Création tampon de caractères de 6 éléments
CharBuffer cb1 = CharBuffer.allocate(t1.length);
cb1.put(t1);    // recopie en gros de t1 dans le tampon
cb1.position(2); // position de cb1 en 2 (sur 'c')

CharBuffer cb2 = CharBuffer.allocate(cb1.capacity());
cb2.put(cb1);    // recopie {c, d, e, f} de cb1 vers cb2
cb2.limit(cb2.position()); // limite de cb2 après 'f'
cb2.position(0);    // et position en 0

// Alloue un tableau du nbre d'élts à lire dans cb2
char[] t2 = new char[cb2.remaining()];
cb2.get(t2);    // lecture du tampon vers le tableau
for (int i=0; i<t2.length; i++) {
    System.out.println(t2[i]); // Affiche c d e f
}
```

Méthodes utilitaires sur les tampons

➤ compact()

- Place l'élément à la position courante p à la position 0, l'élément $p+1$ à la position 1, etc. La nouvelle position courante est placée après le dernier élément décalé. La limite est mise à la capacité et la marque effacée.

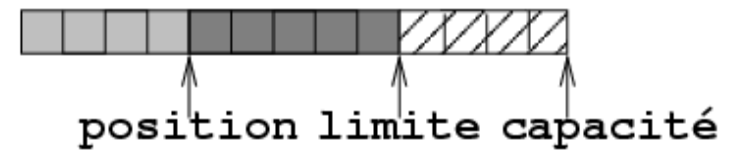
➤ flip()

- $\text{limite} \leftarrow \text{position courante}$
 $\text{position} \leftarrow 0$. Marque indéfinie.

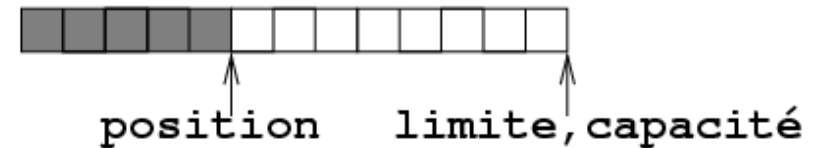
➤ rewind()

- $\text{position} \leftarrow 0$. Marque indéfinie

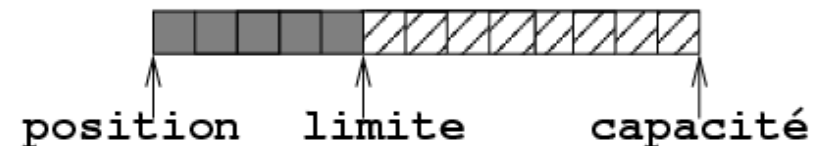
➤ clear() N'efface pas le contenu!



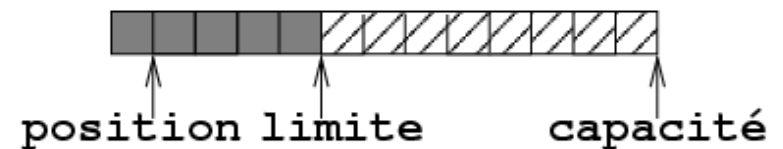
bb.compact();



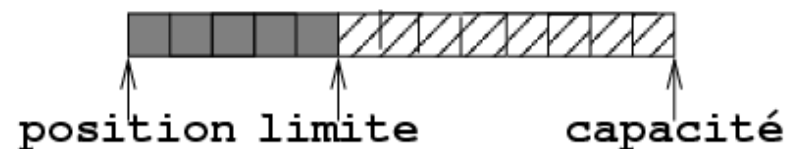
bb.flip();



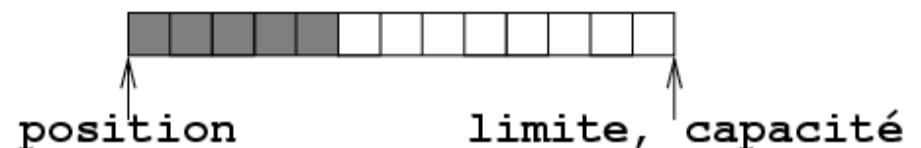
bb.get();



bb.rewind();



bb.clear();



Types primitifs et représentation

- Chaque classe de tampon de prim (type primitif) permet de lire (get) et d'écrire (put) des éléments de type prim
- La classe `ByteBuffer` sait en plus lire et écrire n'importe quel type primitif avec getPrim() ou putPrim()
 - Nécessité de choisir l'ordre de représentation des types primitifs (ordre de stockage des octets)
 - `ByteOrder.BIG_ENDIAN` (gros-boutiste): octet de poids **fort** stocké à l'indice le plus petit (« **network order** »)
 - `ByteOrder.LITTLE_ENDIAN` (petit-boutiste): octet de poids **faible** stocké à l'indice le plus petit

Ordre de représentation des tampons

- `ByteOrder.nativeOrder()` donne l'ordre de stockage natif de la plateforme
- L'ordre d'un `ByteBuffer` peut être consulté ou fixé par `order()`
- Par défaut, tout **tampon d'octet** (`ByteBuffer`) alloué est en **big endian**
- Tous les **autres tampons** sont créés avec l'**ordre natif**
- Les "vues" d'un tampon d'octet ont l'ordre du tampon d'octet au moment de la création de la vue.

Exemple sur les ordres des tampons

