CPS — Programmation par composants

© 2019- Jacques Malenfant

Master informatique, spécialité STL – UFR 919 Ingénierie
Sorbonne Université

Jacques.Malenfant@lip6.fr

CC BY NC ND

Cours 4

Composants parallèles en BCM

BY NC ND

Parallélisme et concurrence

- La programmation parallèle et concurrente traite à la fois :
  - de l'introduction dans les programmes d'une capacité d'exécution parallèle de séquences d'instructions par des :
    - *threads* : fils d'exécution partageant un espace mémoire commun ;
    - *processus* : qui *a priori* ne partagent pas de mémoire mais le peuvent *ponctuellement*.
  - et de la *concurrence* des accès aux données et ressources *partagées* pour les modifier en *cohérence*, *malgré* leur parallélisme.
- Le parallélisme est important à deux principaux titres :
  1. La capacité pour les programmes de profiter *réellement* du parallélisme offert par les *processeurs multi-cœurs*.
  2. La programmation d'architectures logicielles capables de gérer de *gros flux de données et de requêtes* grâce à l'utilisation de groupes de *threads* à taille fixe ou variable.
- Les composants BCM peuvent avoir plusieurs *threads* ce qui nécessite d'en maîtriser l'utilisation et les implications.
- Pour cela, il faut revenir aux principes de la programmation parallèle et concurrente.

CC BY NC ND

Gestion du parallélisme et concurrence en BCM4Java

- BCM4Java adopte un modèle de programmation parallèle et concurrente qui exige une gestion *explicite* des *threads* (création, synchronisation, sections critiques, etc.).
- Mais cette gestion peut être *confinée* au sein des composants :
  - Les *threads* sont créés et gérés par chaque composant.
  - Ils ne sont pas admis à exécuter le code ou à accéder aux données internes des autres composants, sauf exception pour les composants (partiellement) passifs.
  - Les problèmes induits par l'utilisation de plusieurs *threads* à l'intérieur d'un composant sont donc résolus *à l'intérieur* de ce dernier.
- On obtient de ceci des propriétés intéressantes :
  - Classes Java : abstraction de la concurrence (sections critiques).
  - Composants BCM : abstraction de la concurrence *et du parallélisme*.
- Ainsi, un composant gérant correctement son parallélisme et sa concurrence internes peut protéger cette propriété en empêchant tout autre parallélisme et accès concurrents non protégés en son sein.

cc BY NC ND

Plan

cc BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre la relation entre composants et *threads* en BCM.
   - Comprendre les notions de composants passifs et actifs.
   - Comprendre la gestion des *threads* par groupes proposée par BCM.
2. Compétences à acquérir
   - Savoir utiliser les *threads* standards des composants.
   - Savoir créer de nouveaux groupes de *threads* dans les composants et les utiliser pour exécuter les appels et les tâches.

BY NC ND

Composants passifs versus actifs

- Composant passif : qui ne possède pas de *threads*.
  - le code du composant est exécuté par les *threads* des *autres composants*, ce qui peut mener à l'exécution de ses services par plusieurs *threads* extérieurs (d'autres composants) à la fois.
- Composant actif : qui possède au moins un *thread*.
  - Généralement, le code du composant n'est exécuté que par ses propres *threads* (c'est le précepte par défaut de BCM).
  - On distingue :
    - serveur pur : n'utilise ses *threads* que pour répondre aux requêtes des autres composants et se met en sommeil entre les appels ;
    - acteur pur : utilise ses *threads* pour exécuter uniquement ses propres tâches (souvent auto-attribuées via sa méthode `execute`) ;
    - mixte : à la fois serveur et acteur, selon le déroulement de son exécution.
- L'exécution des services du composants par ses *threads* ou le fait d'autoriser les *threads* des autres composants à exécuter ses services dépend de la façon de les appeler dans le composant ou dans ses ports entrants (nous y revenons plus loin).

cc BY NC ND

Groupes de *threads* standards dans les composants

- Les composants peuvent avoir deux types de groupes de *threads* offerts par le *framework* `ExecutorService` du *package* standard `java.util.concurrent` :
  1. des groupes *classiques*, exécutant les requêtes ou les tâches *dès* qu'un *thread* devient disponible ;
  2. des groupes « *ordonnançables* », qui peuvent aussi exécuter des requêtes ou des tâches en les démarrant à des *instants précis* (dans le futur) en temps réel.
- Deux groupes de *threads* standards peuvent être facilement créés dans chaque composant dont les tailles sont déterminées par les paramètres des constructeurs d'`AbstractComponent` :
  - le premier déterminant le nombre de *threads* classiques,
  - le second le nombre de *threads* ordonnançables,
- Le total des deux exprime le nombre maximal de requêtes et de tâches que le composant peut exécuter en (pseudo-)parallèle.

CC BY NC ND

Création de nouveaux groupes de *threads*

- Inconvénient à ne disposer que de deux groupes de *threads* :
  - Une fois les requêtes et les tâches réparties entre les deux groupes, ce sont les ordonnanceurs de ces groupes qui décident quand ces *threads* vont s'exécuter.
  - Dans certains cas, un type de requêtes peut saturer les groupes de *threads* au détriment d'autres types de requêtes (ex.: accès en lecture versus écriture).
- Un composant BCM peut créer ses propres groupes de *threads* et les utiliser pour (mieux) répartir ses tâches et ses requêtes.
- La méthode à employer (souvent dans le constructeur) est :

```
protected int createNewExecutorService(
                    String uri, int nbThreads, boolean schedulable)
```

- Cette méthode crée un nouveau groupe de *threads*, reconnu au sein du composant par l'URI fournie et l'indice retourné, avec le nombre de `threads` donné qui sont ordonnançables ou non selon que le dernier paramètre est vrai ou faux.

