

Systèmes Distribués

Fabrice Huet
fhuet@sophia.inria.fr

Introduction

- ▶ Compléments de programmation Java
- ▶ Étude de 3 mécanismes nécessaires pour les systèmes distribués
 - ▶ Flots de données
 - ▶ Threads et synchronisation
 - ▶ New I/O
- ▶ Très inspiré du cours de Richard Grin (Master 1 Informatique)

1- Java io

Flots

- ▶ Représentent un canal de communication
- ▶ Les données peuvent y être lues ou écrites séquentiellement
 - ▶ Type FIFO
 - ▶ Pas d'accès aléatoire comme dans un tableau
- ▶ Pas de notion de limites de données
 - ▶ Des données écrites en 2 fois peuvent être lues en 1 fois
- ▶ Package java.io.
- ▶ 2 types de flots
 - ▶ Manipulation d'octets
 - ▶ Manipulation de caractères
- ▶ Tous les flots lèvent une IOException

InputStream	Lecture de flots d'octets
OutputStream	Écriture de flots d'octets
Reader	Lecture de flots de caractères
Writer	Écriture de flots de caractères
File	Fichiers et répertoires
StreamTokenizer	Analyse lexicale

Types de flots, Types de classes

- ▶ Flots d'octets
 - ▶ Lecture/Écriture d'octets
- ▶ Flots de caractères
 - ▶ Manipule des caractères Unicode
 - ▶ Codés en Java sur 2 octets
- ▶ 2 Types de classes
 - ▶ Classes de base liés à une source ou une destination (Ex: FileReader)
 - ▶ Classes de décoration (Ex: BufferedReader)

Sources et destinations concrètes

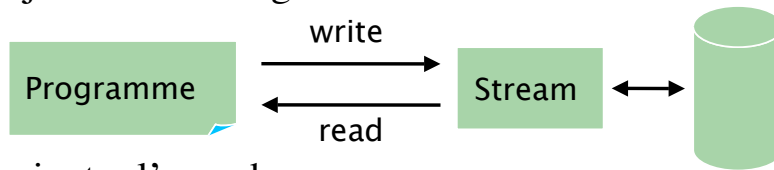
- ▶ Fichiers
 - ▶ File{In|Out}putStream
 - ▶ File{Reader|Writer}
- ▶ Tableaux
 - ▶ ByteArray{In|Out}putStream
 - ▶ CharArray{Reader|Writer}
- ▶ Chaînes de caractères
 - ▶ String{Reader|Writer}

Décoration

- ▶ De base, un flot ne sait que lire ou écrire
 - ▶ Supporté par la plupart des périphériques
 - ▶ Méthodes read et write
- ▶ Mais très vite limitatif
- ▶ Pleins de fonctionnalités possibles
 - ▶ Mise en mémoire tampon (Bufferisation)
 - ▶ Encodage/Décodage des données
 - ▶ Compression/Décompression
- ▶ Idéalement, indépendants de la lecture ou de l'écriture de base
- ▶ Comment ajouter des fonctionnalités sans toucher aux classes de base: la décoration

Décoration

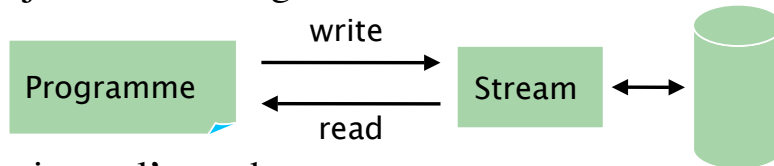
- ▶ Exemple: ajouter l'encodage de données



- ▶ Comment ajouter l'encodage
 - ▶ 1ère solution: modification de la classe Stream
 - ▶ Mais classes multiples (stream normal, stream avec encodage...)
 - ▶ Nécessite de faire le même travail pour tous les périphériques

Décoration

- ▶ Exemple: ajouter l'encodage de données



- ▶ Comment ajouter l'encodage
 - ▶ 1ère solution: modification de la classe Stream
 - ▶ Mais classes multiples (stream normal, stream avec encodage...)
 - ▶ Nécessite de faire le même travail pour tous les périphériques
 - ▶ 2ème solution: on décore la classe Stream, on lui ajoute une fonctionnalité (par héritage ou wrapper)