```
System.out.println(ByteOrder.nativeOrder()); // LITTLE_ENDIAN
System.out.println(Integer.toBinaryString(134480385));
    // affiche 00001000 00000100 00000010 00000001 ie {8,4,2,1}
ByteBuffer bb = ByteBuffer.allocate(8);
System.out.println(bb.order()); // BIG_ENDIAN
    // On écrit les 4 octets en big endian (8 en 0, 4 en 1...)
bb.put((byte)8).put((byte)4).put((byte)2).put((byte)1);
    // Puis les 4 octets dans l'ordre little endian
bb.put((byte)1).put((byte)2).put((byte)4).put((byte)8);
    // On prend une vue entière big endian de ce tampon d'octet
IntBuffer ibBE = ((ByteBuffer)bb.rewind()).asIntBuffer();
System.out.println(ibBE.order()); // affiche BIG_ENDIAN
System.out.println(ibBE.get()); // affiche 134480385
    // On prend une vue entière en little endian du même tampon
IntBuffer ibLE = bb.order(ByteOrder.LITTLE_ENDIAN).asIntBuffer();
System.out.println(ibLE.order()); // affiche LITTLE_ENDIAN
System.out.println(ibLE.get(1)); // affiche 134480385
System.out.println(ibLE.get(0)); // affiche 16909320...
                                // ... ordre inverse
```

Création d'un tampon à partir d'un tampon (une « vue »)

- `duplicate()` retourne un **tampon partagé**
 - toute modification de données de l'un est vue dans l'autre
 - les attributs (position, limite, marque) du nouveau sont initialisés à partir de l'ancien, mais chaque tampon possède ses propres attributs
- `slice()` retourne un **tampon 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

Création d'un tampon à partir d'un tampon

- `asReadOnlyBuffer()` retourne un **nouveau tampon** (une vue) en **lecture seule**
 - Les méthodes comme `put()` lèvent `ReadOnlyException`
 - Peut être testé avec `isReadOnly()`
- Possibilité de créer des **vues d'un tampon d'octet** (`ByteBuffer`) **comme** s'il s'agissait d'un tampon d'un autre type
 - `asCharBuffer()`, `asShortBuffer()`, `asIntBuffer()`, `asLongBuffer()`, `asFloatBuffer()`, `asDoubleBuffer()`

Exemples de duplication et partage

```
ByteBuffer bb1 = ByteBuffer.allocate(10);
ByteBuffer bb2 = bb1.duplicate(); // 2 accès possibles
for (int i=0; i<bb1.capacity(); i++){
    bb1.put((byte)i);           // remplissage par bb1
}
System.out.println(bb1.position()); // affiche 10
System.out.println(bb2.position()); // affiche 0
bb2.put((byte)3);               // déplace la position de b2 en 1
System.out.println(bb1.get(0));  // affiche 3
System.out.println(bb1.position()); // affiche 10

ByteBuffer bb3 = bb2.asReadOnlyBuffer();
System.out.println(bb3.position()); // affiche 1
bb3.rewind();                     // place la position de b3 en 0
System.out.println(bb3.get());    // affiche 3
bb3.put((byte)4);
// throws java.nio.ReadOnlyBufferException
```

Exemples de duplication et partage

```
char[] t1 = {'a', 'b', 'c', 'd', 'e', 'f'};
CharBuffer cb1 = CharBuffer.allocate(t1.length);
cb1.put(t1);           // met le contenu de t1 dans cb1
cb1.position(2);       // place cb1 prêt à lire 'c'
cb1.limit(5);          // place limite de cb1 après 'e'
System.out.println(cb1.remaining()); // affiche 3

// Crée un tampon partagé pour ce qui reste dans cb1
CharBuffer cb2 = cb1.slice();
System.out.println(cb2.capacity()); // affiche 3
cb2.put('x'); // met 'x' en 0 dans cb2 (remplace 'c')
for (cb1.rewind(); cb1.hasRemaining(); )
    System.out.println(cb1.get()); // affiche a b x d e

cb1.put(4, 'y'); // met 'y' en 4 dans cb1 (remplace 'e')
for (cb2.rewind(); cb2.hasRemaining(); )
    System.out.println(cb2.get()); // affiche x d y
```

Tampons directs et non directs

- Par défaut, `allocate()` renvoie un tampon **non-direct**
 - Allocation classique dans le tas "garbage collecté" (tableau)
 - Le GC peut décider de « déplacer » le tampon qui sert à interagir avec le système: nécessite une copie lors des lectures/écritures
- `ByteBuffer.allocateDirect()` renvoie un tampon **direct** (« zone »)
 - L'objet buffer est géré par le GC mais la mémoire associée est allouée dans un espace non géré par le GC
 - Optimise les lectures/écriture natives (évite de recopier les buffers)
 - Coûteux en allocation/libération
 - À réserver pour les buffers de taille et de durée de vie importantes
- Une "vue" d'un tampon d'octets direct est directe
 - Peut être testé par `isDirect()`

Tampon comme enveloppe de tableau

- Méthode statique `wrap()` dans chaque classe de tampon (pour chaque type primitif) pour envelopper un tableau
 - `primBuffer wrap(prim[] tab, int offset, int length)` ou `primBuffer wrap(prim[] 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

Exemple de tampon enveloppe de tableau

```
int[] it = new int[10]; it[2] = 22; it[4] = 44;
IntBuffer ib = IntBuffer.wrap(it);
System.out.println(ib.capacity());           // affiche 10
System.out.println(ib.get(2));               // affiche 22
ib = IntBuffer.wrap(it, 4, 5);
System.out.println(ib.position());           // affiche 4
System.out.println(ib.limit());              // affiche 9
System.out.println(ib.capacity());           // affiche 10
System.out.println(ib.arrayOffset());        // affiche 0
System.out.println(ib.get());                // affiche 44
                                           // et avance la position

