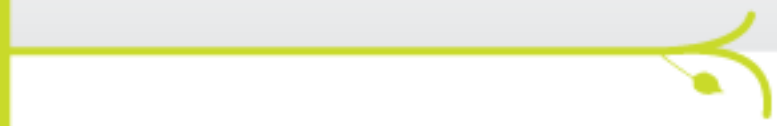
Programmation Java Avancée

La programmation des communications distantes

Partie 3: NIO



jnesis

plan

- IO et NIO
- Les Buffers
- Les Channels
- Les Selectors
- Les Charset
- Les Expressions régulières

New Input/Output API

NIO

- Jusqu'à la version 1.4 toutes les opérations d'entrée/sortie en java étaient de type « flux » (stream)
- Ce fonctionnement oblige la gestion des entrées/sortie octet par octet
- Lors du traitement de grands volumes de données, ce mode de fonctionnement est rapidement très limité (performances...)

NIO

- NIO ou New Input / Output est une API Java introduite au sein du JDK 1.4 (JSR 51)
- Contrairement à l'API IO, l'API NIO propose un fonctionnement en mode « bloc » de données
- NIO essaie dès que possible de s'appuyer sur les l'API IO natives du système

NIO

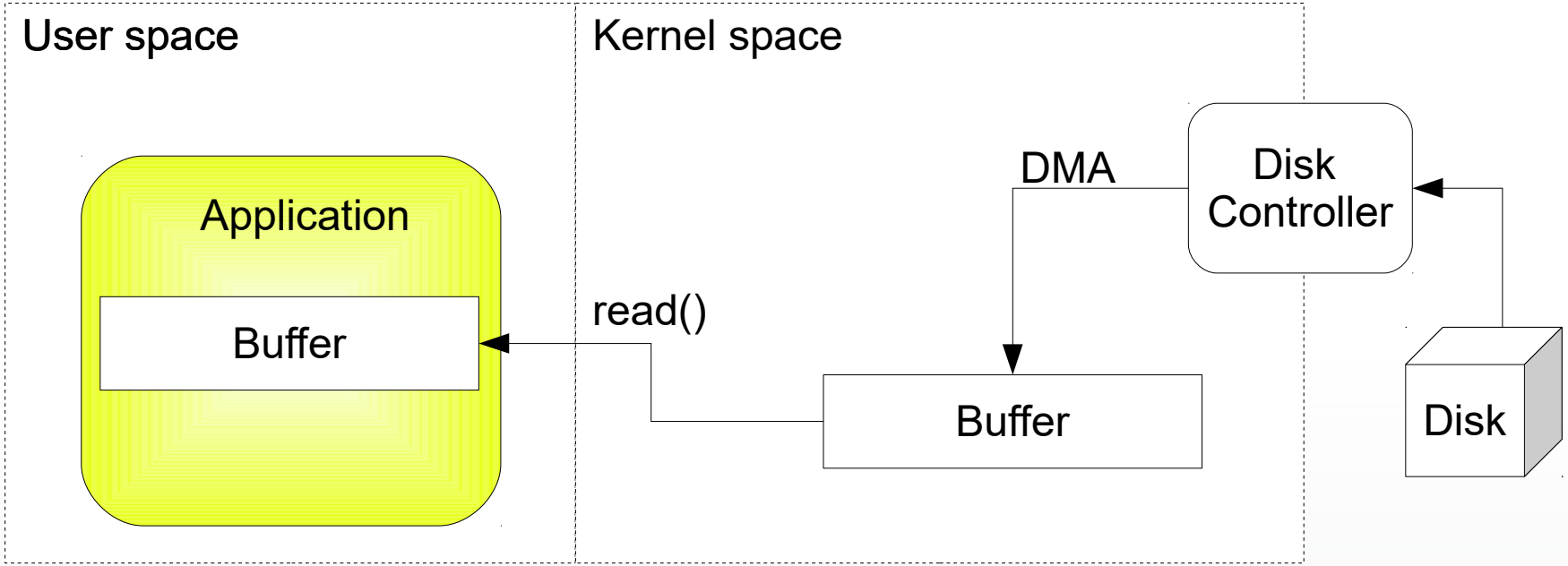
- Le cœur de l'API NIO couvre les domaines suivants
  - Les objets `Channel`
  - Les objets `Buffer`
  - Les objects `Selector`
  - Les expressions régulières
  - La gestion de l'encodage des caractères

NIO : Les Buffers

NIO : Les Buffers

- L'API IO de Java est une API orienté flux (stream), NIO propose une API orienté «bloc» de données
- Un `Buffer` est un espace mémoire de taille fixe constitué de zones mémoires continues (au possible) dans lequel il est possible de lire et d'écrire des données
- Le fonctionnement des `Buffer` est très proche des fonctionnalités natives du système d'exploitation afin de bénéficier de performances accrues

NIO : Les Buffers



NIO : Les Buffers

- Un Buffer s'initialise au travers de méthodes factory proposées par les classes d'implémentation
- La plus courante est la méthode `allocate(int size)`
- **Une fois alloué, la taille d'un `Buffer` ne peut plus être changée**

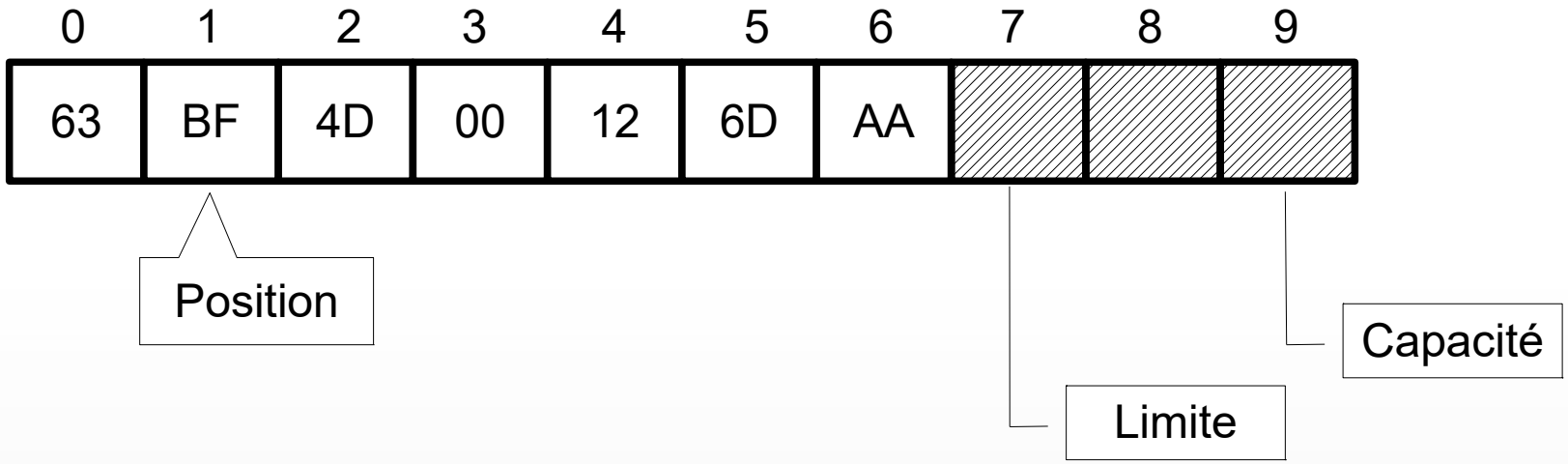
NIO : Les Buffers

- L'API NIO propose plusieurs types de `Buffers`
  - `ByteBuffer`
  - `MappedByteBuffer`
  - `CharBuffer`
  - `DoubleBuffer`
  - `FloatBuffer`
  - `IntBuffer`
  - `LongBuffer`
  - `ShortBuffer`