Décorateurs

- ▶ Bufferisation des I/Os
 - ▶ Buffered{In|Out}putStream
 - ▶ Buffered{Reader|Writer}
- ▶ Lecture/écriture de types primitifs indépendamment de la plateforme
 - ▶ Data{In|Out}putStream
- ▶ Compteur de lignes
 - ▶ LineNumberReader
- ▶ Écriture de données sous forme de chaînes de caractères
 - ▶ PrintStream
 - ▶ PrintWriter
- ▶ Remettre un caractère dans un flot
 - ▶ PushbackInputStream
 - ▶ PushbackReader

1.1- Lecture de flots d'octets

Classe InputStream

- ▶ Classe abstraite
- ▶ Superclasse de toutes les classes permettant la lecture de flots d'octets
- ▶ Toutes les classes manipulant un flot d'octets seront vues comme un InputStream
 - ▶ Équivalent à une interface
- ▶ Possède un constructeur sans paramètre

Méthodes publiques

- ▶ `int read()`
 - ▶ Retourne le prochain octet dans le flot de données
 - ▶ Valeur de retour entre 0 et 255
 - ▶ Retourne -1 si fin de flot atteinte
 - ▶ Bloquante si aucun octet à lire
 - ▶ Abstraite

Méthodes publiques

- ▶ `int read(byte[] b)`
 - ▶ Essaie de lire assez d'octets pour remplir le tableau `b`
 - ▶ Retourne le nombre d'octets effectivement lus ou `-1`
 - ▶ Implémentée en utilisant la méthode `read()`
- ▶ Attention
 - ▶ Le tableau peut n'être rempli que partiellement
 - ▶ Retourne le nombre lu, pas la valeur!

Méthodes publiques

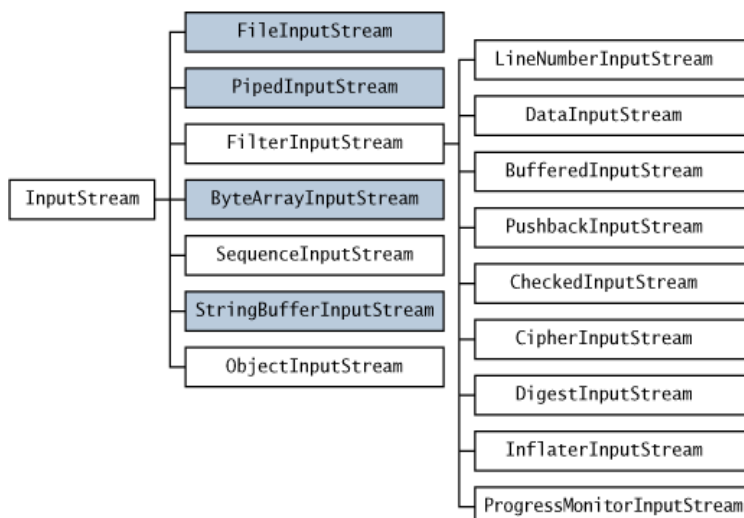
- ▶ `int read(byte[] b, int off, int len)`
 - ▶ Lis `len` octets et les place dans le tableau à partir de l'indice `off`.
 - ▶ Retourne le nombre d'octets effectivement lus ou `-1`
- ▶ `long skip(long n)`
 - ▶ Saute `n` octets dans le flot
 - ▶ Retourne le nombre d'octets sautés
- ▶ `int available()`
 - ▶ Renvoie le nombre d'octets pouvant être lus

Méthodes publiques

- ▶ boolean markSupported()
 - ▶ Test si le flot supporte la notion de marque et de retour en arrière
- ▶ void mark(int readlimit)
 - ▶ Marque la position courante
 - ▶ Readlimit : nombre d'octets lus avant oubli de la marque
- ▶ void reset()
 - ▶ Positionne le flot à la dernière marque

Sous classes de InputStream

- ▶ En gris sont indiquées des classes associées à des sources et destinations concrètes