IntBuffer ib2 = ib.slice();
System.out.println(ib2.position());          // affiche 0
System.out.println(ib2.limit());             // affiche 4
System.out.println(ib2.capacity());          // affiche 4
System.out.println(ib2.arrayOffset());       // affiche 5
```

Comparaisons de tampons

- Les classes des tampons implémentent l'interface `Comparable<tamponDeMemeType>`
 - Comparable uniquement avec un tampon du même type
 - `compareTo()` compare les **séquences d'éléments restants** (au sens de `remaining()`) de manière lexicographique (le prochain, puis le suivant, etc.)
- Deux tampons sont égaux au sens de `equals()` si
 - 1. ils ont le **même type** d'éléments,
 - 2. ils ont le **même nombre d'éléments restants** et
 - 3. les deux séquences d'éléments restants, considérées **indépendamment de leurs positions** de départ, sont **égales élément par élément**.

Les tampons de caractères

- Les tampons de caractères `CharBuffer` implantent l'interface `CharSequence`
 - Permet d'utiliser les expressions régulières directement sur les tampons (`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**.

Appendable et Readable (jdk1.5)

- `CharBuffer` implante ces deux 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)`
 - Exemples: `BufferedWriter`, `CharBuffer`, `FileWriter`,
`OutputStreamWriter`, `PrintStream`, `PrintWriter`, `StringBuffer`,
`StringBuilder`, `StringWriter`, `Writer`...
- **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

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
- Référencé par un nom (**canonique**, US-ASCII, ou **alias** ASCII)
- **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.
- **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.

Classe Charset

- Liste de jeux de caractères officiels gérée par IANA:
<http://www.iana.org/assignments/character-sets>
- 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

L'objet encodeur (classe `CharsetEncoder`)

- Obtenu à partir d'un objet `Charset` par `newEncoder()`
 - Caractères d'entrée fournis par un `CharBuffer`
 - Octets produits placée dans un `ByteBuffer`
- 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

- Par défaut, `MalformedInputException` ou `UnmappableCharacterException` sont levées si problème
- On peut spécifier un comportement spécifique
 - `onMalformedInput()` ou `onUnmappableCharacter()`
 - Via 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)
 - Les méthodes `malformedInputAction()` et `unmappableCharacterAction()` donnent la valeur actuelle

Exemple de codage vers ASCII