cc BY NC ND

Utilisation des groupes de *threads* I

- Au sein d'un composant, on peut désigner les groupes de *threads* de deux manières :
  1. Par leur URI passée en paramètre à leur création.
  2. Par un indice attribué et retourné à leur création, aussi accessible par `getExecutorServiceIndex(String):int`.
- Notez que les groupes de *threads* standards ont des URIs standards *réservées*, désignées par des constantes dans `AbstractComponent` :
  - `STANDARD_REQUEST_HANDLER_URI` : URI du groupe de *threads* classiques ;
  - `STANDARD_SCHEDULABLE_HANDLER_URI` : URI du groupe de *threads* ordonnançables.
- Les méthodes déjà vues pour soumettre les requêtes (`handleRequest`, *etc.*) et les tâches (`runTask`, *etc.*) existent également en deux autres versions :

BY NC ND

Utilisation des groupes de *threads* II

- des versions prenant en premier paramètre l'URI du groupe de *threads* à utiliser ;
  Ex.:
  ```
  static final String READING = "reading";
  handleRequest(READING, serviceCall);
  ```
- des versions prenant en premier paramètre l'indice du groupe de *threads* à utiliser (un peu plus efficaces).
  Ex.:
  ```
  int READING_INDEX = getExecutorServiceIndex(READING);
  handleRequest(READING_INDEX, serviceCall);
  ```

- D'autres méthodes existent permettant de vérifier si un groupe de *threads* existe pour une URI donnée, si un indice est valide pour désigner un groupe de *threads* et si un groupe de *threads* d'une URI donnée est ordonnançable on non.
- Actuellement, tous les groupes de *threads* créés au sein d'un composant sont arrêtés en même temps, lors de sa terminaison par `shutdown` ou `shutdownNow`.

BY NC ND

Plan

cc BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les différents modes d'appel entre composants et leurs conséquences sur l'utilisation des *threads*.
   - Comprendre le lien entre les modes *d'appel* entre composants et l'introduction de parallélisme *inter-composants* *effectif*.
   - Comprendre le lien entre les modes *d'exécution* des tâches internes aux composants et l'introduction de parallélisme *intra-composant* *effectif*.
2. Compétences à acquérir
   - Savoir comment déterminer le nombre de *threads* à utiliser pour un composant.
   - Savoir programmer les composants en utilisant les différents modes d'exécution et d'appel disponibles en BCM.
   - Savoir produire du parallélisme inter-composants en BCM par l'exécution asynchrone de requêtes et de tâches.
   - Savoir utiliser le *framework* `ExecutorService` de Java pour produire du parallélisme intra- et inter-composants.

BY NC ND

Modes d'exécution de code en BCM I

- Les composants BCM s'appellent les uns les autres pour exécuter leurs services.
- BCM propose trois façons de faire exécuter du code dans un composant :
  - **Mode séquentiel** ; le *thread* appelant exécute directement le code appelé sans utiliser d'autre *thread*.
  - **Mode synchrone** : l'exécution du *thread* appelant est bloqué jusqu'à ce que le *thread* appelé ait fini l'exécution du code soumis et retourné son résultat (ou simplement le contrôle si le type de retour est `void`).
  - **Mode asynchrone** : l'appelant exécute son appel puis continue son exécution sans attendre ni la fin de l'appel ni son résultat.
- Ces modes sont applicables autant dans les ports entrants passant les appels de services provenant des autres composants qu'au sein du composant lui-même pour ses appels internes exécutés comme tâches.

cc BY NC ND

Modes d'exécution de code en BCM II

- Pour les appels inter-composants, selon le mode utilisé par les ports entrants, on pourra parler :
  - d'**appel séquentiel** lorsque le *thread* du composant appelant exécute directement le code du composant appelé (généralement pour les composants passifs) ;
  - d'**appel synchrone** lorsque le code du composant appelé est exécuté par un de ses *threads* mais que le *thread* du composant appelant est *bloqué* dans l'attente de la fin d'exécution du code appelé et du retour de son résultat ;
    on peut aussi parler ici d'exécution *séquentialisée* dans la mesure où même si deux *threads* sont impliqués, ils ne s'exécutent jamais en (pseudo-)parallèle ;
  - d'**appel asynchrone** lorsque le code du composant appelé est exécuté par un de ses *threads* et que le *thread* du composant appelant poursuit immédiatement son exécution sans attendre la fin de l'exécution du code appelé ni le retour d'un résultat.

cc BY NC ND

Passage des appels externes à un composant serveur I

- Rappel : lors d'un appel inter-composant, le composant appelant exécute avec son propre *thread* l'appel de son port sortant, du connecteur puis du port entrant côté serveur ; ensuite, l'appel est passé au composant appelé par le port entrant.
- Si le composant appelé est actif, on peut lui faire exécuter le service avec ses propres *threads* par :
  - `handleRequest` : appel synchrone bloquant le *thread* de l'appelant jusqu'à ce que le *thread* de l'appelé ait terminé et retourné le résultat (s'il y en a un) ;
  - `runTask` : appel asynchrone libérant immédiatement le *thread* de l'appelant et ne retournant pas de résultat (`void`).
- Pour obtenir un appel séquentiel, il y a deux façons de procéder :
  - Si le composant est passif, `handleRequest` fera automatiquement exécuter le code par le *thread* du composant appelant (`runTask` n'est pas autorisé pour les composants passifs).

cc BY NC ND

Passage des appels externes à un composant serveur II

- Que le composant soit passif ou actif, faire un appel Java directement sur la référence à l'objet Java le représentant fera exécuter l'appel par le *thread* appelant, typiquement, dans un port entrant : `this.getOwner().method(...);`
  (rare, car introduit potentiellement du parallélisme non maîtrisé).

