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
- 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

- Utilisés par les primitives d'entrées-sorties
  - « remplace » les tableaux de java.io
  - Zone de mémoire contiguë, permettant de stocker
    - une quantité de donnée fixée,
    - d'un type primitif donné
  - Pas prévus pour accès concurrent (il faut synchroniser)
- Représentés par des classes abstraites
  - Classe abstraite Buffer, qui factorise les opérations indépendantes du type primitif concerné
  - Classe abstraite ByteBuffer, dont l'ensemble de méthodes et d'opérations est le plus complet
  - Classes abstraites CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer et DoubleBuffer pour les autres types primitifs

Accès aux tampons

- Pour l'allocation: méthode statique allocate(int capacity)
- Accès **aléatoire** (*absolute*), à un indice (genre tableau)
  - <*tp*> **get**(int index) où <*tp*> est le type primitif du tampon
  - <*tp*>Buffer **put**(int index, <*tp*> value)
  - Peuvent lever IndexOutOfBoundsException
- Accès **séquentiel** (*relative*), genre flot, par rapport à la **position courante** (indice du prochain élément à lire ou à écrire)
  - <*tp*> **get**() et <*tp*>Buffer **put**(<*tp*> value)
  - Peuvent lever des exceptions BufferUnderflowException ou BufferOverflowException

Les attributs et méthodes d'un tampon (1)

- **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. Peut être fixée par un appel à Buffer limit(int newLimit) et connue par un appel à int limit()
- **Position courante**: indice du prochain élément accessible
  - Consultable/modifiable: int position() et Buffer position(int newPosition)
- **Marque** (éventuelle): position dans le tampon
  - Appel à Buffer mark() place la marque à la position courante et appel à Buffer reset() place la position à la marque (ou InvalidMarkException)
  - Toujours inférieure à la position. Si la position ou la limite deviennent plus petite que la marque, la marque est effacée
- Invariant: **0 <= marque <= position <= limite <= capacité**
- Appel à Buffer rewind() met *position* à 0 et supprime *marque*

Les attributs et méthodes d'un tampon (2)

- Quand la position courante vaut la limite
  - Un appel à get() provoque BufferUnderflowException et un appel à put() provoque BufferOverflowException
    - Pour éviter ça:
      - int remaining() donne le nombre d'éléments entre la position et la limite
      - boolean hasRemaining() dit si la position strictement inférieure à limite
- Les méthodes existent en version « par bloc » (*bulk*):
  - <*tp*>Buffer **get**(<*tp*>[] dest, int offset, int length) et <*tp*>Buffer **get**(<*tp*>[] dest) pour lire **le nombre d'éléments spécifié ou rien** (BufferUnderflowException)
  - <*tp*>Buffer **put**(<*tp*>[] src, int offset, int length) et <*tp*>Buffer **put**(<*tp*>[] src) pour écrire **le nombre d'éléments spécifié ou rien** (BufferOverflowException)
  - Version de tampon à tampon: <*tp*>Buffer **put**(<*tp*>Buffer src)

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 par bloc (*bulk*)

```
char[] t1 = {'a', 'b', 'c', 'd', 'e', 'f'};
// Création d'un tampon de caractères de 6 éléments
CharBuffer cb1 = CharBuffer.allocate(t1.length);
cb1.put(t1);       // recopie « en bloc » de t1 dans le tampon
cb1.position(2);  // positionne le tampon en 2 (prêt à lire 'c')
CharBuffer cb2 = CharBuffer.allocate(cb1.capacity());
cb2.put(cb1);     // recopie en bloc {c, d, e, f} de cb1 -> cb2
cb2.limit(cb2.position()); // place la limite de cb2 après 'f'
cb2.position(0);
// Alloue un tableau du nbre d'éléments de cb2
char[] t2 = new char[cb2.remaining()];
cb2.get(t2);  // lecture « en bloc » du tableau depuis tampon
for (int i=0; i<t2.length; i++) {
  System.out.println(t2[i]); // Affiche c d e f
}
```