```
Charset ascii = Charset.forName("ASCII");
CharsetEncoder versASCII = ascii.newEncoder();
CharBuffer cb = CharBuffer.wrap("accentués et J2€€");
// (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 J2$$" en décommentant (a)
} catch(CharacterCodingException cce) {
    cce.printStackTrace();
}
```

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:
 - `isError()` vrai si erreur produite (*malformed* ou *unmappable*)
 - `isUnderflow()` vrai si pas assez de caractères dans `in`
 - `isOverflow()` vrai si pas assez de place dans `out`
 - `isMalformed()` si un caractère mal formé a été rencontré
 - `isUnmappable()` si caractère pas codable dans le jeu de sortie

Méthode `encode()` plus complète (2)

- Le 3^o argument booléen `endOfInput`
 - 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
- L'encodage est terminé lorsque
 - Le dernier appel à `encode()`, avec `endOfInput` à `true`, a renvoyé un `CoderResult` tel que `isUnderflow()` soit `true` (plus rien à lire)
 - Il faut faire `flush()` pour terminer le codage (purge des états internes)

Principe d'utilisation pour codage

- 1. 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
 - On peut traiter ces problèmes éventuels
- 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 utilitaires pour l'encodage

- Étant donné un jeu de caractères
- 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 un encodage est en cours)

Le décodage

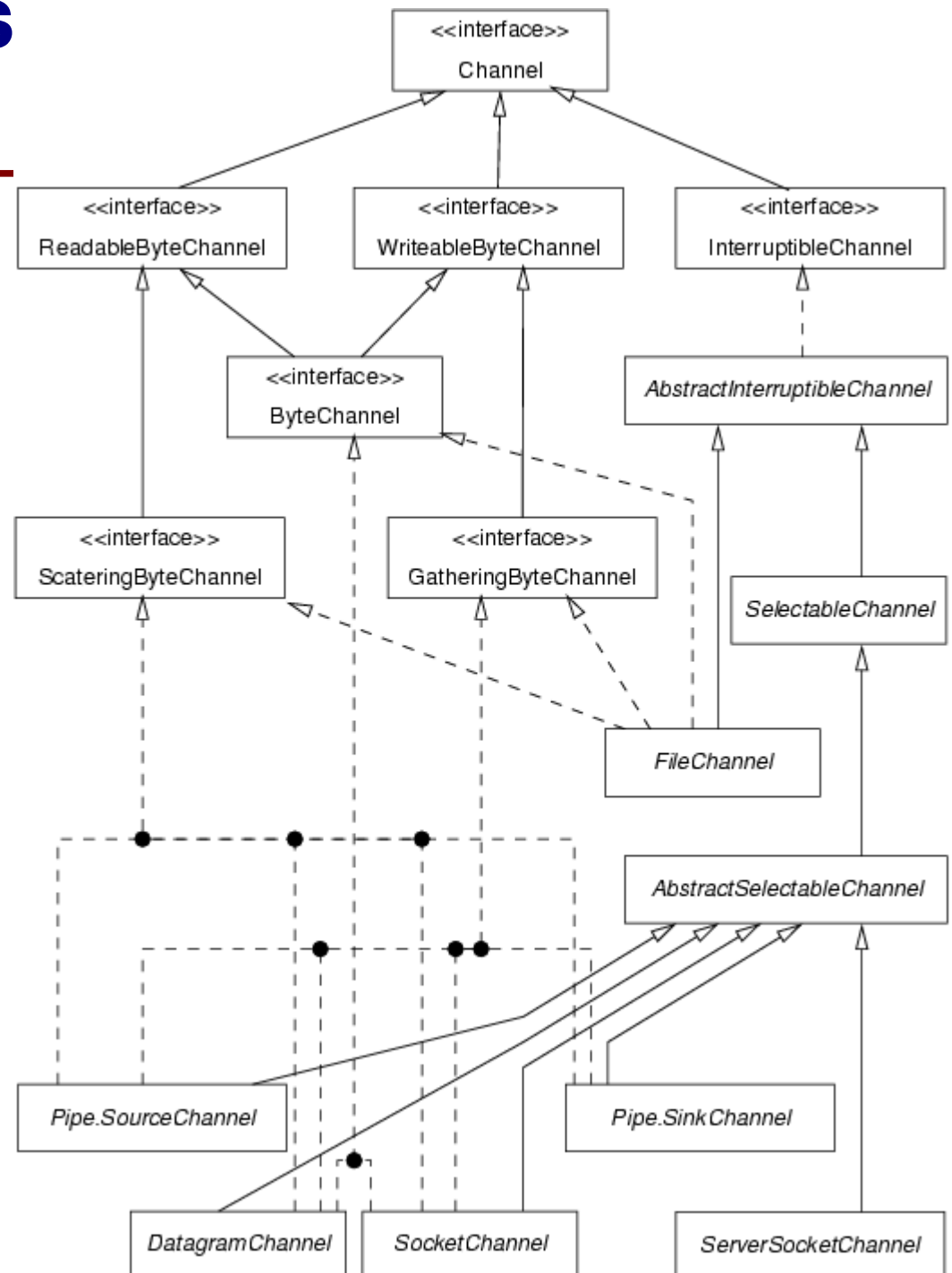
- Principe semblable à celui du codage
 - Instance d'une sous-classe de la classe abstraite `CharsetDecoder`
 - Peut être récupérée par `Charset.newDecoder()`
 - Méthodes `decode()`
 - Lit un tampon d'octets et produit un tampon de caractères
 - Version complète avec `ByteBuffer` d'entrée, `CharBuffer` 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[]`
 - `maxCharsPerByte()` et `averageCharsPerByte()`

Les canaux (*channels*)

- Représentent des connexions ouvertes vers des entités capables d'effectuer des opérations d'entrées-sorties comme des fichiers, des sockets ou des tubes
 - Un canal est ouvert à sa création.
 - Il ne peut plus être utilisé une fois qu'il est fermé.
 - À la différence des flots, il peut être utilisé en mode **bloquant** ou **non bloquant** (*on y reviendra plus tard*).
 - En mode non bloquant, toutes les lectures/écritures retournent immédiatement, même si rien n'est lu ou écrit.
 - On peut alors utiliser un **sélecteur** pour attendre en même temps la possibilité d'effectuer des entrées-sorties sur différents canaux.

Classes/interfaces fondamentales

- `java.nio.channels.*`
 - `Channel`
 - `close()`, `isOpen()`
 - `ReadableByteChannel`
 - `int read(ByteBuffer)`
 - Tente de lire au plus `remaining()` octets.
 - `WritableByteChannel`
 - `int write(ByteBuffer)`
 - Tente d'écrire au plus `remaining()` octets
 - `ByteChannel`
 - Hérite des deux



Canaux et flots

- Quand les canaux sont **bloquants**
 - `read()` et `write()` se comportent comme pour les flots
 - Tous les octets seront écrits au retour de `write()`
 - Au moins un octet lu au retour de `read()` ou alors retourne -1
- Deux lectures ou deux écritures **concurrentes** sur un même canal ont toujours lieu l'une après l'autre
 - Sans interférer entre elles. Évite souvent de synchroniser.
 - En revanche l'interférence entre une lecture et une écriture peut dépendre du type de canal.
- La classe utilitaire **Channels** permet d'obtenir
 - Des canaux à partir de flots et des flots à partir de canaux
 - Ils sont liés:et la fermeture de l'un entraîne celle de l'autre

Exemple : canal associé à une Socket

- `java.nio.SocketChannel` (une connexion TCP)
 - `SocketChannel sc = SocketChannel.open()` puis `sc.connect(InetSocketAddress server) ;`
 - Ou plus directement
 - `SocketChannel sc = SocketChannel.open(InetSocketAddress server) ;`
 - Mais aussi `bind()` et d'autres méthodes « remontées » de `Socket`
 - Plus généralement `sc.socket()` permet de récupérer l'objet de la classe `Socket` sous-jacent

Lecture et écriture dans SocketChannel

- Lecture depuis la connexion
 - `ByteBuffer bb = ...`
`int n = sc.read(bb);`
 - lit au plus `bb.remaining()` octets et retourne ce nombre `n`, ou `-1` si le flot de lecture est fermé
- Écriture dans la connexion
 - `int n = sc.write(bb);`
 - Écrit exactement `bb.remaining()` octets et retourne ce nombre `n`
- Les deux peuvent lever `IOException`

Canal associé à une ServerSocket

- java.nio.ServerSocketChannel (socket serveur d'écoute TCP)
 - **ServerSocketChannel ssc =**
 ServerSocketChannel.open() puis
 ssc.bind(SocketAddress localSocketAddress);
 - Plus généralement **ssc.socket()** permet de récupérer l'objet de la classe **ServerSocket** sous-jacent
- Attente de connexion client :
 - **SocketChannel sc = ssc.accept();**

Interfaces de canaux

- **ScatteringByteChannel extends ReadableByteChannel**
 - Ajoute les méthodes à multiple tampons ("*dispatcheur*")
 - `long read(ByteBuffer[] dsts)` et
`long read(ByteBuffer[] dsts, int offset, int length)`
 - Pratique pour lire des en-têtes de taille fixe (ex: protocoles)
- **GatheringByteChannel extends WritableByteChannel**
 - Ajoute les méthodes ("*collecteur*")
 - `long write(ByteBuffer[] srcs)` et
`long write(ByteBuffer[] srcs, int offset, int length)`
 - Pratique pour toujours débuter une écriture par une suite d'en-têtes spécifiques

Canaux vers les fichiers

- Classe abstraite `FileChannel`
 - Instances obtenues par les méthodes `getChannel()` de
 - `FileInputStream`, `FileOutputStream` ou `RandomAccessFile`
 - Ne supporte que les méthodes correspondantes, sinon `NonWritableChannelException` ou `NonReadableChannelException`
 - Méthodes `read()` et `write()` conformes aux interfaces
 - `read()` retourne -1 quand la fin de fichier est atteinte
 - Deux lectures ou deux écritures concurrentes ne s'entrelassent pas
 - Le flot et le canal sont liés et se reflètent leurs états respectifs
 - Utilisation de caches pour améliorer les performances
 - Méthode `force()` pour écriture des données sur le fichier

Exemple de canaux associés à des fichiers