- Sauf si le composant appelé est passif, le choix entre appel synchrone ou asynchrone est fait *lors de la programmation des ports entrants* par l'utilisation de l'un ou l'autre des procédés précédents pour passer la requête à leur composant.
  - L'appelant n'est donc *pas maître* du type d'appel qui est utilisé puisque le choix se fait côté appelé (*i.e.*, port entrant).
  - Si l'appelant veut être sûr de ne pas se bloquer en attente d'un résultat et poursuivre immédiatement son exécution, il doit user d'autres moyens.
    - Par exemple, il peut créer un nouveau *thread* temporaire, à part, qui fera l'appel, attendra le résultat et sera possiblement bloqué, évitant qu'un *thread* « normal » du composant soit bloqué.

cc BY NC ND

Exemples des trois types d'appels vu du port entrant

- Reprenons l'exemple du composant `Calculator` définissant `addition` appelable via `CalculatorServicesCI`.
- Pour un appel séquentiel, on écrira dans le port entrant :

```
public double add(double x, double y) throws Exception {
  return ((Calculator)this.getOwner()).addition(x, y);
}
```

- Pour un appel synchrone, on écrira :

```
public double add(...) throws Exception {
  return this.getOwner().handleRequest(o -> ((Calculator)o).addition(x, y));
}
```

- Et pour un appel asynchrone d'une méthode `m` sur un composant de type `C`, on écrira :

```
public void m(...) throws Exception {
  this.getOwner().runTask(o -> ((C)o).m(...));
}
```

  - Pour un appel asynchrone, le service `m` devrait faire des *effets de bord* car l'appel asynchrone ne retourne pas de résultat.

cc BY NC ND

Modes d'exécution et parallélisme effectif

- Il ne suffit pas d'avoir plusieurs *threads* pour avoir du parallélisme effectif, il faut encore les activer en leur faisant exécuter différentes séquences de code en parallèle.
- Le mode d'exécution du code a un impact direct sur le *parallélisme réel* (potentiel) qui va apparaître au sein d'un composant.
  - Le mode d'exécution séquentiel n'introduit pas de parallélisme réel supplémentaire car c'est le *thread* du composant appelant qui exécute le code du composant appelé.
  - Le mode d'exécution synchrone n'introduit pas non plus de parallélisme réel supplémentaire entre l'appelant et l'appelé car même si deux *threads* distincts sont utilisés, le *thread* appelant est bloqué tant que le *thread* appelé s'exécute (séquentialisation).
  - Seul le mode asynchrone introduit du parallélisme réel, puisqu'il y aura deux *threads* qui vont s'exécuter, l'appelant et l'appelé, tant que les deux n'ont pas terminé leur exécution.

cc BY NC ND

Contrôle du parallélisme effectif dans un composant

- Pour obtenir un composant dans lequel il n'y aura jamais qu'une seule requête ou tâche qui va s'exécuter, il suffit de ne créer qu'un seul groupe de *threads* avec un seul *thread* et d'exécuter toutes les requêtes et tâches avec ce groupe.
  - Cela *séquentialise* toute l'exécution de code dans le composant, ce qui peut avoir ses vertus (voir prochain cours).
- Pour obtenir de l'exécution parallèle de code au sein d'un composant, on peut créer un ou plusieurs groupes de *threads* totalisant plus d'un *thread* et diriger les exécutions vers tous ces *threads* :
  - par plusieurs exécutions en mode synchrone (`handleRequest`), en même temps, engendrant du parallélisme entre elles ;
  - par des exécutions en mode asynchrone (`runTask`), ce qui produira du parallélisme entre composant appelé et appelant ;
  - par plusieurs exécutions de tâches internes en même temps (voir plus loin), ce qui produira du parallélisme intra-composant.

Attention, cela reste du pseudo-parallélisme, à moins qu'on utilise réellement plusieurs cœurs ou processeurs !

CC BY NC ND

Plan

cc BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les politiques d'ordonnancement des tâches sur les groupes de *threads* selon le *framework* `ExecutorService` de Java.
   - Comprendre les difficultés induites par ces politiques d'ordonnancement dans l'exécution de tâches entre groupes de *threads* et donc en BCM4Java.
   - Comprendre les préceptes et règles de bonne programmation BCM permettant d'éviter les difficultés induites par Java.
2. Compétences à acquérir
   - Savoir déterminer le nombre minimal de *threads* à donner à un composant pour qu'il s'exécute sans blocage.
   - Savoir utiliser des techniques de programmation de base permettant de programmer de telle façon à bien utiliser les *threads* et éviter les interblocages liés aux limitations de l'ordonnancement des groupes de *threads* en Java.

cc BY NC ND

Interblocage par famine de *threads* en BCM I

- L'interblocage en programmation parallèle arrive lorsque deux ou plusieurs *threads* s'exécutant en parallèle s'empêchent les uns les autres de progresser dans leur exécution d'une manière ou d'une autre.
  - On parle aussi, de manière imagée, d'*étreinte fatale*, surtout si tous les *threads* s'interbloquent et que l'ensemble du programme se retrouve bloqué définitivement.
- Par sa façon d'utiliser les *threads*, le mode synchrone en BCM peut engendrer de l'interblocage, ce que l'on peut comprendre en considérant un scénario simple :
  - Deux composants A et B sont créés avec un seul *thread* chacun.
  - Le composant A commence l'exécution d'une requête sur son *thread* qui, à un moment, appelle B de manière *synchrone*.
  - Cet appel résulte en une requête exécutée sur le *thread* de B et le *thread* de A est mis en attente du résultat.