Allocation d'un tampon à partir d'un tampon existant

- 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 sont propres à chaque tampon
- 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
- asReadOnlyBuffer() retourne un nouveau tampon en lecture
  - Les méthodes comme put() lèvent ReadOnlyException
  - Peut être testé avec isReadOnly()
- Possibilité de créer des « vues » d'autre type sur ByteBuffer
  - asCharBuffer(), asShortBuffer(), asIntBuffer(), asLongBuffer(), asFloatBuffer(), asDoubleBuffer()

Exemples de duplication et partage (1)

```
ByteBuffer bb1 = ByteBuffer.allocate(10); // tampon d'octets
  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);                   // place la position 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 en 0
  System.out.println(bb3.get());      // affiche 3
  bb3.put((byte)4);
                // throws java.nio.ReadOnlyBufferException
```


Exemples de duplication et partage (2)

```
char[] t1 = {'a', 'b', 'c', 'd', 'e', 'f'};
  CharBuffer cb1 = CharBuffer.allocate(t1.length);
  cb1.put(t1);       // met le contenu du tableau dans le tampon
  cb1.position(2);  // place cb1 prêt à lire 'c'
  cb1.limit(5);      // place la limite de cb1 après 'e'
  System.out.println(cb1.remaining());    // affiche 3
  // crée un tampon partagé pour ce qui reste à lire dans cb1
  CharBuffer cb2 = cb1.slice();
  System.out.println(cb2.capacity()); // affiche 3
  cb2.put('x');     // place 'x' en 0 dans cb2, i.e. à la place de 'c'
  for (cb1.rewind(); cb1.hasRemaining(); )
    System.out.println(cb1.get());        // affiche a b x d e
  cb1.put(4,'y');  // place 'y' en 4 dans cb1, i.e. à la place de 'e'
  for (cb2.rewind(); cb2.hasRemaining(); )
    System.out.println(cb2.get());        // affiche x d y
```


Tampons directs et non directs

- Par défaut, allocate() renvoit un tampon **non-direct**
  - Allocation classique dans le tas « garbage collecté »
- ByteBuffer.allocateDirect() renvoit un tampon **direct**
  - La mémoire associée peut être réservée en dehors du tas classique (mémoire interne de la JVM) afin d'optimiser les opérations de lecture et d'écriture natives (éviter de recopier les buffers)
  - Coûteux en allocation et déallocation
  - À réserver pour les tampons d'entrées-sorties de taille et de durée de vie importantes.
- Une « vue » d'un tampon d'octet direct est directe
  - Peut être testé par isDirect()

Tampon comme enveloppe de tableau

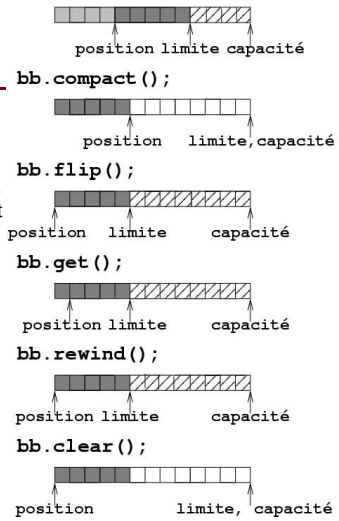
- Méthode statique wrap() dans chaque classe de tampon (pour chaque type primitif) pour envelopper un tableau
  - <*tp*>Buffer **wrap**(<*tp*>[] tab, int offset, int length) ou <*tp*>Buffer **wrap**(<*tp*>[] tab)
    - Enveloppe la totalité du tableau: la capacité vaut toujours tab.length
    - La position du tampon produit est mise à offset (sinon 0)
    - Sa limite est mise à offset+length (sinon tab.length)
  - Position à 0, limite et capacité à la taille du tableau enveloppé
  - 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
ib = ib.slice();
System.out.println(ib.position());      // affiche 0
System.out.println(ib.limit());         // affiche 4
System.out.println(ib.capacity());      // affiche 4
System.out.println(ib.arrayOffset()); // affiche 5
```


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()
  - limite <- position courante position <- 0. Marque indéfinie.
- rewind()
  - position <- 0. Marque indéfinie
- clear()
  - N'efface pas le contenu!



Comparaisons de tampons

- Les classes des tampons implémentent l'interface Comparable
  - 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.

Types primitifs et représentation

- Chaque classe de tampon de <*tp*> (type primitif) permet de lire (get) et d'écrire (put) des éléments de type <*tp*>
- La classe ByteBuffer sait en plus lire et écrire n'importe quel type primitif avec get<*tp*>() ou put<*tp*>()
  - 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
  - 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 i.e. {8,4,2,1}
ByteBuffer bb = ByteBuffer.allocate(8);
System.out.println(bb.order());  // BIG_ENDIAN
// On écrit les 4 octets dans l'ordre big endian
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
```