NIO : Les Buffers

- Tous les types de `Buffer` partagent au minimum les propriétés suivantes :
  - Une capacité : le nombre maximal d'élément qu'ils peuvent contenir
  - Une limite : l'index du premier élément qu'il est impossible de lire ou d'écrire (limite < capacité). Un buffer n'est pas systématiquement plein.
  - Une position : l'index du prochain élément qui sera lu ou écrit

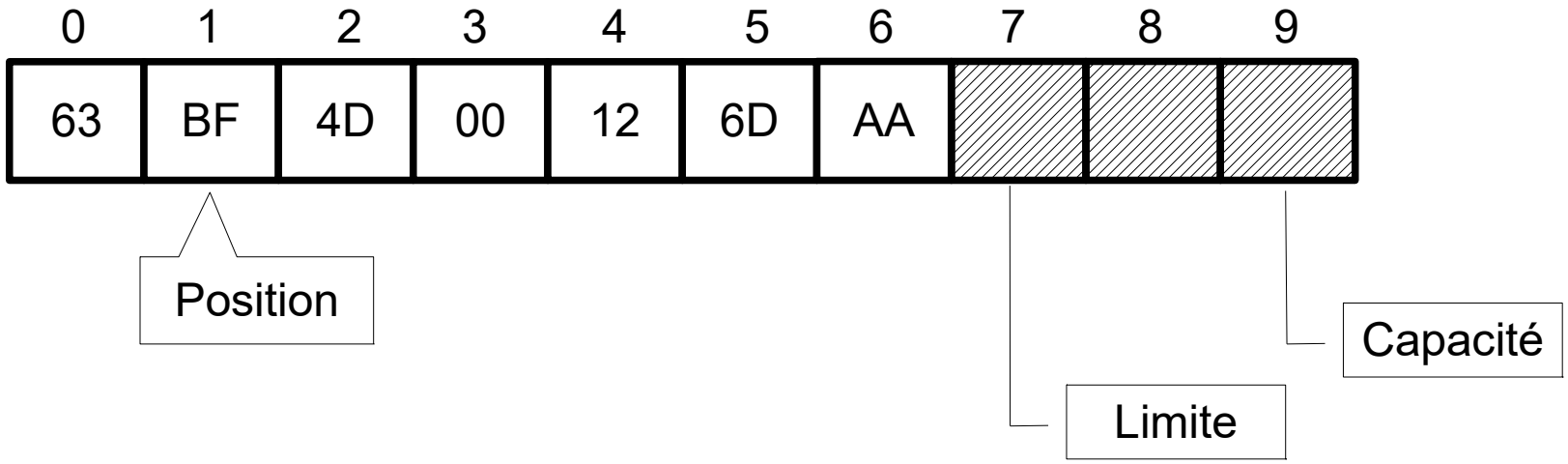
NIO : Les Buffers



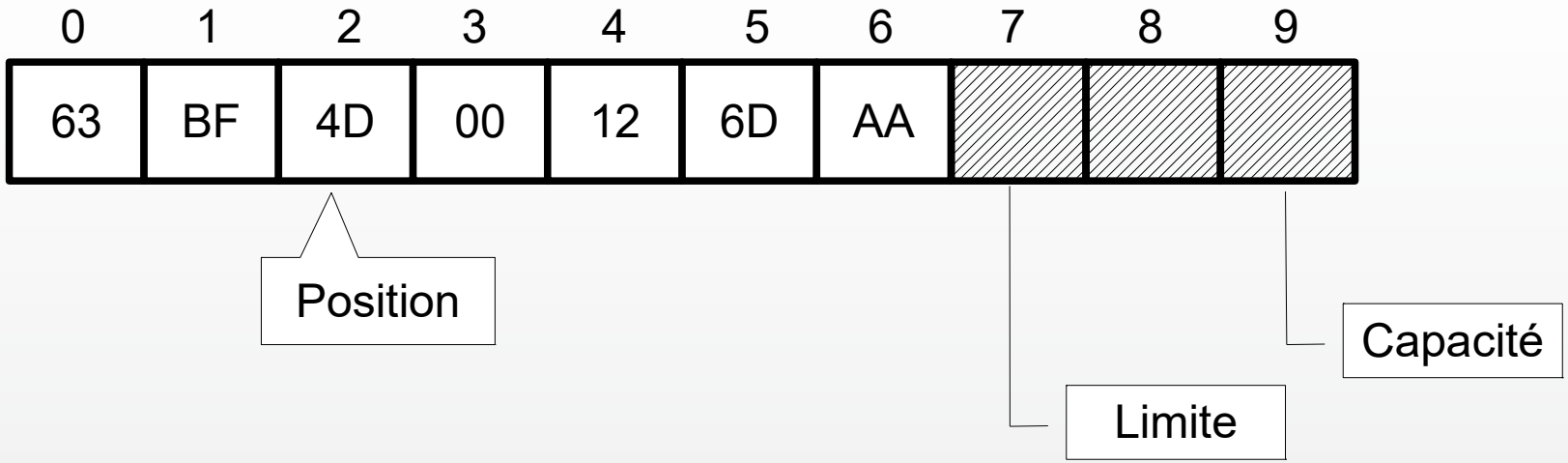
NIO : Les Buffers

- La lecture des données d'un `Buffer` s'effectue au travers des méthodes `get()`
- L'écriture dans un `Buffer` s'effectue au travers des méthodes `put()`
- Toute opération de lecture ou d'écriture d'un `Buffer` provoque la mise à jour de la position courante
- La position indique de ce fait l'index de la valeur qui sera lue lors du prochain appel à la méthode `get()`