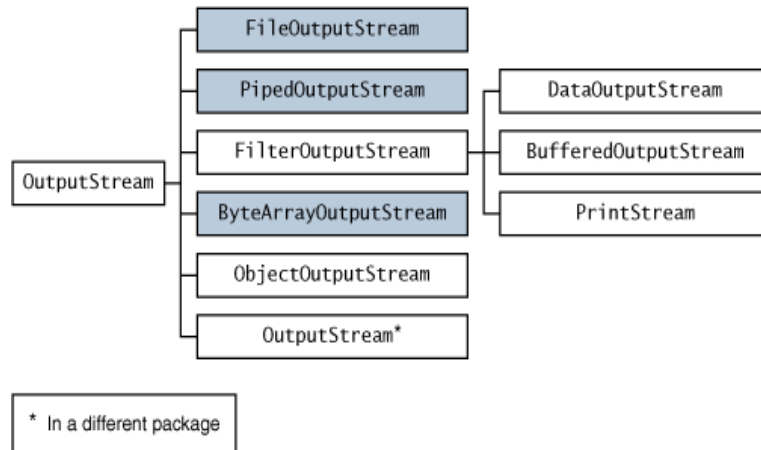
1.2- Écriture de flots d'octets

Classe OutputStream

- ▶ Classe abstraite
- ▶ Méthodes
 - ▶ `int write(int c)`
 - ▶ `int write(byte cbuf[])`
 - ▶ `int write(byte cbuf[], int offset, int length)`
- ▶ Dans le cas de `write(int c)` seuls les 8 bits de poids faible sont écrits

Sous classes de OutputStream

- ▶ En gris sont indiquées des classes associées à des sources et destinations concrètes



1.3- Décorateurs et Filtres

Principe

- ▶ Un objet décorateur ajoute une fonctionnalité à un objet décoré
- ▶ Le constructeur du décorateur reçoit l'objet qu'il décore
- ▶ Quand une méthode est appelée sur le décorateur
 - ▶ Il effectue un traitement
 - ▶ Utilise, si nécessaire, les méthodes de l'objet décoré
 - ▶ Retour un éventuel résultat

Principe

- ▶ Pour fonctionner, le décorateur doit être compatible avec le décoré
 - ▶ Utilisation d'interfaces ou héritage
- ▶ Principe récursif
 - ▶ Un décorateur peut être décoré
- ▶ En Java, les décorateurs sont sous classes d'InputStream et d'OutputStream

Exemple

- ▶ Décorer pour lire des flux depuis une Socket
- ▶ Obtenir le flot d'entrée de la socket
 - ▶ `maSocket.getInputStream()`
- ▶ Le décorer pour bufferiser les lectures
 - ▶ `BufferedInputStream bis = new BufferedInputStream(maSocket.getInputStream())`
- ▶ Ensuite, on peut lire des flots d'octets depuis bis
- ▶ Même fonctionnement pour l'écriture
 - ▶ `BufferedOutputStream bos = new BufferedOutputStream(maSocket.getOutputStream());`

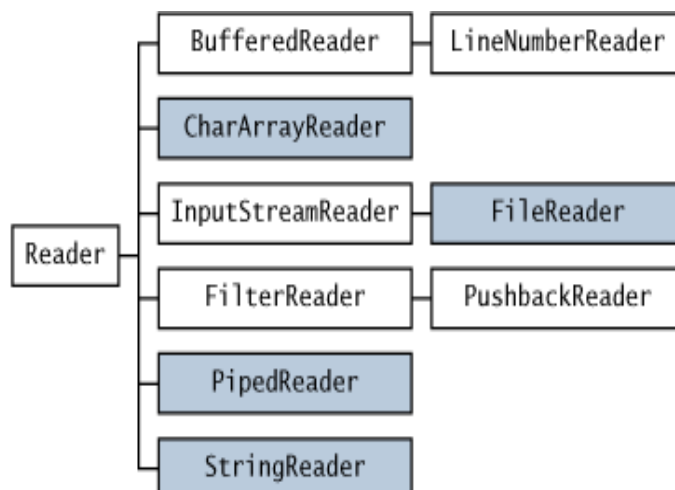
1.4- Lecture de flots de caractères

Classe Reader

- ▶ `int read()`
- ▶ `int read(char[] b)`
- ▶ `int read(char[] b, int off, int len)`
- ▶ `long skip(long n)`
- ▶ `boolean ready()`
- ▶ `abstract void close()`
- ▶ `void mark()`
- ▶ `void reset()`
- ▶ `boolean markSupported()`