Les tampons de caractères

- Les tampons de caractères CharBuffer implantent l'interface CharSequence
  - 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 un char[] ou n'importe quel CharSequence : String, StringBuffer ou CharBuffer
    Dans ces derniers cas, le tampon obtenu est en ***lecture seule***.

Les jeux de caractères (*charsets*)

- Des classes dans le paquetage 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
    - Chargé de transformer 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
    - Opération inverse. À partir d'une séquence d'octets représentant des caractères dans un jeu de caractères donné, doit produire 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
  - Méthode statique 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
- Méthode statique isSupported() dit si la JVM supporte le jeu de caractères dont le nom est passé en argument
- Méthode statique forName() donne le Charset associé au nom fourni en argument
- 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

- De classe CharsetEncoder
  - Obtenu à partir d'un objet Charset par newEncoder()
  - Séquence de caractères d'entrée fournie par un tampon de caractères
  - Séquence d'octets produits placée dans un tampon d'octets
  - Méthode encode() la plus simple
    - Accepte un CharBuffer et encode son contenu (*remaining*) dans un ByteBuffer alloué pour l'occasion
    - Peut lever IllegalStateException si une opération de codage est déjà en cours ou bien une CharacterCodingException qui peut être:
      - MalformedInputException si une valeur d'entrée est incorrecte
      - UnmappableCharacterException si l'élément 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, les exceptions MalformedInputException ou UnmappableCharacterException sont levées si problème
- Il est possible de spécifier, par onMalformedInput() ou onUnmappableCharacter(), un comportement spécifique pour un encodeur, sous la forme d'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 (correspond par défaut à '?')
    - byte[] replacement() permet de la consulter
    - replaceWith(byte[]) permet d'en spécifier une nouvelle
  - REPORT provoque la levée d'exception (par défaut)
- Les comportements sont récupérables par les méthodes malformedInputAction() et unmappableCharacterAction()