BY NC ND

Interblocage par famine de *threads* en BCM II

- Supposons que B rappelle maintenant A, que se passe-t-il ?
- Cet appel résulte en une requête qui doit être exécutée par le *thread* de A, MAIS il est en attente du résultat à son appel à B.
- *La nouvelle requête sur A ne pouvant être exécutée et B attendant son résultat, il y a interblocage entre le thread de A et celui de B* ;

- C'est une manifestation du « ***thread starvation deadlock*** ».
- Ce petit scénario utilise l'*appel en retour* : une méthode m1 de A appelle une méthode m2 de B qui rappelle une méthode m3 de A avant de terminer son exécution et retourner son résultat.
- Bien sûr, on peut complexifier ce scénario de plusieurs manières :
  - en faisant intervenir une chaîne de composants plus longue dans le cycle : A, B, C, ..., A ;
  - en utilisant des appels récursifs (directs ou indirects), ce qui fait que le cycle devrait se faire un nombre indéterminé de fois.

BY NC ND

Relation avec l'ordonnancement des *threads* I

- Dans le scénario précédent, on est tenté de croire que l'ordonnancement devraient résoudre le problème.
  - L'ordonnancement des *threads* suit généralement la même politique que pour les processus dans un système d'exploitation : *round-robin* avec quantum de temps.
  - Sous Unix, chaque processus reçoit l'autorisation de s'exécuter à tour de rôle pendant un certain quantum de temps ; si le processus est suspendu ou termine son exécution pendant le quantum ou encore le quantum arrive à expiration, l'ordonnanceur passe la main au processus suivant, et ainsi de suite.
- Bien que les groupes de *threads* de Java utilisent ce type d'ordonnancement, le problème d'interblocage par famine de *threads* n'est pas dû à l'ordonnancement des *threads*.
- En fait, le problème découle du mode d'attribution des requêtes/ tâches aux *threads* dans les groupes de *threads* de l'`ExecutorService` :

CC BY NC ND

Relation avec l'ordonnancement des *threads* II

- Pour les groupes de *threads*, les tâches (`Runnable` ou `Callable`) sont mises en file d'attente et ce n'est que lorsqu'un *thread* *termine sa tâche* qu'on lui attribue la prochaine tâche de la file d'attente pour exécution.
- Le *thread* va donc exécuter chaque tâche *jusqu'à terminaison*.
- Si la tâche subit une suspension (par appel synchrone, par exemple), le *thread ne va pas* changer de tâche pour en prendre une autre.
- Si la tâche en cours d'exécution ne peut pas progresser, le *thread* ne sera pas ordonnançable mais reste occupé et bloqué jusqu'à ce qu'il puisse reprendre et terminer la tâche qui lui a été assignée.
- Pour reprendre notre exemple, le *thread* de A occupé mais bloqué ne va pas mettre de côté sa tâche non terminée pour en prendre une autre et comme il n'y a pas d'autre *thread* disponible dans A pour exécuter l'appel de B, ce dernier ne pourra être servi, et donc tout est bloqué.

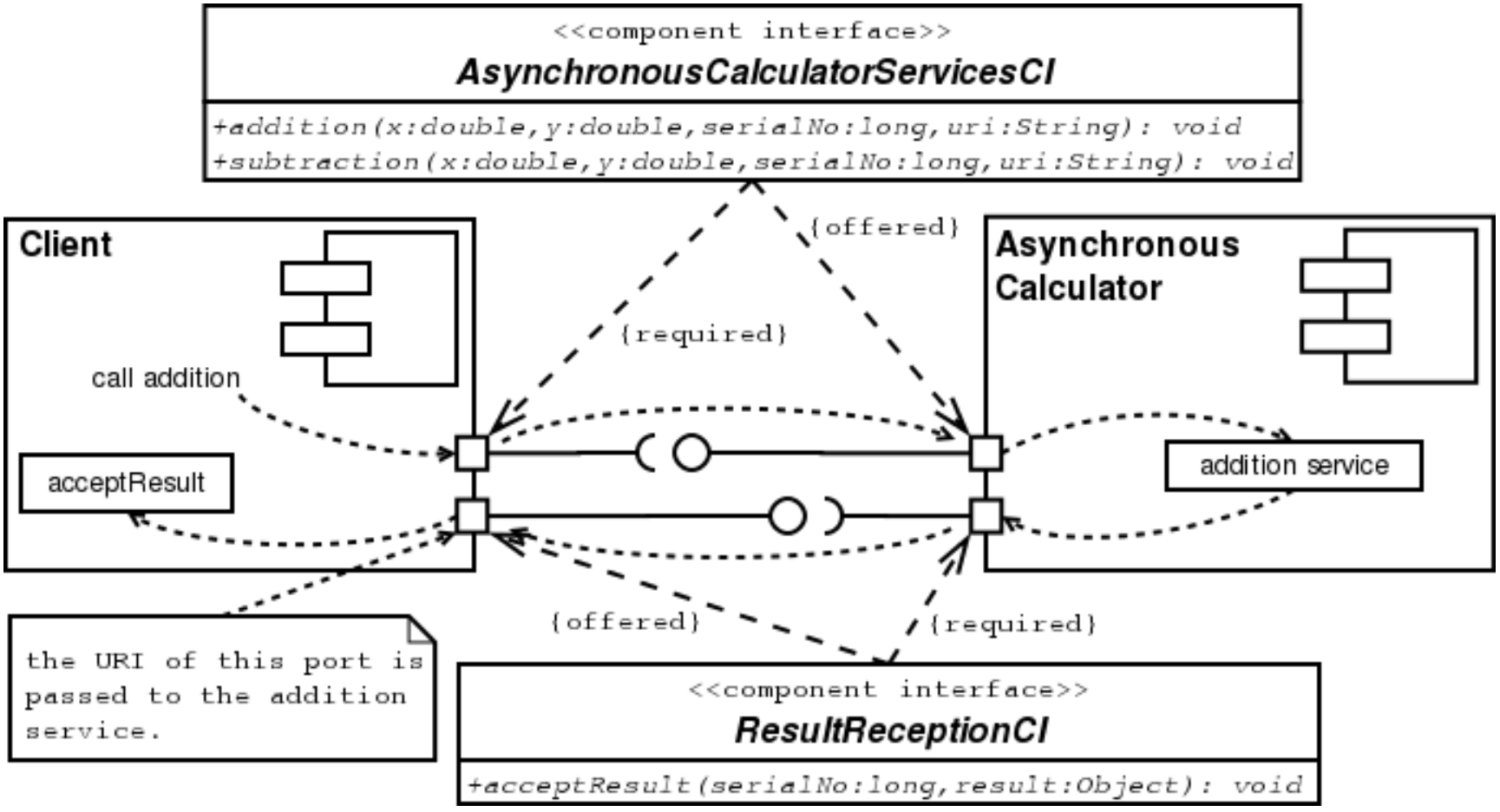
cc BY NC ND

Comment éviter ce type d'interblocages ?