Sous classes de Reader

- ▶ Pour lire des lignes de texte, on utilise `BufferedReader` et la méthode `readLine()`



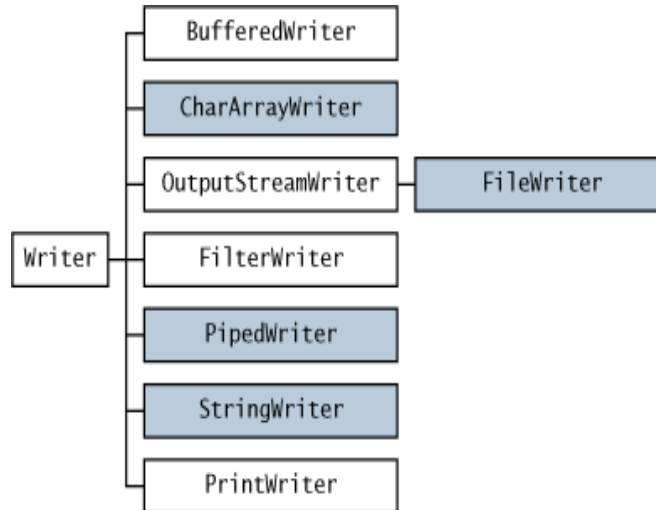
1.5- Écriture de flots de caractères

Classe Writer

- ▶ `int write(int c)`
- ▶ `void write(char[] b)`
- ▶ `void write(char[] b, int off, int len)`
- ▶ `void write(String s)`
- ▶ `void write(String s, int off, int len)`
- ▶ `abstract void flush()`
- ▶ `abstract void close()`

Sous classes de Writer

- ▶ Pour écrire des lignes de texte, on utilise `PrintWriter` et la méthode `println()`



Exemple

- ▶ Lecture de flux depuis une socket en mode caractère
 - ▶ `in = new BufferedReader(new InputStreamReader(maSocket.getInputStream()))`
;
- ▶ Même fonctionnement pour l'écriture
 - ▶ `out = new PrintWriter(maSocket.getOutputStream(), true);`
- ▶ La transformation entre octets et caractères se fait avec les classes `InputStreamReader` et `PrintWriter`

2- Threads

Introduction

- ▶ Un programme est dit multitâches (multithread) lorsque plusieurs parties de son code s'exécutent en même temps
- ▶ Tous les OS modernes sont multitâches et permettent l'exécution de programmes multitâches
- ▶ Exécution parallèle
 - ▶ Simulée en mono-processeur
 - ▶ Réelle en multi-processeur
- ▶ Un système est dit préemptif si un programme peut être interrompue pour laisser une autre s'exécuter
 - ▶ Sinon, c'est un programme d'annoncer qu'il n'a plus besoin de s'exécuter

Thread

- ▶ Le parallélisme est construit sur la notion de threads (processus léger)
- ▶ Un processus représente un programme en cours d'exécution
 - ▶ Plusieurs processus existent dans le système
 - ▶ Chacun a son espace d'adressage privé
- ▶ Un processus contient une ou plusieurs Threads
 - ▶ Même espace d'adressage, donc même variables
- ▶ Pourquoi les threads?
 - ▶ Le changement de contexte (enlever un processus du processeur pour en exécuter un autre) est très coûteux
 - ▶ Pas de changement de contexte pour les threads
- ▶ Ça sert?
 - ▶ Énormément dans les interfaces graphiques

Threads en Java

- ▶ Un programme Java standard possède plusieurs threads
 - ▶ Le code utilisateur est souvent exécuté par plusieurs threads à son insu
- ▶ Java fournit une API standard pour manipuler les threads
 - ▶ Au final, ce sont (peut-être) des threads de l'OS qui sont manipulées
- ▶ A tout thread est associé
 - ▶ Un objet qui représente le code exécuté par ce thread
 - ▶ Un objet permettant de manipuler/contrôler ce thread depuis l'application (classe Thread)
 - ▶ Parfois une seule classe contient le code et agit comme un contrôleur...
- ▶ Un programme Java (la JVM) ne s'arrête pas tant qu'il existe au moins un thread en activité