Exemple de codage vers ASCII

```
Charset ascii = Charset.forName("ASCII");
CharsetEncoder versASCII = ascii.newEncoder();
CharBuffer cb = CharBuffer.wrap("caractères accentués et J2€€");
// (a)
// versASCII.replaceWith(new byte[]{'$'});
// versASCII
//      .onUnmappableCharacter(CodingErrorAction.REPLACE);
try {
  ByteBuffer bb = versASCII.encode(cb);
        // lève UnmappableCharacterException avec le commentaire (a)
  while (bb.hasRemaining()) {
    System.out.print(((char)bb.get()));
  }      // affiche "caract$res 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 et écrit au plus out.remaining() octets dans out, en faisant évoluer les positions
  - Retourne un objet qui représente un code de retour de la classe CoderResult qui peut être testé
    - isError() si une erreur s'est produite (*malformed* ou *unmappable*)
    - isUnderflow() si il n'y avait pas assez de caractères dans in
    - isOverflow() si il n'y avait plus de place dans out
    - isMalformed() si un caractère mal formé a été rencontré
    - isUnmappable() si un caractère n'est pas codable dans le jeu de sortie
  - L'argument endOfInput à false indique que d'autres caractères suivent
    - L'état interne du codeur peut les attendre
    - Le dernier appel à cette fonction doit avoir endOfInput à true
  - Après le dernier appel à encode(), avec endOfInput à true et tel que isUnderflow() sur le résultat soit true (plus rien à lire)
    => Faire flush() pour terminer le codage (purge des états internes)

Principe d'utilisation pour codage

1. Remettre à jour l'encodeur avec reset()
   - Le codage nécessite un ensemble d'états (#octets *vs* #caractères)
   - maxBytesPerChar() et averageBytesPerChar() retournent des float
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 entre chaque invocation
   - Cette méthode ne retourne que lorsqu'il n'y a plus rien à lire, plus de place pour écrire ou qu'un problème de codage est survenu
     - 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()
   - Pour purger les états internes de l'encodeur dans le buffer de sortie

Le décodage

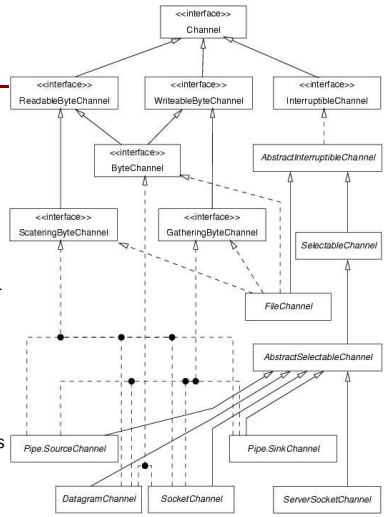
- Principe semblable à celui du codage
  - Instance d'une sous-classe de la classe abstraite CharsetDecoder
    - Peut être récupérée par Charset.newDecoder()
  - Méthodes decode()
    - Lit un tampon d'octets et produit un tampon de caractères
    - Version complète avec ByteBuffer d'entrée, CharBuffer 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.
  - 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 et de placer dans le buffer au plus remaining() octets.
  - WritableByteChannel
    - int write(ByteBuffer)
    - Tente d'écrire au plus remaining() octets depuis le buffer
  - 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 -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 la synchronisation.
    - 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 de canaux associés à des flots