NIO : Les Buffers



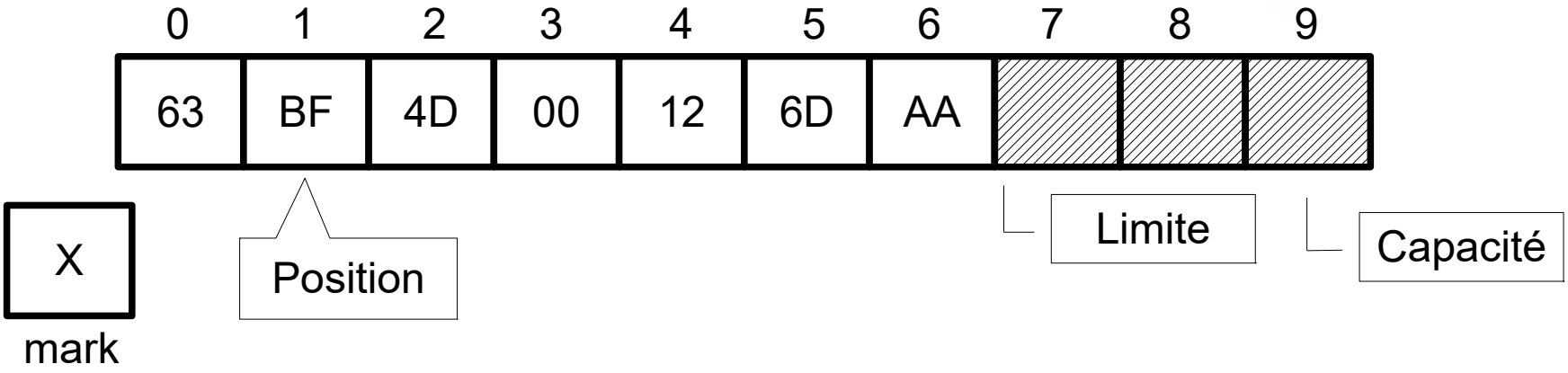

```
myBuffer.get() ;
```



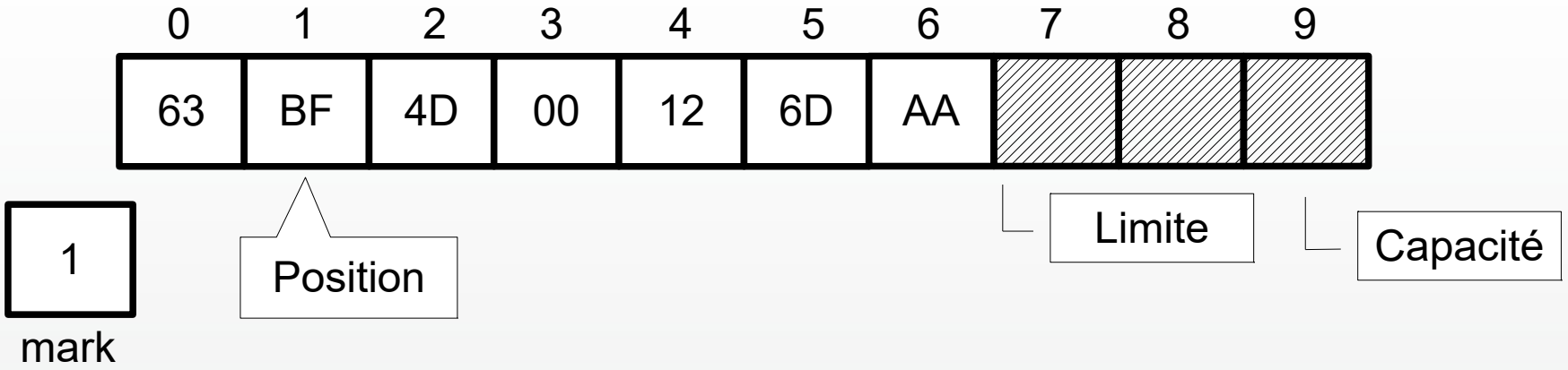
NIO : Les Buffers

- Les objets `Buffer` proposent différentes méthodes facilitant leur utilisation :
  - `mark()` / `reset()` :
    La méthode `mark()` permet de mémoriser l'index courant. Cet index sera restauré lors de l'appel a la méthode `reset()`.