Runnable et Thread

- ▶ Le code exécuté par un thread se trouve dans la méthode run() de l'interface Runnable
- ▶ Toute classe implémentant cette interface peut être exécutée par un thread
- ▶ Pour l'exécuter, il faut
 - ▶ Relier une instance à un contrôleur de thread (classe Thread)
 - ▶ Appeler la méthode start() du contrôleur
- ▶ Un Thread peut s'endormir pour une certaine durée avec la méthode Thread.sleep()

Exemple

```
public class Test implements Runnable {
    public void run() {
        System.out.println("Running!« );
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        new Thread(t1).start();
    }
}
```

Suspendre et Continuer

- ▶ suspend : interrompt un thread
- ▶ resume : reprends l'exécution
- ▶ stop : arrêt
- ▶ En java, les méthode suspend(), resume() et stop() sont deprecated
 - ▶ Considérées comme non sûres
 - ▶ On peut implémenter le même comportement sans ces méthodes
- ▶ Suivant les versions de Java, ces méthodes ne font rien
 - ▶ Mais aucune levée d'exception si utilisées

Problème de synchronisation

- ▶ Plusieurs threads partagent les même données
 - ▶ Attributs d'un objet
 - ▶ Variables statiques
- ▶ Les variables locales aux méthodes ne sont pas partagées
- ▶ Donc un thread peut modifier une donnée qu'un autre thread utilise

Problème de synchronisation

- ▶ Exemple: Soit le code suivant exécuté par 2 threads, quels sont les résultats possibles

```
X=2;  
tmp=x;  
tmp++;  
X=tmp  
return x;
```

Il est nécessaire d'avoir un mécanisme permettant de contrôler l'exécution d'un code par plusieurs threads
Une partie de code qui doit n'être exécutée par un unique thread est une section critique

Synchronisation

- ▶ La synchronisation en Java se fait avec des moniteurs
 - ▶ Un moniteur est un objet qui protège l'accès à certaines méthodes
 - ▶ On ne peut exécuter qu'une seule méthode d'un moniteur à la fois
- ▶ Chaque objet possède un moniteur qui ne peut être entré que par un seul thread
- ▶ Une classe possède aussi un moniteur
- ▶ On ne manipule pas directement ce moniteur
 - ▶ La synchronisation se fait avec le mot clé synchronized

Synchronisation

- ▶ L'unité de synchronisation est indiquée par le mot `synchronized`
 - ▶ Méthode
 - ▶ Suite d'instructions

```
public synchronized void test() {  
    .....  
}
```

```
Object o = ...  
  
public void test() {  
    synchronized(o) {  
        ...  
    }  
}
```

Un thread prend le moniteur en entrant dans une méthode `synchronized` et le libère automatiquement en sortie
Moniteur de l'objet courant si méthode
Moniteur de l'objet cible si bloc

Synchronisation

- ▶ Un seul thread peut exécuter du code synchronisé
 - ▶ Mais possible pour les autres méthodes
- ▶ Si un autre thread veut exécuter ce code, il est mis en attente
- ▶ Plusieurs threads peuvent attendre
- ▶ Quand le thread initial a fini, le système réveille ceux en attente
 - ▶ Tout le monde est réveillé
 - ▶ On ne sait pas qui aura le moniteur (non déterminisme apparent)
 - ▶ Ceux qui n'auront pas le moniteur seront mis en attente (risque de livelock)

Collaboration de threads

- ▶ Dans un programme multitâches, il arrive que
 - ▶ Un thread t1 ne puisse pas continuer son exécution tant qu'une condition n'est pas remplie
 - ▶ La condition dépende d'un thread t2
- ▶ Exemple: un thread qui fait afficher à l'écran une variable quand sa valeur a changée
- ▶ Solution naïve
 - ▶ t1 vérifie périodiquement si la condition est remplie
 - ▶ Coûteux
- ▶ Solution optimale
 - ▶ t1 « dort » et t2 le réveille quand il a changé la condition

wait/notify

- ▶ Permet à des threads de se synchroniser suivant une condition
- ▶ Utilisation
 - ▶ t1 exécute une section critique, mais ne peut la continuer sans qu'une condition ne soit remplie. Il appelle `wait()`
 - ▶ t2 exécute une section critique et modifie une condition. Il avertit un thread en sommeil avec `notify()`, ce qui provoque son réveil.
 - ▶ t1 ne pourra s'exécuter que lorsque t2 aura rendu le moniteur (sortie de la section critique par exemple)