```
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 données lues
    cout.write(byteB);
    byteB.clear(); // position=0, limite=capacité
  }
} finally {
  cin.close();     // Ferme le canal et le flot !
  cout.close();
}
```

Interfaces de canaux

- ScatteringByteChannel extends ReadableByteChannel
  - Ajoute les méthodes à multiple tampons
    - 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
    - long write(ByteBuffer[] srcs) et
      long write(ByteBuffer[] srcs, int offset, int length)
    - Pratique pour toujours débuter une écriture par un en-tête spécifique
- InterruptibleChannel extends Channel
  - Canal *fermable* et *interruptible* de manière asynchrone
    - AsynchronousCloseException reçu par threads bloqués sur I/O si close()
    - ClosedByInterruptException reçu par threads bloqués sur I/O si interrupt()
      - Statut d'interruption positionné dans ce cas

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

Canaux vers les fichiers

- Classe abstraite FileChannel
  - Instances obtenues par les méthodes getChannel() de
    - FileInputStream, FileOutputStream ou RandomAccessFile
    - Ne supporte que les méthodes correspondantes, i.e. peuvent lever
      - NonWritableChannelException ou NonReadableChannelException
  - Les 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
  - Utilisation de caches pour améliorer les performances
    - Méthode force() contraint l'écriture des données dans le fichier
  - Le flot et le canal sont liés et se reflètent leurs états respectifs

Fichiers mappés en mémoire

- Permet de voire un fichier comme un tampon d'octets dans lequel on peut lire ou écrire
  - Opérations beaucoup plus rapides mais
  - Coût pour réaliser le mapping
- MappedByteBuffer extends ByteBuffer
  - Obtenu par la méthode map() sur un FileChannel
    avec comme arguments
    - FileChanel.MapMode
      - READ_ONLY, le 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 récupéré par le ramasse-miettes

Canaux sur les tubes

- Comme les tubes des flots (java.io)
  - PipedReader, PipedWriter, PipedInputStream, PipedOutputStream
  - Permettent de faire communiquer simplement deux processus légers
  - Tout ce qui est écrit dans un tube par un thread est lu dans l'ordre par l'autre thread
- À la différence des tubes des flots
  - 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 appeler
    - 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

Canaux vers les sockets UDP

- 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 a été créée à partir d'un canal, on peut récupérer ce canal par getChannel(). Sinon, retourne null.
  - DatagramChannel.open() crée et retourne un canal associé à une socket UDP (non attachée)
    - Par défaut, le canal est bloquant. Peut être configuré non bloquant.
    - socket() récupère alors la DatagramSocket correspondante
      - Elle n'est pas attachée. On peut faire bind(SocketAddress) sur la socket
    - Les méthodes send() et receive() sont accessibles sur le canal
    - On doit faire une pseudo-connexion pour pouvoir utiliser les méthodes read() et write() classiques du canal

Envoi/réception sur canal de socket UDP

- int send(ByteBuffer src, SocketAddress target)
  - Provoque l'envoit 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
- SocketAddress receive(ByteBuffer dst)
  - Par défaut, 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 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

Canaux vers les sockets TCP

- java.nio.channels ServerSocketChannel et SocketChannel
  - Acceptation de connexion et représentation des connexions
- ServerSocketChannel
  - Objet canal d'écoute de connexions entrantes retourné par open()
  - N'est que l'abstraction d'un canal sur une ServerSocket, récupérable par la méthode socket()
    - Il faut attacher cette ServerSocket par bind() avant de pouvoir accepter des connexions (sinon, NotYetBoundException)
  - Appeler accept() sur le canal pour accepter des connexions
    - 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 pendante
    - Quel que soit le mode (bloquant / non bloquant) du ServerSocketChannel le SocketChannel retourné est initialement en mode bloquant

SocketChannel

- SocketChannel.open() crée un canal de socket non connecté
  - Attachement éventuel de la socket sous-jacente Socket socket() Par un appel à bind() sur cet objet Socket
- 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()
  - Si canal non connecté,opération d'IO lève 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 (fermé pendant l'opération de connexion)
      - ClosedByInterruptException (interrompu pendant l'opération de connexion)

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

- Utilisés avec les canaux en mode non bloquant qui héritent de SelectableChannel
  - Ceux-ci peuvent être enregistrés auprès d'un sélecteur
    - Après avoir été configurés non bloquant (configureBlocking(false))
  - java.nio.channels.Selector dont les instances sont créées par appel à la méthode statique open() et fermés par close()
  - 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
        - channel() renvoit le canal lui même
        - interestOps() renvoit les opérations « intéressantes »
        - attachment() renvoit l'objet 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 à jour la clé) mais 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() renvoit ses opérations « valides »

Utilisation des sélecteurs

- Appel à la méthode select()
  - Entraine 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 séléctionnés
  - 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 prêt 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()

Exemples

- Serveur TCP de mise en majuscule utilisant des canaux
  - Interrogeable avec netcat ou telnet
  - Différentes versions
- Itératif avec canaux bloquants
  - 1 thread
- Concurrent avec canaux bloquants
  - N threads
- Acceptation de nouvelles connexions bloquante, mais lecture non bloquante sur toutes les connexions établies
  - 2 threads
- Acceptation et lectures non bloquantes
  - 1 thread
  - D'abord sans, puis avec mécanisme d'attachement de gestionnaire