- On peut augmenter le nombre de *threads* de A, certes, mais on ne peut pas toujours facilement borner le nombre nécessaire.
  - Si on a une récursivité directe ou indirecte par exemple, il ne sera pas possible de borner le nombre *threads* nécessaire pour qu'aucun interblocage ne se produise.
- Si on ne peut pas fournir suffisamment de *threads*, il faut faire en sorte de *libérer* les *threads* existants au plus tôt :
  - Lorsqu'il n'y a pas de résultat à attendre, utiliser des appels asynchrones et terminer au plus tôt l'exécution de la méthode appelante pour libérer son *thread*.
  - Si un résultat d'une chaîne d'appels $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n$ doit être retourné à C_1, adopter un style de programmation où :
    - tous les appels de la chaîne sont *asynchrones*, *sans retourner de résultat*, et les méthodes terminent très vite après cet appel ;
    - pour que le composant C_n puisse envoyer à C_1 le résultat final, prévoir dans C_1 une méthode que C_n pourra appeler à la fin de son calcul pour lui passer le résultat de l'appel initial comme paramètre.

cc BY NC ND

Exemple 1 : le calculateur en programmation asynchrone



Pour les curieux, cela s'apparente au style *passage à la continuation* en programmation fonctionnelle. . .

Plan

cc BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Approfondir comment introduire du parallélisme supplémentaire dans les programmes par l'exécution de tâches parallèles et les appels asynchrones.
   - Comprendre comment synchroniser les tâches asynchrones grâce à la notion de variable future et leur rôle.
2. Compétences à acquérir
   - Savoir maîtriser le parallélisme induit pas l'exécution de requêtes et de tâches asynchrones grâce aux variables futures de Java.
   - Apprendre à utiliser ces mécanismes à travers deux exemples complets.

cc BY NC ND

Appel asynchrone avec variable future

- Rappels :
  - Appel synchrone : le *thread* appelant est bloqué jusqu'à la terminaison du code appelé (retour du résultat ou simplement du contrôle pour les méthodes de type `void`).
  - Appel asynchrone : le *thread* appelant n'est pas bloqué, mais il ne peut pas savoir quand le code appelé aura terminé.
- Alternative **en Java** : appel *asynchrone avec variable future*.
  - Une variable future (en Java, qui implante l'interface `Future<T>`) représente un *résultat à venir*, en cours de calcul qui peut être récupéré par un appel à sa méthode `get` :
    - si, au moment de l'appel à `get` le calcul a terminé, le résultat est simplement retourné ;
    - si le calcul n'a pas terminé, le *thread* appelant `get` est bloqué jusqu'à ce que le résultat soit disponible.
- Ce mode produit un parallélisme similaire à celui de l'appel asynchrone entre le code appelant et le code appelé mais il est limité jusqu'au `get` côté appelant et, en plus, il permet de synchroniser les deux *threads* sur le `get`.

BY NC ND

Appels asynchrones avec futurs en BCM4Java

- Pour l'instant, BCM4Java ne permet pas l'appel asynchrone avec variable future *entre composants* et ce pour deux raisons :
  1. Les principales classes standard implantant `Future<T>` ne sont pas sérialisables, et donc pas utilisables avec RMI.[1]
  2. Java a fait le choix de s'en remettre au cadriciel *Java Messages Services* (JMS) pour proposer l'appel asynchrone avec futurs en réparti.
- Par contre, *au sein d'un composant*, il est parfaitement possible d'utiliser des appels asynchrones avec futurs *de Java* pour produire et contrôler du parallélisme *interne* ; les principales méthodes à utiliser sont :
  - `protected <T> Future<T> baselineHandleRequest(ComponentService<T> r)`
    appel asynchrone avec futur sur un groupe de *threads* standards.
  - `protected Future<?> runTaskOnComponent(ComponentTask t)`
    tâche asynchrone avec futur sur un groupe de *threads* standards.
  - Ces deux méthodes ont des versions permettant de viser un groupe de *threads* spécifique (en donnant son URI ou son index).

[1] Une forme limitée existe depuis Java 7 avec la classe `ForkJoinTask<V>` et ses descendantes, mais BCM ne l'utilise pas encore.

cc BY NC ND

Utilisation des appels internes asynchrones

- Idiome standard pour un appel asynchrone interne avec futur :

```
// code dans une méthode m du composant C; aMethod retourne un entier
...
Future<Integer> f =
    this.baselineHandleRequest(     // notez l'utilisation de cette méthode
         o -> ((C)this.getServiceOwner()).aMethod(...));
// partie de code (dans m, par exemple) s'exécutant en parallèle
// avec aMethod
...
// m accède au résultat; bloquant s'il n'est pas encore disponible
int resultat = f.get();
// poursuite de l'exécution de m, fin du parallélisme
...
```

- L'interface `Future<T>` permet aussi de faire d'autres opérations, comme l'annulation de la tâche ou tester si une tâche est terminée.
  (*Ne pas utiliser pour faire de l'attente active !* Voir cours suivant.)