wait

- ▶ `public final void wait() throws InterruptedException`
- ▶ Nécessite que le thread appelant possède le moniteur de cet objet
- ▶ S'appelle depuis une méthode synchronized (`wait()`) ou dans un bloc synchronized sur un objet `o` (`o.wait()`)
 - ▶ Bloque le thread jusqu'à ce qu'un autre thread appel notify

wait

- ▶ Mais si un thread fait un `wait` dans une section critique
 - ▶ Il possède le moniteur
 - ▶ Donc aucun autre thread ne peut théoriquement s'exécuter
- ▶ Fait un `wait` rend libère temporairement le moniteur
 - ▶ Le thread est mis immédiatement à "dormir"
- ▶ À son réveil, il récupère automatiquement le moniteur et reprend son exécution après le `wait`

notify/notifyAll

- ▶ `public final void notify()`
- ▶ `public final void notifyAll()`
- ▶ Nécessite que le thread appelant possède le moniteur de cet objet
- ▶ Notify réveille un seul des threads en attente
 - ▶ Lequel? Dépend de l'implémentation de la JVM (i.e non déterministe)
- ▶ NotifyAll réveille tous les threads
 - ▶ Mais un seul aura le moniteur pour s'exécuter
 - ▶ Les autres devront attendre leur tour (mais pas besoin d'autre notify)
 - ▶ Lequel? Dépend de l'implémentation
- ▶ Attention: un notify est perdu si aucun thread n'est en attente

Synchronisation sur condition

- ▶ Un thread veut attendre qu'une condition soit true
- ▶ Cette condition est modifiée par un autre thread
- ▶ Thread attendant
 - ▶ `if (!condition) wait();`
- ▶ Thread changeant la condition
 - ▶ `notify();`
- ▶ A priori bonne solution... ou pas
- ▶ Quel problème peut arriver?

Synchronisation sur condition

- ▶ Si on quitte le wait cela signifie qu'un autre thread a appelé notify sur le moniteur
- ▶ Mais un autre thread a pu se réveiller, s'exécuter, modifier la condition et sortir de la section critique, avant notre exécution
- ▶ Il est aussi possible que le notify concernait une autre condition
- ▶ Bonne solution
- ▶ `while(!condition) object.wait();`

Exemple

```
public class ObjetSynchronized {
    boolean condition = false;

    public synchronized void changeCondition() {
        condition = true;
        notify();
    }

    public synchronized void waitCondition() {
        while (!condition) {
            System.out.println("Je vais dodo");
            wait();
        }
        System.out.println("Fini");
    }
}
```

Exemple

```
public class Test implements Runnable {
    ObjectSynchronized o;
    public Test(ObjectSynchronized o) {
        this.o = o;
    }
    public void run() {
        o.waitCondition();
    }
    public static void main(String[] args) {
        ObjectSynchronized o = new ObjectSynchronized();
        Test t1 = new Test(o);
        new Thread(t1).start();
        Thread.sleep(5000);
        o.changeCondition();
    }
}
```

Retour sur suspend/resume/stop

- ▶ Que se passe-t-il si un thread qui a un moniteur appelle suspend?
 - ▶ Si il le garde, personne ne peut accéder à l'objet
 - ▶ Si il le rend, à son réveil rien ne garantie qu'il l'ait à nouveau, contrairement à un wait
- ▶ Solution Java
 - ▶ On fait pas
- ▶ Solution C#
 - ▶ On peut faire, mais attention...

Safe/Unsafe

- ▶ Du code qui peut s'exécuter de manière sûre en multithread est dit Thread Safe
- ▶ Il est vital d'indiquer dans la documentation du code que l'on écrit si il est safe ou unsafe
- ▶ Exemple: la documentation Java
 - ▶ « Note that this implementation is not synchronized. If multiple threads access a set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. »

3- Java nio

Introduction

- ▶ I/O classiques en Java, métaphore des flots
 - ▶ Les octets sont lus 1 par 1
 - ▶ Des mécanismes plus complexes sont obtenus par décoration
 - ▶ Opérations coûteuses, car faites en grande partie dans Java
- ▶ New I/O
 - ▶ Permet de programmer des I/Os performantes sans avoir recours à du code natif
 - ▶ Utilise les fonctions du système d'exploitation pour les opérations coûteuses
 - ▶ Manipule les données par bloc
- ▶ Introduit en Java 1.4
- ▶ Ré implémentation de java.io sur java.nio