NIO : Les Buffers

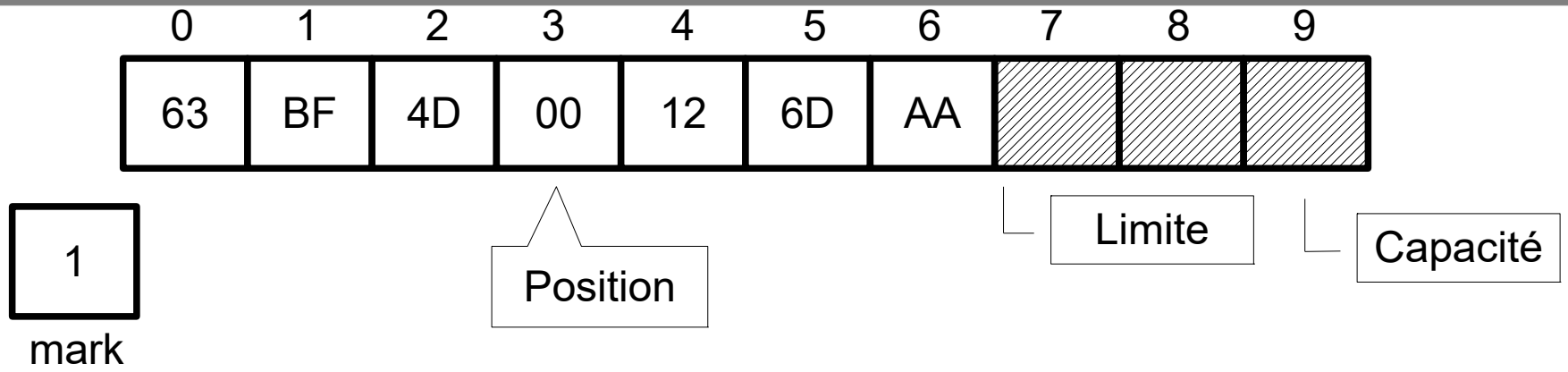



```
MyBuffer.mark() ;
```

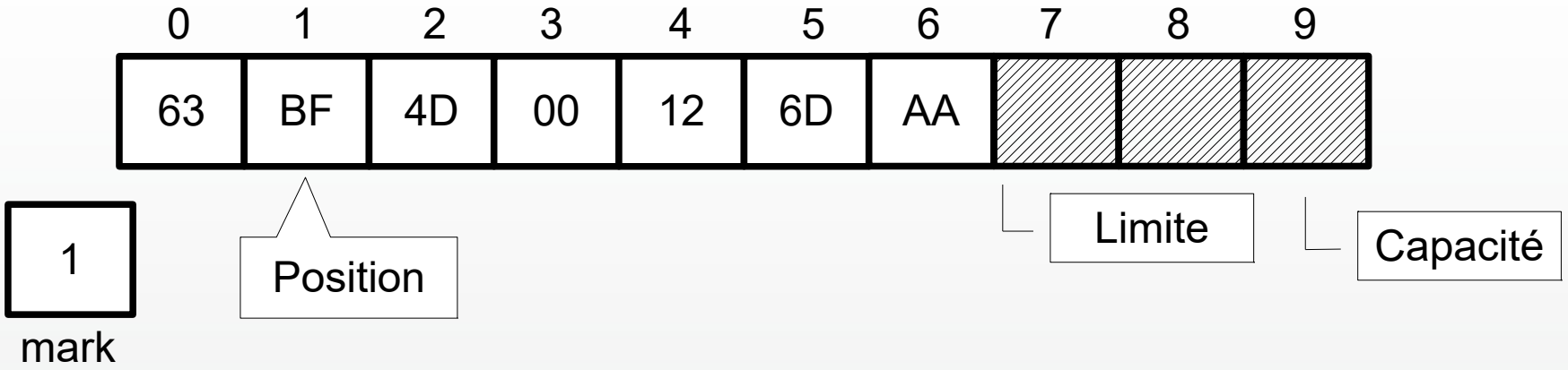


NIO : Les Buffers

```
myBuffer.get() ; ...
```




```
myBuffer.reset() ;
```

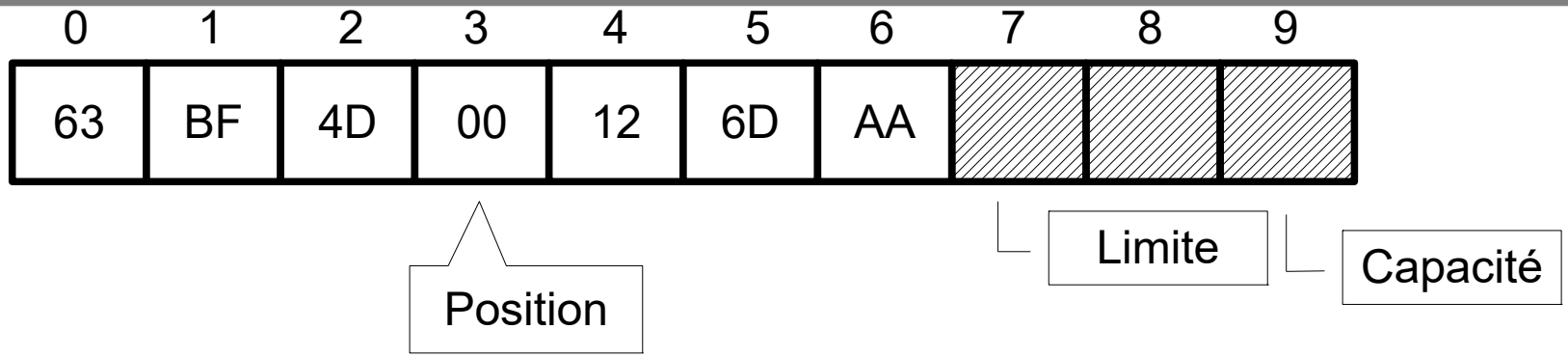


NIO : Les Buffers

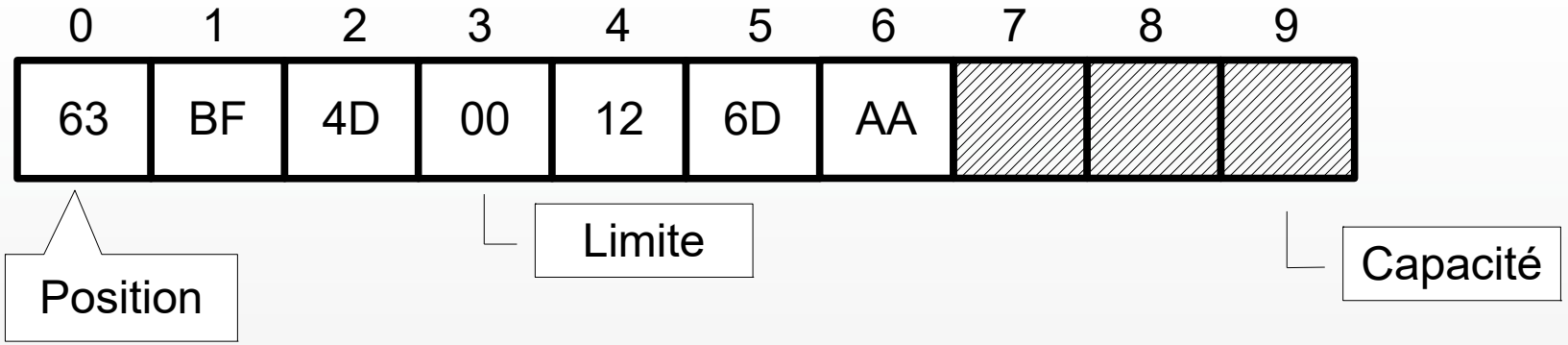
- `flip()` :
  La méthode `flip()` permet de définir la position courante comme limite et de remettre à zéro la position courante.
  Cette méthode est très utilisée après l'écriture d'un `Buffer` lorsque l'on souhaite lire à nouveau son contenu
- `rewind()` :
  Même principe que la méthode `flip()` sauf que la limite n'est pas changée
- `clear()` :
  Définit la limite à la taille `Buffer` et remet la position à zéro

NIO : Les Buffers

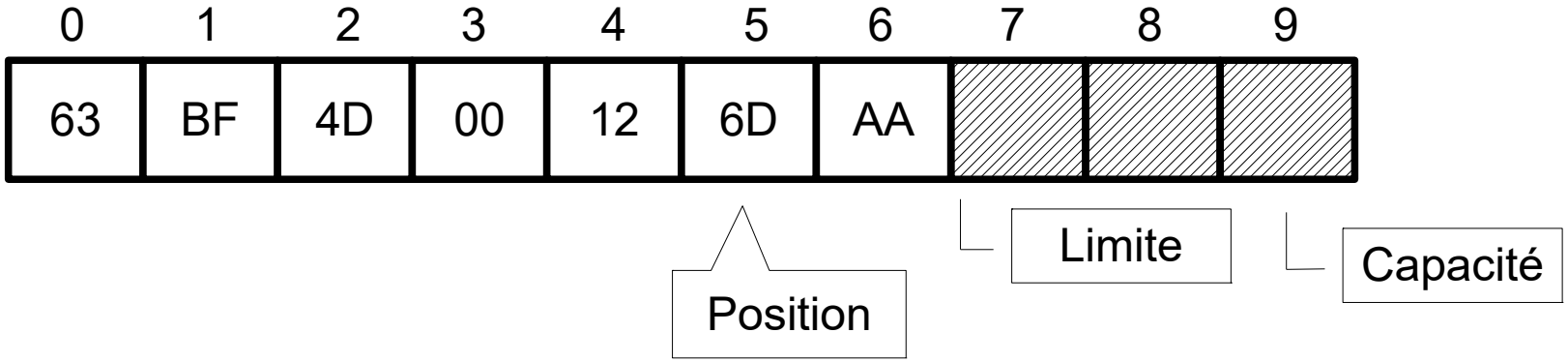
```
myBuffer.put(...) ; ...
```



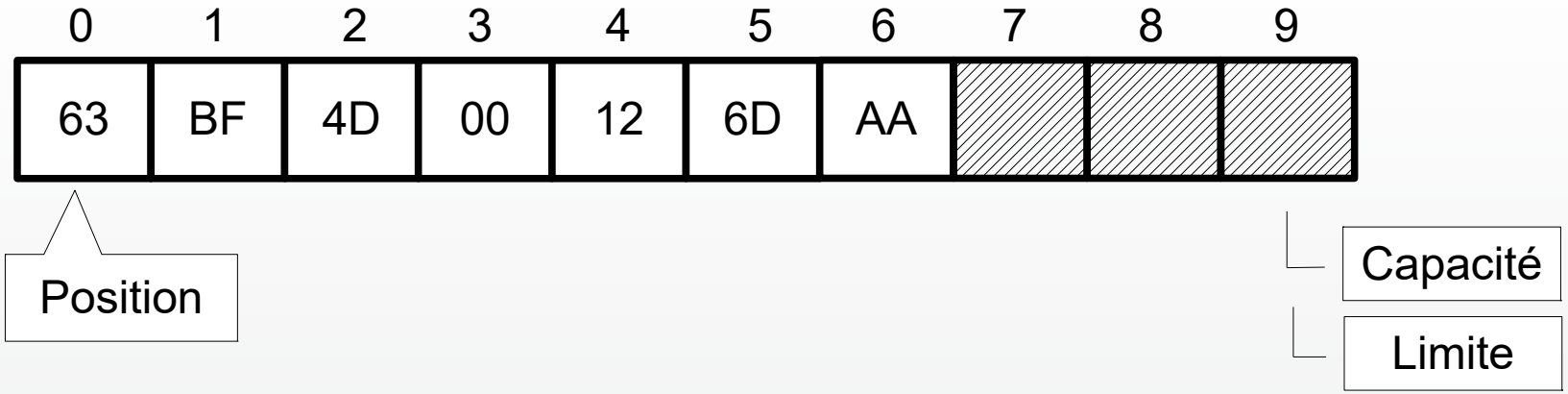

```
myBuffer.flip() ;
```



NIO : Les Buffers




```
myBuffer.reset() ;
```



NIO : Les Buffers

- Méthodes remarquables :
  - `hasRemaining()` / `remaining()` permettent de vérifier s'il reste des éléments (et leur nombre) à lire dans le `Buffer`
  - `compact()` copie les éléments restant dans le `Buffer` en début de `Buffer`
  - `equals()` permet de comparer deux `Buffer`. Deux `Buffer` sont considérés comme égaux s'ils sont de même type et si les éléments restant à lire sont au même nombre et s'ils sont les mêmes