```
FileInputStream in = new FileInputStream(args[0]);
ReadableByteChannel cin = Channels.newChannel(in);
FileOutputStream out = new FileOutputStream(args[1]);
WritableByteChannel cout = Channels.newChannel(out);
ByteBuffer byteB = ByteBuffer.allocate(1000);
int nb;
try {
    while ((nb=cin.read(byteB)) != -1) {
        byteB.flip();    // position=0, limite=fin des
                        // données lues
        cout.write(byteB);
        byteB.clear(); // position=0, limite=capacité
    }
} finally {
    cin.close();    // Ferme le canal et le flot !
    cout.close();
}
```

FileChannel et transferts optimisés

- **FileChannel** permet des transferts par bloc **optimisés** par le système
 - Lecture d'un bloc depuis un **ReadableFileChannel** et écriture dans le **FileChannel** courant
long **transferFrom**(ReadableByteChannel src, long position, long count)
 - Tente de lire dans `src` au plus `count` bytes et de les copier à partir de `position` dans le canal courant. Ne modifie pas la position courante, mais fait évoluer celle de `src`. Le nombre d'octets transférés est retourné.
 - Écriture d'un bloc depuis le **FileChannel** courant vers un **ReadableFileChannel**
long **transferTo**(long position, long count, WritableByteChannel target)
 - Tente d'écrire dans `target` au plus `count` bytes lus dans le canal courant à partir de `position`. Ne modifie pas la position courante, mais fait évoluer celle de `target`.

Fichiers mappés en mémoire

- Permet de voir un fichier comme un tampon d'octets dans lequel on peut lire ou écrire
 - Opérations beaucoup plus rapides mais
 - Coût pour réaliser le mapping (alloué direct via malloc)
- `MappedByteBuffer extends ByteBuffer`
 - 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
 - `long position`
 - `long taille`
- Le fichier mappé reste valide jusqu'à ce que le `MappedByteBuffer` soit garbage-collecté

Canaux sur les tubes

- Comme les tubes des flots ([java.io](#))
 - [PipedReader](#), [PipedWriter](#), [PipedInputStream](#), [PipedOutputStream](#)
 - Faire communiquer simplement deux processus légers
 - Ce qui est écrit un thread est lu dans l'ordre par l'autre
- À la différence des tubes des flots, ceux des canaux
 - [java.nio.channels](#) : [Pipe](#), [Pipe.SinkChannel](#) et [Pipe.SourceChannel](#) sont des objets qui ont une « réalité système »
- Créés avec la méthode statique [open\(\)](#) de type [Pipe](#)
 - Retourne un objet de type [Pipe](#), sur lequel on peut faire
 - [Pipe.SinkChannel sink\(\)](#) pour écrire des données
([AbstractSelectableChannel](#), [WritableByteChannel](#), [GatheringByteChannel](#))
 - [Pipe.SourceChannel source\(\)](#) pour lire des données
([AbstractSelectableChannel](#), [ReadableByteChannel](#), [ScatteringByteChannel](#))
- Les deux canaux supportent le passage en mode non bloquant

Accès à UDP *via* les canaux

- Depuis jdk 1.4, [java.nio.channels.DatagramChannel](#)
- Canal vers une socket UDP
 - On peut créer une [java.net.DatagramSocket](#) à partir d'un [DatagramChannel](#), mais pas le contraire
 - Si une socket UDP [su](#) a été créée à partir d'un canal, on peut récupérer ce canal par [su.getChannel\(\)](#). Sinon, cette méthode retourne [null](#).
 - [DatagramChannel.open\(\)](#) crée et retourne un canal associé à une socket UDP (non attachée)
 - [DatagramChannel](#) n'est pas une abstraction complète des sockets UDP: pour les opérations précises (binding, etc...) on récupère l'objet [DatagramSocket](#) sous-jacent

DatagramChannel

- Par défaut, un canal `dc` récupéré par `DatagramChannel.open()` est bloquant. Il peut être configuré non bloquant.
 - `dc.socket()` récupère alors la `DatagramSocket` correspondante
 - Elle n'est pas attachée. On peut faire `bind(SocketAddress)` sur cette socket
- Les méthodes `send()` et `receive()` sont accessibles depuis le canal
 - Elles manipulent des `ByteBuffer` et `receive()` retourne un objet `SocketAddress` identifiant l'émetteur des données reçues
- On doit faire une pseudo-connexion pour pouvoir utiliser les méthodes `read()` et `write()` avec des `ByteBuffer`, plus classiques sur les canaux (interlocuteur implicite pour ces méthodes)

Accès à UDP *via* les canaux

- Depuis jdk 1.4, [java.nio.channels.DatagramChannel](#)
- Canal vers une socket UDP
 - On peut créer une [java.net.DatagramSocket](#) à partir d'un [DatagramChannel](#), mais pas le contraire
 - Si une socket UDP [su](#) a été créée à partir d'un canal, on peut récupérer ce canal par [su.getChannel\(\)](#). Sinon, cette méthode retourne [null](#).
 - [DatagramChannel.open\(\)](#) crée et retourne un canal associé à une socket UDP (non attachée)
 - [DatagramChannel](#) n'est pas une abstraction complète des sockets UDP: pour les opérations précises (binding, etc...) on récupère l'objet [DatagramSocket](#) sous-jacent

DatagramChannel

- Par défaut, un canal `dc` récupéré par `DatagramChannel.open()` est bloquant. Il peut être configuré non bloquant.
 - `dc.socket()` récupère alors la `DatagramSocket` correspondante
 - Elle n'est pas attachée. On peut faire `bind(SocketAddress)` sur cette socket
- Les méthodes `send()` et `receive()` sont accessibles depuis le canal
 - Elles manipulent des `ByteBuffer` et `receive()` retourne un objet `SocketAddress` identifiant l'émetteur des données reçues
- On doit faire une pseudo-connexion pour pouvoir utiliser les méthodes `read()` et `write()` avec des `ByteBuffer`, plus classiques sur les canaux (interlocuteur implicite pour ces méthodes)

Envoi sur un DatagramChannel

- `int send(ByteBuffer src, SocketAddress target)`
- Provoque l'envoi des données restantes du tampon `src` vers `target`
- Semblable à un `write()` du point de vue du canal
- Si **canal bloquant**, la méthode retourne lorsque tous les octets ont été émis (leur nombre est retourné)
- Si **canal non bloquant**, la méthode émet tous les octets ou aucun
- Si une autre écriture est en cours sur la socket par une autre thread, l'invocation de cette méthode bloque jusqu'à ce que la première opération soit terminée

Réception depuis un DatagramChannel

- `SocketAddress receive(ByteBuffer dst)`
 - Par défaut, méthode bloquante tant que rien n'est reçu par la socket
 - Au plus `dst.remaining()` octets peuvent être reçus
 - Le reste est tronqué
 - Retourne l'adresse de socket (IP+port) de l'émetteur des données
 - Si **canal non bloquant**, soit tout est reçu, soit le tampon n'est pas modifié et la méthode retourne `null`
 - Bug: on ne peut pas limiter le temps d'attente en lecture, comme c'est le cas avec `DatagramSocket.setSoTimeout()`
 - Dans ce cas, c'est ignoré par le flot de lecture sur le canal
 - Si besoin, il faut le faire « à la main »

Exemple de client UDP avec canaux : envoi