Buffer - Channels

- ▶ Toutes les données sont manipulées à travers un buffer
 - ▶ Plus de manipulation directe comme en java.io
- ▶ Conceptuellement, c'est une sorte de tableau
- ▶ Chaque type primitif à un type de buffer associé
 - ▶ ByteBuffer
 - ▶ CharBuffer ...
- ▶ Un canal (Channel) est un objet servant à lire ou écrire des données
- ▶ Il ne manipule que des Buffers
 - ▶ Lire un canal provoque donc l'écriture dans le buffer
- ▶ Contrairement aux flots, les canaux sont bi-directionnels
 - ▶ Un canal peut être ouvert en lecture, en écriture, ou les deux
 - ▶ Plus proche du système d'exploitation
- ▶ En pratique, on manipule un channel qui manipule un buffer

Buffers

- ▶ L'état d'un buffer est contrôlé par 3 variables dont la sémantique dépend de l'opération (lecture/écriture)
- ▶ position :
 - ▶ Position où seront mis les prochains éléments lus
 - ▶ Position du prochain élément à écrire
- ▶ limit :
 - ▶ Quantité de données pouvant être lues
 - ▶ Espace libre dans le buffer pour la prochaine écriture
- ▶ capacity : quantité maximale d'information pouvant être stocké
- ▶ C'est en fait un buffer circulaire sur un tableau
- ▶ La création d'un buffer se fait avec une méthode statique
 - ▶ `allocate(int capacity)`
 - ▶ `allocateDirect(int capacity)` : optimise la lecture/écriture au prix d'une création plus coûteuse

Buffers – get()

- ▶ L'accès direct aux données d'un buffer se fait avec la méthode `get()`
- ▶ Plusieurs versions (ex: `ByteBuffer`)
 - ▶ `byte get()`
 - ▶ `ByteBuffer get(byte dst[])`
 - ▶ `ByteBuffer get(byte dst[], int off, int len)`
 - ▶ `byte get(int index)`
- ▶ Les 3 premiers `get` sont relatifs, ils tiennent compte de `limit` et `position`
- ▶ La 4ème est absolue, elle ne tient pas compte des variables précédentes, et ne les modifie pas

Buffers – put()

- ▶ L'écriture de données dans un buffer se fait avec les méthodes put()
- ▶ Plusieurs versions (ex: ByteBuffer)
 - ▶ `ByteBuffer put(byte b)`
 - ▶ `ByteBuffer put(byte src[])`
 - ▶ `ByteBuffer put(byte src[], int off, int len)`
 - ▶ `ByteBuffer put(ByteBuffer src)`
 - ▶ `ByteBuffer put(int index, byte b)`
- ▶ Les 4 premiers put sont relatifs, ils tiennent compte de limit et position
- ▶ La 5ème est absolue, elle ne tient pas compte des variables précédentes, et ne les modifie pas

Buffers – clear()/flip()/rewind()

- ▶ La sémantique de position et limit change suivant l'opération (lecture ou écriture)
- ▶ Il faut donc indiquer à un buffer quelle opération nous allons faire
 - ▶ Méthode `clear()` pour placer le buffer en écriture (lecture sur un canal)
 - ▶ Méthode `flip()` pour placer le buffer en lecture (écriture sur un canal)
 - ▶ Méthode `rewind()` pour relire les données déjà lues

Exemple

```
FileInputStream fin = new FileInputStream(« lecture.txt »);
FileOutputStream fout = new FileOutputStream(« ecriture.txt »);
FileChannel fcin = fin.getChannel();
FileChannel fcout = fout.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(1024);

while (true) {
    buffer.clear();
    int r = fcin.read(buffer);
    if (r == -1) {
        break;
    }
    buffer.flip();
    fcout.write(buffer);
}
```