NIO : Les Buffers

- Cas particulier : `ByteBuffer`
  - Cette classe est la plus optimisée car de plus bas niveau : La mémoire n'est qu'une suite d'octets
  - Les `ByteBuffer` peuvent être indirects (créés ou gérés par la JVM) en les allouants avec la méthode classique `allocate()`
  - Ou directs - résidant en dehors de la machine virtuelle, ne sont pas gérés par le garbage collector, au plus proche du système (allocation avec `allocateDirect()`)
    - Dans ce dernier cas ils présentent le moyen le plus efficace de la JVM concernant les entrées sorties

NIO : Les Buffers

- Cette classe offre la possibilité de créer des « vues » de données vers d'autres types : `asCharBuffer()`, `asDoubleBuffer()`...
- Ces méthodes permettent de créer de nouveaux buffers, liés au buffer initial mais dont les attributs `position`, `limit` et `mark` sont indépendantes
- Les buffers sont liés toute modification dans l'un est visible dans l'autre et inversement

NIO : Les Channels

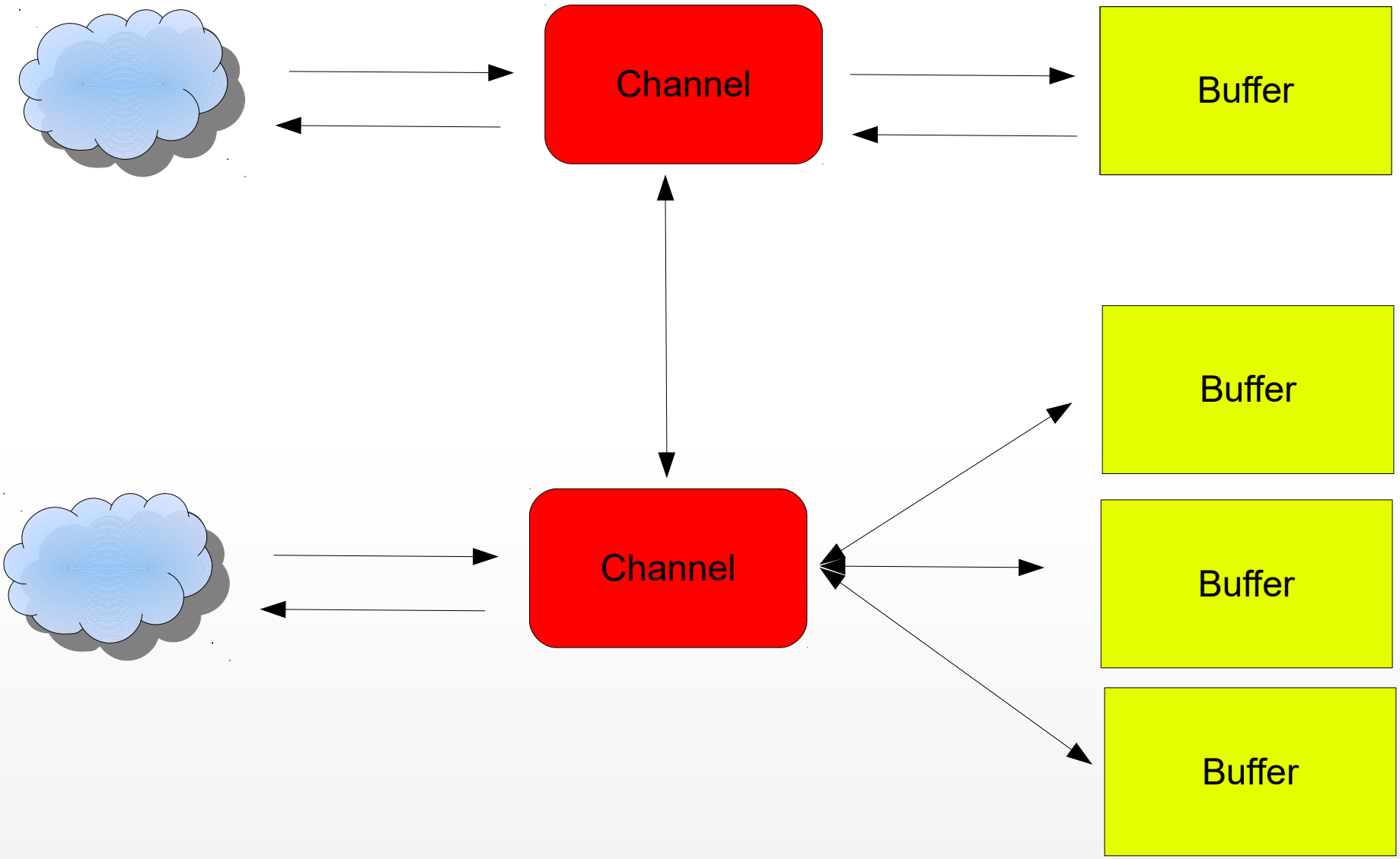
NIO : Les Channels

- Les `Buffers` ne sont qu'une « partie du problème »
- Il est nécessaire de disposer d'objets permettant de faire la transition entre l'entrée/sortie (fichier, socket...) et les `Buffers`
- Ces objets sont appelés `Channel`
- Un `Channel` peut être considéré comme une connexion directe à une entité physique, un fichier, un socket capable de lire et d'écrire des données

NIO : Les Channels

- Les Channel sont conçus pour interagir avec les `Buffer` :
  - Les données lues d'un `Channel` sont transférés dans un `Buffer`
  - Les données d'un `Buffer` peuvent être écrites dans les `Channel`
- Un même objet Channel peut également contrôler (lecture et écriture) plusieurs `Buffer` a la fois

NIO : Les Channels



NIO : Les Channels

- NIO propose plusieurs interfaces dérivées de de l'interface `Channel`
- Tous ces dérivés possèdent des particularités qui leurs sont propres
- Les principales implémentations sont les classes suivantes
  - `FileChannel`
  - `DatagramChannel`
  - `SocketChannel`
  - `ServerSocketChannel`
  - `Pipe`

NIO : Les Channels

- Comment créer des objets `Channel` ?
  Ces classes possèdent tous des constructeurs privés !
- Leur création peut s'effectuer de différentes manières :
  - Au travers de la méthode `getChannel` d'un objet support (exemple `FileOutputStream.getChannel()` renvoie un objet `FileChannel`)

```
FileOutputStream s = new FileOutputStream(path);
FileChannel channel = s.getChannel();
```

NIO : Les Channels

- Au travers des méthodes statiques de la classe utilitaire `java.nio.channels.Channels` qui permet de créer des `Channel` qui englobe des objets de type flux

```
Socket s = new Socket(InetAddress.getLocalHost(),999);
ReadableByteChannel channel = Channels.newChannel(s.getInputStream());
```