```
// Récupération de l'adresse IP et du port
InetAddress server = InetAddress.getByName(args[0]);
int port = Integer.parseInt(args[1]);
InetSocketAddress isa = new InetSocketAddress(server, port) ;

// Création d'un objet canal UDP non attaché
DatagramChannel dc = DatagramChannel.open() ;

// Les données à envoyer doivent être dans un ByteBuffer
ByteBuffer bb = ByteBuffer.wrap("Hello".getBytes("ASCII")) ;

// L'attachement de la socket UDP sous-jacente est implicite
dc.send(bb,isa) ;

// si j'attends une réponse, je ne ferme pas le canal...
```

Exemple de client UDP avec canaux : réception

```
// pour me préparer à recevoir la réponse...
// j'alloue une zone de données de taille suffisante
ByteBuffer bb = ByteBuffer.wrap(new byte[512]);

// la réception place les données dans la zone de données et
// retourne la socket représentant l'émetteur
SocketAddress sender = dc.receive(bb) ;

bb.flip();
System.out.println(bb.remaining()+" octets ont été reçus de "
                    +sender+":");
System.out.println(new String(bb.array(),
                               0,bb.remaining(),
                               "ASCII"));

// L'échange est fini: je ferme (définitivement) le canal...
dc.close();
```

Exemple de serveur UDP avec canaux

```
// Création d'un objet canal UDP (non attaché)
DatagramChannel dc = DatagramChannel.open() ;

// attachement (explicite) de la socket sous-jacente
// au port d'écoute
InetSocketAddress isa = new InetSocketAddress(port);
dc.socket().bind(isa) ;