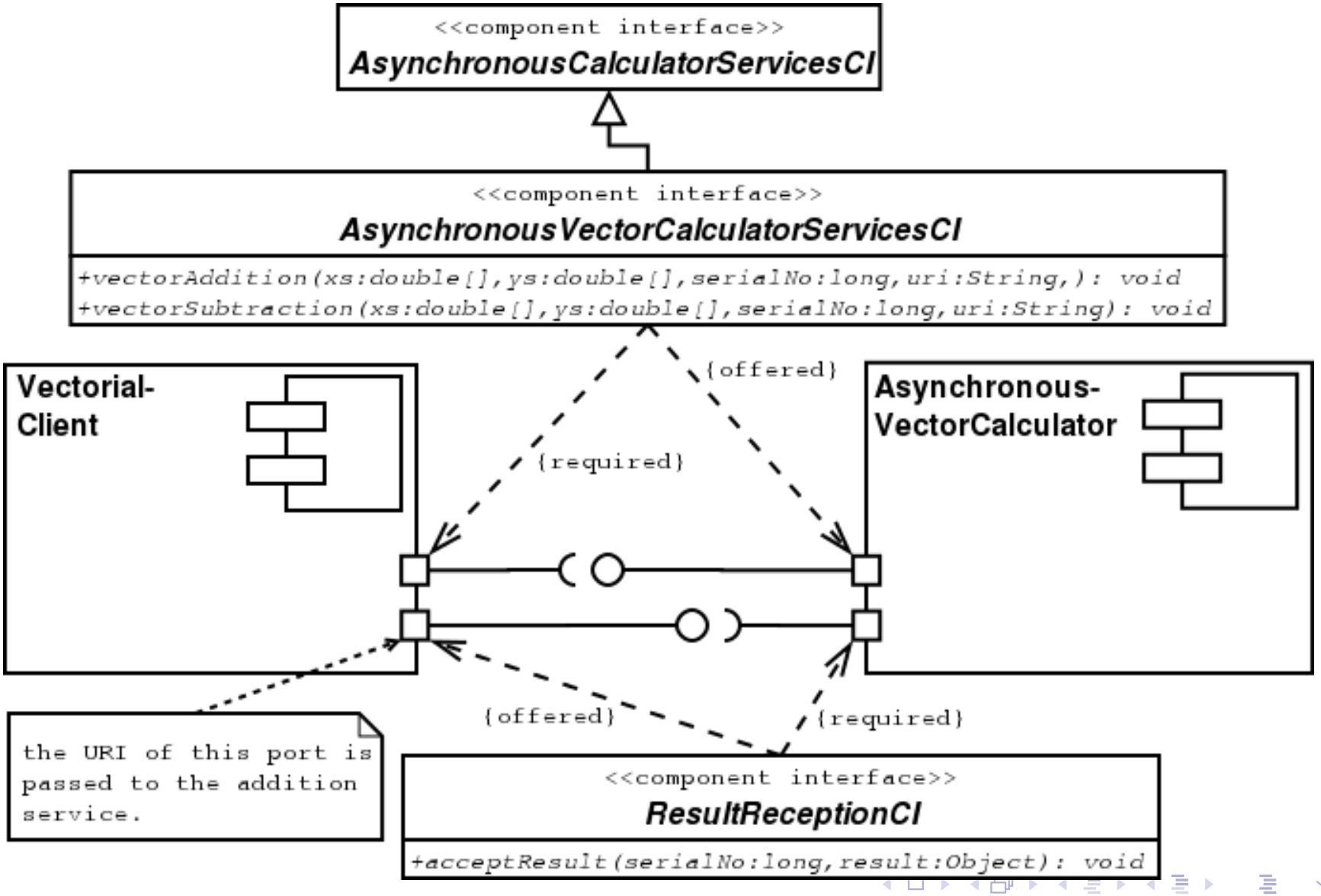
BY NC ND

Exemple 2 : le calculateur vectoriel asynchrone I

- Nous reprenons l'exemple du calculateur asynchrone introduit précédemment pour ajouter un peu de calcul vectoriel.
- L'interface `AsynchronousVectorCalculatorServicesCI` ajoute deux méthodes :
  - `vectorialAddition` qui prend comme opérandes deux tableaux représentant des vecteurs et les additionne position par position.
  - `vectorialSubtraction` qui fait de même mais en les soustrayant.
- Le composant `AsynchronousVectorCalculator` implante ces deux services en calculant toutes les additions (soustractions) en parallèle avec des tâches asynchrones et récupère les résultats grâce aux variables futures avant de les mettre dans le vecteur résultat pour le renvoyer à l'appelant.

BY NC ND

Exemple 2 : le calculateur vectoriel asynchrone II



cc BY NC ND

Idiomes d'attente sur des tâches et requêtes parallèles

- Assez souvent, lorsqu'on lance plusieurs tâches ou requêtes asynchrones en parallèle, on souhaite se synchroniser sur la terminaison de la première ou de toutes ces dernières.
- Les `ExecutorService` de Java proposent deux méthodes pour réaliser facilement ces deux idiomes : `invokeAny` et `invokeAll`.
  - Les deux prennent en paramètre une liste de requêtes (`Callable<T>` *i.e.*, `ComponentRequest<T>` pour BCM, ou `Runnable` *i.e.*, `ComponentTask` pour BCM) et les lancent en parallèle sur le *pool* de *threads*.
  - Les méthodes `invokeAny` retournent le premier résultat *i.e.*, celui retourné par la première requête qui termine et les autres sont *automatiquement annulées*.
  - Les méthodes `invokeAll` retournent une liste de `Future<T>` permettant ensuite de récupérer chacun des résultats par `get`. *Ex.: dans* `AsynchronousVectorCalculator`*, on pourrait utiliser* `invokeAll` *plutôt que faire les* `baselineHandleRequest` *indépendamment les uns des autres.*

BY NC ND

Attentes plus complexes

- `invokeAny` et `invokeAll` ne sont pas adaptées dans le cas où on souhaite avoir tous les résultats mais où on veut les traiter dans l'ordre de leur production (et non dans l'ordre des `get`).
- Java offre d'autres possibilités pour programmer des attentes plus complexes, dont l'`ExecutorCompletionService` :
  - Pour l'utiliser, on crée d'abord une instance `ecs` d'`ExecutorCompletionService` en lui passant un *pool* de *threads*.
  - On soumet ensuite des tâches au *pool mais* en passant par la méthode `ExecutorCompletionService#submit`.
  - Ensuite, l'appel à la méthode `take` sur `ecs` retourne un à un les `Future` de chacune des tâches soumises, dans l'ordre de terminaison des tâches.
- Le prochain exemple montre comment utiliser l'`ExecutorCompletionService`.