- Au travers de méthodes dites « factory » : `SocketChannel.open()`

NIO : Les Channels

Exemple : Lecture d'un fichier

```
RandomAccessFile file = new RandomAccessFile("/Developer/Makefiles/bin/version.pl", "r");
FileChannel channel = file.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(1024);

int read = channel.read(buffer);
while (read != -1) {

  System.out.println("Nb of Bytes Read " + read);
  buffer.flip();

  while(buffer.hasRemaining()){
    System.out.print((char) buffer.get());
  }

  buffer.clear();
  read = channel.read(buffer);
}
file.close();
```

```
Read 1024
#!/usr/bin/perl
#.....
Read 1024
.....
Read 967
.....
```

NIO : Les Channels

- Certains types de `Channel` peuvent être en lecture seule ou en écriture seule
- Certains types de `Channel` peuvent être placés en lecture/ecriture non bloquante qui contrairement à l'API IO ne bloque plus l'exécution d'un programme en attendant les entrées/sorties (cas notamment des `SocketChannel`)
- Dans tous les cas, les `Channel` se basent sur des fonctionnalités natives du système d'exploitation

NIO : Les Channels

- Il est possible d'utiliser certains `Channel` sans passer par un objet `Buffer` intermédiaire grâce aux méthodes `transferFrom()` et `transferTo()`
- Ces méthodes sont extrêmement optimisées et rapides

NIO : Les Channels

- L'API NIO propose des fonctionnalités très intéressantes quant à le lecture et à l'écriture des fichiers au travers de l'objet `FileChannel`
- Un tel objet peut s'obtenir au travers de la méthode `getChannel()` d'un objet de type `RandomAccessFile`

```
RandomAccessFile file = new RandomAccessFile("/Developer/Makefiles/bin/version.pl", "r");
FileChannel channel = file.getChannel();
```

- `RandomAccessFile` a été introduite avec Java 1.4 afin de pouvoir lire un fichier en s'y positionnant (fonctionnement similaire aux `Buffer`)

NIO : Les Channels

- Cet objet permet entre autres de gérer des verrous sur une partie ou la totalité des fichiers ouverts

```
RandomAccessFile file = new RandomAccessFile("/Developer/Makefiles/bin/version.pl", "r");
FileChannel channel = file.getChannel();
FileLock lock  = channel.lock(0, channel.size(), true);

      …

lock.release();
```

- Les verrous peuvent être partagés (shared) et permettre une lecture concurrente mais pas d'écriture, ou exclusifs en gardant tout accès au fichier.

NIO : Les Channels

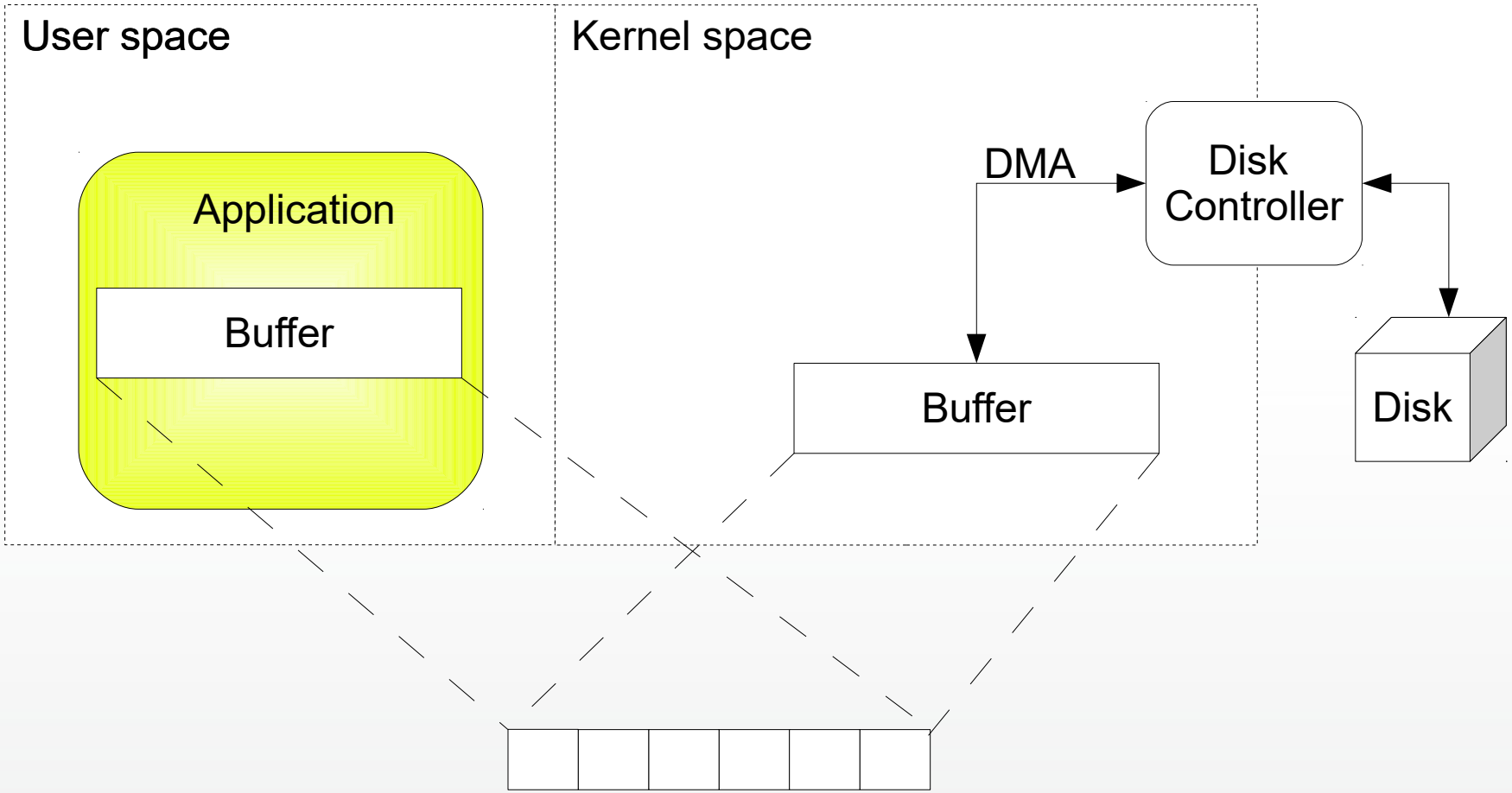
- Il est important de comprendre que les verrous sont gérés par fichier
- Les verrous sont destinés à coordonner l'accès aux ressources par des processus concurrents non pas par les Threads d'une même application

NIO : Les Channels

- Le principal intérêt de la classe `FileChannel` réside dans la méthode `map()`
- Cette méthode permet de créer un objet `Buffer` qui est la représentation directe en mémoire d'une partie d'un fichier : `MappedByteBuffer`
- Toute modification du `Buffer` est directement reproduite sur le disque et inversement
- Les opérations sur un `MappedByteBuffer` sont donc extrêmement rapide car aucune conversion/copie de données n'est nécessaire

NIO : Les Channels

```
RandomAccessFile file = new RandomAccessFile("/Developer/Makefiles/bin/version.pl", "r");
FileChannel channel = file.getChannel();
ByteBuffer b = channel.map(MapMode.READ_WRITE, 0, 1024);
```



NIO : Les Channels

- Remarques :
  - La taille des objets `Buffer` est très importante
  - Une taille non adaptée au contexte peut à l'inverse dégrader les performances
  - De façon globale, l'API NIO permet (en fonction des scénarios) des gains pouvant atteindre 250 % par rapport à l'API IO ...
  - ...mais ceci n'est pas systématique