// allocation d'une zone de données de taille suffisante
ByteBuffer bb = ByteBuffer.allocateDirect(512);
while (true) {
    SocketAddress sender = dc.receive(bb); // réception
    bb.flip() ;
    // traitement éventuel
    dc.send(bb, sender); // réponse à l'émetteur
    bb.clear() ;
    // raz de la zone de réception
}
```

Canaux fermables et "interruptibles"

- Interface `InterruptibleChannel` extends `Channel`
- Un canal qui implante `InterruptibleChannel` est un
 - Canal ***fermable de manière asynchrone***
 - Si une thread `t` est bloquée par une opération d'entrée/sortie sur un tel canal, et qu'une autre thread ferme le canal par `close()`, alors `t` reçoit une exception `AsynchronousCloseException`
 - Canal ***interruptible***
 - Si une thread `t` est bloquée par une opération d'entrée/sortie sur un tel canal, et qu'une autre thread fait `t.interrupt()`, alors le canal est fermé et `t` reçoit une exception `ClosedByInterruptException`
 - Dans ce cas, le statut d'interruption de `t` est positionné
 - Si le statut d'interruption de `t` est positionné au moment de l'appel une opération d'entrée/sortie bloquante, ça lève aussi `ClosedByInterruptException`
 - Statut d'interruption reste positionné, le canal est fermé

Canaux à mode non bloquant

- Héritent de la classe abstraite [SelectableChannel](#)
 - Lecture et écriture ne bloquent jamais
 - Le nombre d'octets transférés peut être inférieur à l'indication
 - Éventuellement nul, en lecture comme en écriture
- Tous les canaux standards peuvent être non bloquants SAUF:
 - Ceux obtenus par la classe utilitaire [Channels](#)
 - Les canaux sur les fichiers
- À leur création, les canaux sont en mode bloquant
 - Changement de mode par [configureBlocking\(false\)](#)
 - Consultation du mode actuel par [isBlocking\(\)](#)
- Ils sont les seuls à pouvoir être multiplexés avec un **sélecteur** de la classe [java.nio.channels.Selector](#)

ServerSocketChannel

- Appeler `accept()` sur le canal pour accepter des connexions (la server socket sous-jacente doit être attachée)
 - Si le canal est **bloquant**, l'appel bloque
 - Retourne un `SocketChannel` quand la connexion est acceptée ou
 - Lève une `IOException` si une erreur d'entrée-sortie arrive
 - Si le canal est **non bloquant**, retourne immédiatement
 - `null` s'il n'y a pas de connexion pendant
 - Quel que soit le mode (bloquant / non bloquant) du `ServerSocketChannel` le `SocketChannel` retourné est **initialement en mode bloquant**

SocketChannel

- `boolean connect(SocketAddress remote)` sur le canal
- Si `SocketChannel` en mode **bloquant**, l'appel à `connect()` bloque et retourne `true` quand la connexion est établie, ou lève `IOException`
- Si `SocketChannel` en mode **non bloquant**, l'appel à `connect()`
 - Peut retourner `true` immédiatement (connexion locale, par exemple)
 - Retourne `false` le plus souvent: il faudra plus tard appeler `finishConnect()`
- Tant que le canal est non connecté, les opérations d'entrée/sortie lèvent `NotYetConnectedException` (peut être testé par `isConnected()`)
- Un `SocketChannel` reste connecté jusqu'à ce qu'il soit fermé
 - Possibilité de faire des half-closed et des fermetures asynchrones
 - `AsynchronousCloseException` (si canal fermé alors qu'une opération d'écriture en cours était bloquée, ou pendant l'opération de connection)
 - `ClosedByInterruptException` (si autre thread interrompt la thread courante lorsqu'elle effectue une opération, IO, connection, etc.)

Établissement de connexion

- Méthode `finishConnect()`
 - Si connexion a échoué, lève `IOException`
 - Si pas de connexion initiée, lève `NoConnectionPendingException`
 - Si connexion déjà établie, retourne immédiatement `true`
 - Si la connexion n'est pas encore établie
 - Si mode **non bloquant**, retourne `false`
 - Si mode **bloquant**, l'appel bloque jusqu'au succès ou à l'échec de la connexion (retourne `true` ou lève une exception)
 - Si cette méthode est invoquée lorsque des opérations de lecture ou d'écriture sur ce canal sont appelés
 - Ces derniers sont bloqués jusqu'à ce que cette méthode retourne
 - Si cette méthode lève une exception (la connexion échoue)
 - Alors le canal est fermé

Communication sur canal de socket TCP

- Une fois la connexion établie, le canal de socket se comporte comme un canal en lecture et écriture
 - `extends AbstractSelectableChannel` et `implements ByteChannel, ScatteringByteChannel, GatheringByteChannel`
- Méthodes `read()` et `write()`
 - En mode **bloquant**, `write()` assure que tous les octets seront écrits mais `read()` n'assure pas que le tampon sera rempli (au moins un octet lu ou détection de fin de connexion: retourne -1)
 - En mode **non bloquant**, lecture comme écriture peuvent ne rien faire et retourner 0.
- Fermeture de socket par `close()` entraine la fermeture du canal

Les sélecteurs

- Objets utilisés avec les canaux configurés en mode **non bloquant** qui héritent de `SelectableChannel`
 - Ces canaux peuvent être enregistrés auprès d'un sélecteur après avoir été configurés non bloquant (`configureBlocking(false)`)
 - `java.nio.channels.Selector` : les instances sont créées par appel à la méthode statique `open()` et fermés par `close()`

Les sélecteurs (suite)

- Enregistrement d'un canal auprès d'un sélecteur
 - Se fait par un appel, sur le canal à enregistrer, de:
 - `SelectionKey register(Selector sel, int ops)` ou `SelectionKey register(Selector sel, int ops, Object att)`