CC BY NC ND

Exemple 3 : le calculateur parallèle asynchrone

- Nous reprenons encore l'exemple du calculateur asynchrone introduit précédemment pour ajouter du parallélisme.
- L'interface `AsynchronousParallelCalculatorServicesCI` ajoute deux méthodes :
  - `parallelAddition` qui prend comme opérandes deux tableaux et fait les additionne position par position mais pour les retourner une à une dans l'ordre où elles produisent leur résultat.
  - `parallelSubtraction` fait de même mais avec des soustractions.
- Le composant `AsynchronousParallelCalculator` implante ces deux services en calculant toutes les additions (soustractions) en parallèle avec des tâches asynchrones et récupère les résultats grâce à un `ExecutorCompletionService` pour les renvoyer à l'appelant dans l'ordre de leur production.

BY NC ND

Plan

CC BY NC ND

Objectifs de la séquence

1. Objectifs pédagogiques
   - Comprendre les notions de fonctions, d'interfaces fonctionnelles, de λ-expressions et de *streams* ajoutés depuis Java 1.8.
   - Comprendre le lien entre ces notions et les concepts de la programmation fonctionnelle.
   - Comprendre l'idiôme de calcul fonctionnel *map/reduce*.
   - Comprendre le fonctionnement général des *streams* parallèles et de leur exécution sur les *pools* de *threads* de Java.
2. Compétences à acquérir
   - Connaître et savoir utiliser les principales interfaces fonctionnelles standards de Java pour créer des fonctions et des λ-expressions.
   - Savoir créer de nouvelles interfaces fonctionnelles et des fonctions à partir de celles-ci.
   - Savoir créer des *streams* à partir de collections et de tableaux.
   - Savoir utiliser les calculs *map/reduce* sur les *streams*.
   - Savoir utiliser des *streams* parallèles en s'appuyant soit sur le *pool* de *threads* `ForkJoinPool` standard, soit sur un *pool* créé et désigné pour ce faire.

CC BY NC ND

Programmation fonctionelle en Java 1.8 et + I

- Fonctions et λ-expressions ajoutés à Java *i.e.* :
  - Fonctions et λ-expressions implantent des *interfaces fonctionnelles i.e.*, annoté par `FunctionalInterface`.
  - Syntaxe générale des λ-expressions *paramètres* -> *corps*.
    Ex.: `i -> i + 1`, `(x, y) -> x + y`, `() -> 1`.
  - Interfaces fonctionnelles : annotation `FunctionalInterface`.
    - Posée sur une interface, celle-ci devient une interface fonctionnelle.
    - Une interface fonctionnelle possède exactement une méthode abstraite qui permet d'implanter une fonction. Ex.:

```
@FunctionalInterface
public static interface F3I {
  public int apply(int i, int j, int k);
}
```

    - Une fonction est ensuite crée comme une classe implantant l'interface fonctionnelle et donc sa méthode abstraite.
      Ex.: une fonction à trois paramètres :

```
public class Somme3 implements F3I {
  public int apply(int i, int j, int k) { return i + j + k; }
}
```

cc BY NC ND

Programmation fonctionelle en Java 1.8 et + II

- Utilisation pour créer une instance de fonction `Somme3`.
  Ex.: `System.out.println((new Somme3()).apply(1,2,3));`

- Le *package* `java.util.functions` définit de nombreuses interfaces fonctionnelles standards dont :
  - `Function<T,R>` : fonction d'un paramètre de type `T` retournant un résultat de type `R`.
  - `Predicate<T>` : fonction d'un paramètre de type `T` retournant un booléen.
  - `BiFunction<T,U,R>` : fonction de deux paramètres de types `T` et `U` retournant un résultat de type `R`.
  - `BinaryOperator<T>` : restriction de `BiFunction<T,U,R>` où les deux paramètres et le résultat sont tous de type `T`. `R`.

- Exemples :
  - `Function<Integer,Double>` : `x -> Math.pow(x, 2.0)`
  - `Predicate<Integer>` : `i -> i % 2 == 0`
  - `BiFunction<String,Character,Integer>` :

    ```
    (s, c) -> Integer.parseInt(s + c)
    ```

  - `BinaryOperator<Double>` : `(x, y) -> x + y`

BY NC ND

Les « *streams* » de Java 1.8 et +

- Les *streams* ou *flux de données* ont été ajoutés à Java 1.8 dans l'esprit d'une extension fonctionnelle à Java.
- Comme un itérateur, un flux n'est pas une structure de données mais plutôt un parcours *linéaire* d'un ensemble de données.
  - Un flux a une source de données et il offre des méthodes pour parcourir et traiter ces données.
  - L'hypothèse générale est un traitement *fonctionnel*, *sans effet de bord* sur les données, *consommable* (ne sert qu'une fois) et paresseux (*lazy*) si le résultat attendu n'exige pas de parcourir toutes les données.
- L'intégration des flux dans Java est complétée par l'introduction de méthodes sur des classes standards pour en faire des sources de données : `Collection::stream`, `Collection::parallelStream`, `Stream::of`, `Arrays.stream`, etc.
- L'implantation des flux, en grande partie réalisée en boîte noire, s'appuie sur la notion de *pipeline de données*.