TP n°6

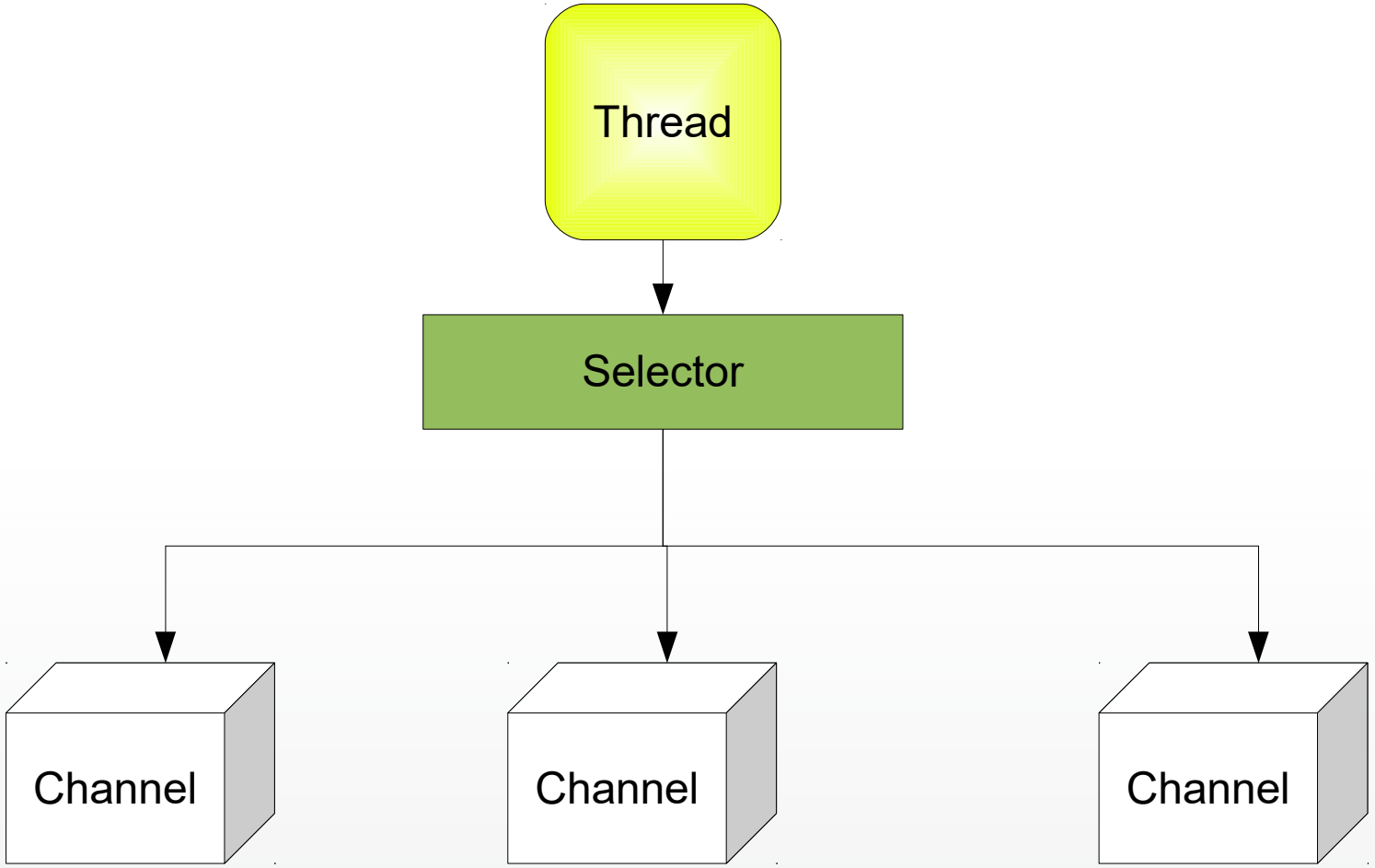
- Voir Sujet

NIO : Les Selectors

NIO : Les Selectors

- NIO introduit le concept d'entrées/sorties non bloquantes : les `Channel` configurés comme étant non bloquants ne suspendent pas l'exécution du programme lors de telles opérations
- Afin d'exploiter ce mécanisme, NIO introduit le concept de `Selector`
- Un `Selector`, est un objet capable de gérer plusieurs `Channel` et de sélectionner automatiquement en fonction de certains paramètres, les `Channel` qui sont prêts à être traités

NIO : Les Selectors



NIO : Les Selectors

- `Selector` est comme un gestionnaire d'évènements : les `Channel` sont liées au `Selector` en fonction de « centre d'intérêts » définissant la notion de « prêt »
- Un Selector se créé via la méthode `Selector.open()`
- Un objet `Channel` peut se « lier » à un objet `Selector` au travers de sa méthode register : `channel.register(Selector s, SelectionKey k)`

NIO : Les Selectors

- Lors de cette opération, l'objet `SelectionKey` permet de définir quelles sont les opérations que l'on souhaite « monitorer »
- Les opération disponibles sont les suivantes :
  - `SelectionKey.OP_ACCEPT`
  - `SelectionKey.OP_CONNECT`
  - `SelectionKey.OP_READ`
  - `SelectionKey.OP_WRITE`

NIO : Les Selectors

- Une fois les objets Channel « enregistrés » auprès du `Selector` il suffit d'utiliser la méthode `select()` du `Selector`
- L'appel est bloquant et suspend le programme jusqu'à ce qu'au minimum un `Channel` réponde aux critères
- L'appel à la méthode `selectedKeys` retourne la liste des `Channel` répondant aux critères d'enregistrement

NIO : Les Selectors

Exemple

```
socketChannel1.configureBlocking(false);
socketChannel2.configureBlocking(false);
socketChannel3.configureBlocking(false);

Selector selector = Selector.open();
socketChannel1.register(selector, SelectionKey.OP_READ);
socketChannel2.register(selector, SelectionKey.OP_WRITE);
socketChannel3.register(selector, SelectionKey.OP_ACCEPT);

while (true) {
     // Blocks until one Channel is meeting required criteria
     selector.select();
     // Test the operation
     for (SelectionKey key : selector.selectedKeys()) {
          if (key.selector().equals(SelectionKey.OP_ACCEPT)) {
               SocketChannel c = (SocketChannel) key.channel();
               // Do something with the Channel...
          }
     }
}
```

NIO : Les Selectors

- La classe `Selector` permet la gestion efficace d'une multitude de `Channel` par un seul `Thread`
- L'intérêt premier des objets `Selector` est de réduire la consommation des ressources liée au multi-threading
- Cependant avec les machines modernes multi-coeurs l'utilisation d'un `Selector` peut s'avérer plus pénalisante que d'utiliser des `Thread`

TP n°7

- Voir Sujet

NIO : Les Charset

NIO : Les Charset

- Un « charset » est la conjonction d'un code représentant un caractère et de sa représentation binaire
- Il existe de nombreux charset chacun représentant des particularités liés aux différentes langues
- Toute JVM supporte au minimum les charset suivants :
  - US-ASCII
  - ISO-8859-1
  - UTF-8
  - UTF-16 / UTF-16BE / UTF-16LE

NIO : Les Charset

- En fonction de la JVM / du système d'exploitation, d'autres charset peuvent être disponibles
- En interne, Java utilise le charset Unicode
- L'objectif d'Unicode est de fédérer tous les charset au sein d'un standard unique qui peut se substituer aux charset « nationaux »
- Chaque système possède un charset par défaut, exploité par les API d'entrée/sortie de la JVM
- En fonction de la source des données, il est souvent nécessaire de les convertir