Réseau en nio

- ▶ Fonctionne comme les autres opérations en nio
 - ▶ Utilise des channels (SocketChannel)
 - ▶ Utilise des buffers
 - ▶ Les channels sont créés à partir de streams
- ▶ Nouveauté
 - ▶ I/O Asynchrones
- ▶ L'appelant n'est plus bloqué lors d'un read() ou d'un write() ou de toute autre méthode
- ▶ Fonctionnement par évènements
 - ▶ On enregistre son intérêt pour un événement (arrivée de données, nouvelle connexion...)
 - ▶ Le système nous appelle quand un événement se produit
 - ▶ Beaucoup plus efficace que du polling
- ▶ Enlève le besoin de gérer les connexions avec des threads

Selector et SelectionKey

- ▶ Permet de multiplexer des `SelectableChannel`
- ▶ On indique au selector
 - ▶ Une liste de channels
 - ▶ Une liste d'évènements
- ▶ Ces informations sont maintenues dans une `SelectionKey` sous forme de couples (canal, evt)
- ▶ Il nous indique quels évènement se produit sur quel canal
- ▶ Construction par méthode statique
 - ▶ `Selector.open()`
- ▶ L'enregistrement avec les méthodes du channel
 - ▶ Méthode `register()`
 - ▶ Prend en paramètre le selector et la `SelectionKey` qui nous intéresse

Selector et SelectionKey

- ▶ Objet relativement complexe en théorie, en pratique, très simple à utiliser
- ▶ On ne fabrique jamais directement une `SelectionKey`
- ▶ On indique au selector l'évènement qui nous intéresse
- ▶ 4 evts, champs statiques
 - ▶ `SelectionKey.OP_ACCEPT`
 - ▶ `SelectionKey.OP_CONNECT`
 - ▶ `SelectionKey.OP_READ`
 - ▶ `SelectionKey.OP_WRITE`

Selector et SelectionKey

- ▶ On peut attendre un évènement avec la méthode `select()` d'un selector
 - ▶ Appel bloquant (!?)
 - ▶ Mais permet de surveiller plusieurs canaux en même temps
- ▶ La liste des évènements actifs est un `Set<SelectionKey>`
 - ▶ Obtenu avec la méthode `selectedKeys`
 - ▶ En pratique, on parcourt cette liste pour traiter les évènements qu'on supprime
 - ▶ Le canal ayant provoqué l'évènement est accessible avec la méthode `channel()` de `SelectionKey`

ServerSocketChannel

- ▶ Représentation sous forme de channel des sockets coté serveur
- ▶ Partielle
 - ▶ Toujours nécessaire de faire appel à la socket pour certaines opérations
- ▶ Construction par méthode statique
 - ▶ Design Pattern Factory
 - ▶ `ServerSocketChannel.open()`
- ▶ Bloquante par défaut
 - ▶ Utiliser `configureBlocking(boolean)`
- ▶ Il faut faire un bind explicite sur la socket, accessible avec la méthode `socket()`
- ▶ Attention, `accept()` est non bloquant
 - ▶ On crée donc un selector et on s'y enregistre

Exemple

```
Selector selector = Selector.open();
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking(false);
ssc.socket().bind(new InetSocketAddress(2048));
ssc.register(selector, SelectionKey.OP_ACCEPT);
selector.select();
Iterator it = selector.selectedKeys().iterator();
while (it.hasNext()) {
    a
    it.remove();
    if (sel.isAcceptable()) {
        ssc = (ServerSocketChannel) selKey.channel();
        SocketChannel sc = ssc.accept();
        ..... }}
}
```

SocketChannel

- ▶ Représentation sous forme de channel des sockets coté client
- ▶ Construction par méthode statique
 - `SocketChannel.open()`
 - `SocketChannel.open(InetSocketAddress)`
- ▶ Bloquante par défaut
 - ▶ Utiliser `configureBlocking(boolean)`
- ▶ Utilisation d'un selector avec des `SelectionKey`
`OP_CONNECT`, `OP_READ` et `OP_WRITE`
- ▶ Supporte `read/write` avec des `ByteBuffer`

Conclusion

- ▶ Mécanisme très puissant et très flexible
- ▶ Améliore les performances de Java en I/O
 - ▶ Meilleure adéquation avec l'OS
 - ▶ Moins de Threads pour gérer des connexions multiples avec les sockets
- ▶ Programmation Sockets
 - ▶ Apparemment plus complexe dans l'initialisation
 - ▶ Mais on « économise » sur le code hors sockets