 - `ops` représente les opérations "intéressantes" pour ce sélecteur
 - `SelectionKey` retourné représente la **clé de sélection** de ce canal, qui permet de connaître, outre le sélecteur:
 - `channel()` renvoie le canal lui même
 - `interestOps()` renvoie les opérations « intéressantes »
 - `attachment()` renvoie l'objet `att` éventuellement attaché

Autour des sélecteurs

- `keys()` appelé sur un sélecteur retourne toutes ses clés
 - `SelectionKey keyFor(Selector sel)` sur un canal donne la clé de sélection du sélecteur pour ce canal
- On peut enregistrer plusieurs fois un même canal auprès d'un sélecteur (il met la clé à jour)
 - Il est plus élégant de modifier sa clé de sélection
 - En argument de `interestOps()` ou de `attach()` sur cette clé de sélection
- Les opérations intéressantes sont exprimées par les bits d'un entier (faire des OU binaires (|))
 - `SelectionKey.OP_READ`, `SelectionKey.OP_WRITE`, `SelectionKey.OP_CONNECT`, `SelectionKey.OP_ACCEPT`
 - Étant donné un canal, `validOps()` renvoie ses opérations « valides »

Utilisation des sélecteurs

- Appel à la méthode `select()`
 - Entraîne l'attente passive d'événements intéressants pour les canaux enregistrés
 - Dès qu'une de ces opérations peut être effectuée, la méthode retourne le nombre de canaux sélectionnés
 - Elle ajoute également les clés de sélection de ces canaux à l'ensemble retourné par `selectedKeys()` sur le selecteur
 - Il suffit de le parcourir avec un itérateur
 - C'est à l'utilisateur de retirer les clés de sélection correspondant aux canaux sélectionnés qu'il a « utilisé »
 - méthode `remove()` de l'ensemble ou de l'itérateur ou
 - méthode `clear()` de l'ensemble qui les retire toutes
- Si une clé est intéressée par plusieurs opérations
 - `readyOps()` donne celles qui sont prêtes
 - raccourcis `isAcceptable()`, `isConnectable()`, `isReadable()` et `isWritable()`

Sélection bloquante ou non bloquante

- Les méthodes `select()` ou `select(long timeout)` sont bloquantes
 - Elles ne retournent qu'après que
 - un canal soit sélectionné ou
 - la méthode `wakeup()` soit appelée ou
 - la thread courante soit interrompue ou
 - le `timeout` ait expiré
- La méthode `selectNow()` est non bloquante
 - Retourne 0 si aucun canal n'est sélectionné
- La méthode `wakeup()` permet d'interrompre une opération de sélection bloquante, depuis un autre processus léger
- L'annulation de l'enregistrement d'un canal auprès d'un sélecteur peut se faire par `cancel()` sur la clé de sélection. Elle est aussi réalisée implicitement à la fermeture du canal.

Fonctionnement des sélecteurs

- Un sélecteur maintient 3 ensembles de clés de sélection
 - **key set**
 - Ensemble des clés de tous les canaux enregistrés dans ce sélecteur.
 - Méthode `keys()`
 - **selected-key set**
 - Ensemble des clés dont les canaux ont été identifiés comme **prêts**
 - pour au moins une des opérations spécifiées dans l'ensemble des opérations de cette clé
 - à l'occasion d'une **opération de sélection précédente**
 - Méthode `selectedKeys()`
 - **canceled-key set**
 - Ensemble des clés qui ont été annulées mais dont les canaux n'ont pas encore été désenregistrés de ce sélecteur
 - Cet ensemble n'est pas directement accessible

Opération de sélection

- 1. Chaque clé de ***canceled-key set*** est supprimée
 - De chacun des ensembles dans lesquels elle est présente
 - À la fin de 1. *canceled-key set* est vide
- 2. L'OS est interrogé
 - Pour savoir quels canaux sont prêts à réaliser une opération qui est décrite par sa clé comme "d'intérêt" au début de l'opération de sélection
 - Pour chacun des canaux prêts à faire quelque chose
 - Soit sa clé n'était pas déjà dans *selected-key set*, alors elle y est ajoutée avec un ensemble d'opérations prêtes reflétant l'état du canal
 - Soit sa clé était déjà dans *selected-key set*, alors son ensemble d'opérations prêtes est mis à jour par un OU binaire avec les anciennes
- 3. Si, pendant l'étape 2, des clés ont été ajoutée dans *canceled-key set*, alors elles sont traitées comme dans 1.

Sélection et concurrence

- Les sélecteurs sont "thread-safe", mais pas leurs ensembles
- Une opération de sélection synchronise:
 - 1. sur le sélecteur
 - 2. sur le *key-set*
 - 3. sur le *selected-key set*
 - Et sur le *canceled-key set* pendant les étapes 1. et 3. de la sélection
- Un thread bloqué sur `select()` ou `select(long timeout)` peut être interrompu par un autre thread de trois manières:
 - En appelant `wakeup()` sur le sélecteur (`select()` retourne)
 - En appelant `close()` sur le sélecteur (`select()` retourne)
 - En appelant `interrupt()` sur le thread
 - Pose le statut d'interruption et appelle la méthode `wakeup()`

Code-type d'un serveur non bloquant

```
public void launch() throws IOException {
    // registers the server socket for 'accept' operations
    serverSocket.register(selector, SelectionKey.OP_ACCEPT);
    // retrieves the selectedKeys set reference
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    while(!Thread.interrupted()) {
        // blocking selection operation, until there is something to do
        selector.select();

        for(SelectionKey key : selectedKeys) {
            if(key.isAcceptable()) {
                doAccept(key);
            }
            if(key.isValid() && key.isWritable()) {
                doWrite(key);
            }
            if(key.isValid() && key.isReadable()) {
                doRead(key);
            }
        }
    }
}
```