NIO : Les Charset

- Afin de faciliter le traitement de ces encodages NIO met à disposition la classe `Charset`
- Cet objet permet d'obtenir un convertisseur permettant d'encoder et de décoder un flux de caractères Unicode
- Un `Charset` s'obtient grâce à la méthode statique `Charset.forName(String name)`
- Le nom passé doit être un nom valide reconnu par l'IANA (http://www.iana.org/assignments/character-sets)

NIO : Les Charset

- Les `Charset` peuvent s'utiliser grâce aux méthodes :
  - `encode()` afin de transformer une chaine unicode vers le `Charset` en question
  - `decode()` afin de transformer une source de données vers le `Charset` unicode
- En spécifiant le Charset à utiliser lors des opérations de lecture/écriture

```
InputStreamReader r = new InputStreamReader(System.in, "ISO-8859-1");
```

- Ou la création de chaines de caractères

```
String s = new String(bytes, "ISO-8859-1");
```

NIO : Les Charset

- Lorsque le `Charset` n'est pas précisé, la JVM utilisera toujours le `Charset` par défaut du système d'exploitation pour la conversion de / vers unicode.
- La conversion des flux de caractères s'effectue grâce à des objets `CharsetEncoder` et `CharsetDecoder`, utilisés de façon transparente par la classe `Charset`
- Ces classes permettent un contrôle plus fin de l'encodage, notamment de la substitution en cas de caractères non encodable/décodable

NIO : Les Charset

Exemple

```
Scanner s = new Scanner(new InputStreamReader(new FileInputStream("test.txt"), "UTF-8"));
String text = s.useDelimiter("\\Z").next();
CharsetEncoder encoder = Charset.forName("ISO-8859-1").newEncoder();
CharBuffer b = CharBuffer.wrap(text);
ByteBuffer out = encoder.encode(b);
```

- Toute opération de conversion résultera en la représentation des données encodées sous la forme d'un tableau d'octets ou un buffer spécialisé (`CharBuffer`)

NIO : Les expressions régulières

NIO : Les expressions régulières

- Les expressions régulières encore appelées expressions rationnelles sont des chaines de caractères utilisant une syntaxe définie permettant de décrire un modèle de chaine de caractères :
  - [a-z]*\.[a-z]*@[a-z]*\.(com|fr)
- Elles sont indispensables pour la recherche et l'extraction de données textuelles
- Elles constituent le moyen le plus rapide de rechercher des informations dans une grande masse de données textuelles

NIO : Les expressions régulières

- Exemple :
  - *[a-z]*[1-9]{3}*
    - *Chaine ne contenant que des caractères mais finissant par exactement trois chiffres :*
      - *Abcde123*
      - *A587*
    - [a-z] : n'importe quel caractère entre « a » et « z »
    - * : zero ou plusieurs fois
    - [1-9] : n'importe quel chiffre entre 1 et 9
    - {3} : exactement trois fois
  - ([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})

NIO : Les expressions régulières

- La classe `String` contient un support partiel des expressions régulières au travers des méthodes
  - `s.matches(String regex)` :
    Retourne `true` si toute la chaine correspond à l'expression régulière
  - `s.split(String regex)` :
    Convertit la chaine en un tableau en se basant sur l'expression régulière (non incluse dans le résultat final)
  - `s.replace(String regex, String s)` :
    Effectue des remplacements dans la chaine en se basant sur l'expression régulière

NIO : Les expressions régulières

- Les méthodes de la classe String ne sont pas optimisées
- La classe `java.util.regex.Pattern` constitue le cœur des expressions régulières en Java.
- La JavaDoc de cette classe résume l'ensemble de la syntaxe supportée :
  - http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

NIO : Les expressions régulières

- Cette classe s'initialise au travers de la méthode statique
  `Pattern.compile(String regexp)`
- La méthode
  `Pattern.matches(String regexp, String input)` permet de vérifier si une chaine correspond à l'expression régulière passée en paramètres

NIO : Les Channels

Exemple : Adresse Email

```
System.out.println(Pattern.matches("[a-z]*\\.[a-z]*@[a-z]*\\.(com|fr)", "a.b@c.com"));
System.out.println(Pattern.matches("[a-z]*\\.[a-z]*@[a-z]*\\.(com|fr)", "a.b@c.fr"));
System.out.println(Pattern.matches("[a-z]*\\.[a-z]*@[a-z]*\\.(com|fr)", "not working"));
System.out.println(Pattern.matches("[a-z]*\\.[a-z]*@[a-z]*\\.(com|fr)", "a.b@c.de"));
```

```
True
True
False
False
```

NIO : Les expressions régulières

- Afin de bénéficier de plus de contrôle sur les résultats de recherche, la classe `Pattern` permet d'obtenir un objet de type `Matcher` au travers de la méthode `Pattern.matcher(Sting input)`
- La classe `Matcher` permet d'effectuer des recherches incrémentales (`find()`) ou de vérifier la cohérence de l'expression régulière par rapport à l'intégralité de la chaine de caractères passée en paramètres (`matches()`)

NIO : Les expressions régulières

- Les expressions régulières permettent également l'extraction de données au travers de ce qui est communément appelé les « capturing groups »
- Un capturing group est une partie d'une expression régulières délimitée par des parenthèses que l'on souhaite pouvoir extraire pour un usage ultérieur

NIO : Les expressions régulières

- ([a-z]*\.[a-z]*)@[a-z]*\.(com|fr)
  - Cette expression régulière comprend deux groupes de capture
  - Lorsque cette expression est appliquée à l'adresse : nom.prenom@entreprise.fr
    - Le premier groupe contiendra : « nom.prenom »
    - Le second groupe : « fr »

NIO : Les expressions régulières

Exemple : Adresse Email

```
Pattern p = Pattern.compile("([a-z]*\\.[a-z]*)@[a-z]*\\.(com|fr)");
Matcher m = p.matcher("nom.prenom@domaine.fr");

if(m.matches()){
     System.out.println("Nom/prenom : "+m.group(1));
     System.out.println("Extension : " +m.group(2));
}

p = Pattern.compile("([1-9])");
m = p.matcher("1 little bird, 2 little birds, 3 little birds");
while(m.find()){
     System.out.print(m.group(1)+"\n");
}
```

```
Nom/prenom : nom.prenom
Extension : fr
1
2
3
```

NIO : Les expressions régulières

Exemple : Pages web

```
URL url = new URL("http://www.lemonde.fr");
HttpURLConnection cx = (HttpURLConnection) url.openConnection();
InputStream in = cx.getInputStream();
StringBuffer buff = new StringBuffer();
int i;
while((i = in.read())!=-1){
     buff.append((char)i);
}

Pattern p = Pattern.compile("<a class=\"titre\".*?><span.*?>(.*?)<\\/span>(.*?)<\\/a>",
Pattern.DOTALL|Pattern.MULTILINE);

Matcher m = p.matcher(buff);
while(m.find()){
     System.out.println(m.group(1)+"-"+m.group(2));
}
```

NIO : Les expressions régulières

- Quelques ressources
  - La javadoc de la classe Pattern détaille avec précision la syntaxe des expressions régulières Java
  - http://www.rubular.org - un excellent débugger d'expressions régulières en ligne et gratuit
- Quelques remarques
  - *Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.*
  - Attention au HTML

TP n°8

- Voir Sujet