CC BY NC ND

Utilisation des « *streams* »

- Construction d'un *stream*, principaux cas :
  - `Collection<E>::stream` crée un flux à partir du contenu d'une collection.
  - L'interface `Stream` définit la méthode `static <T> Stream<T> of(T... values)` qui peut prend un tableau et retourner un flux ;
    Ex.: `Stream<Integer> intStream = Stream.of(new Integer[]{1, 2, 3, 4, 5});`
- Principales méthodes d'intérêt de `Stream<T>` :
  - `Stream<T> filter(Predicate<? super T> p)`
  - `<R> Stream<R> map(Function<? super T,? extends R> f)`
  - `<U> U reduce(U identity, BiFunction<U,? super T,U> reductor, BinaryOperator<U> combiner)`
- Exemple : somme des carrés des entiers pairs

```
Stream.of(new Integer[]{1, 2, 3, 4, 5})
    .filter(i -> i % 2 == 0)          // retire les valeurs impaires
    .map(i -> Math.pow(i, 2.0))       // met au carré, sous forme de doubles
                                      // fait la somme des carrés en int
    .reduce(0, (u, d) -> u + d.intValue(), (u1, u2) -> u1 + u2);
```

cc BY NC ND

Les « *streams* » parallèles

- Construction d'un flux parallèle :
  - La méthode `parallel` de l'interface `BaseStream<T,S extends BaseStream<T,S> >` (héritée par `Stream<T>`) « transforme » un flux en flux parallèle.
    Ex.: `Stream<Integer> parIntStream = Stream.of(new Integer[]{1, 2, 3, 4, 5}).parallel();`
- Les méthodes balayant ce flux s'exécutent alors en parallèle.
  - Le parallélisme est implanté par le framework `SplitIterator` qui lui-même utilise les *threads* de `ForkJoinPool`.
  - Par défaut, le *pool* standard `ForkJoinPool.commonPool()` est utilisé, mais il est possible de désigner un autre *pool*, comme illustré au transparent suivant.
  - L'intérêt des *pools* spécifiques est de contrôler le degré de parallélisme et l'allocation des *threads* entre les tâches d'une application :
    - en limitant le nombre de *threads* de chacun de ces *pools* ;
    - en arbitrant entre les nombres de *threads* alloués à chacun des *pools*.

cc BY NC ND

Streams parallèles sur *pool* désigné

```
// création d'un pool limité à 2 threads
ForkJoinPool fjp = new ForkJoinPool(2);
Stream<Integer> parIntStream =
  Stream.of(new Integer[]{1, 2, 3, 4, 5}).parallel();
// soumission d'une tâche avec récupération de sa référence
ForkJoinTask<Integer> fjt =
  fjp.submit(
    () -> parIntStream
                .filter(i -> i % 2 == 0)
                .map(i -> Math.pow(i, 2.0))
                .reduce(0, (u, d) -> f(u, d), (u1, u2) -> u1 + u2));
try {
  // récupération du résultat avec synchronisation
  System.out.println(fjt.get());
} catch (InterruptedException | ExecutionException e) {
  e.printStackTrace();
}
```

cc BY NC ND

Exécution I

- Exécution avec trace de :

```
Stream.of(new Integer[]{1, 2, 3, 4, 5}).filter(i -> i % 2 == 0)
    .map(i -> Math.pow(i, 2.0))
    .reduce(0, (u, d) -> u + d.intValue(), (u1, u2) -> u1 + u2);
```

- donne :

```
Thread no 1 executes selector(1) giving false
Thread no 1 executes selector(2) giving true
Thread no 1 executes processor(2) giving 4.0
Thread no 1 executes reductor(0, 4.0) giving 4
Thread no 1 executes selector(3) giving false
Thread no 1 executes selector(4) giving true
Thread no 1 executes processor(4) giving 16.0
Thread no 1 executes reductor(4, 16.0) giving 20
Thread no 1 executes selector(5) giving false
Executing verbose map/reduce on
   Stream.of(new Integer[]{1, 2, 3, 4, 5}) gives: 20
```

cc BY NC ND

Exécution II

- Exécution du même calcul sur un `ForkJoinPool` avec 2 *threads* donne :

```
Thread no 11 executes selector(7) giving false
Thread no 11 executes selector(6) giving true
Thread no 11 executes processor(6) giving 36.0
Thread no 11 executes reductor(0, 36.0) giving 36
Thread no 11 executes combinator(36, 0) giving 36
Thread no 10 executes selector(3) giving false
Thread no 11 executes selector(9) giving false
Thread no 11 executes selector(10) giving true
Thread no 10 executes selector(5) giving false
Thread no 11 executes processor(10) giving 100.0
Thread no 11 executes reductor(0, 100.0) giving 100
Thread no 10 executes selector(4) giving true
Thread no 11 executes combinator(0, 100) giving 100
Thread no 10 executes processor(4) giving 16.0
Thread no 11 executes selector(8) giving true
Thread no 10 executes reductor(0, 16.0) giving 16
```

cc BY NC ND

Exécution III

```
Thread no 11 executes processor(8) giving 64.0
Thread no 10 executes combinator(16, 0) giving 16
Thread no 11 executes reductor(0, 64.0) giving 64
Thread no 11 executes combinator(64, 100) giving 164
Thread no 11 executes combinator(36, 164) giving 200
Thread no 10 executes combinator(0, 16) giving 16
Thread no 10 executes selector(1) giving false
Thread no 11 executes selector(2) giving true
Thread no 11 executes processor(2) giving 4.0
Thread no 11 executes reductor(0, 4.0) giving 4
Thread no 11 executes combinator(0, 4) giving 4
Thread no 11 executes combinator(4, 16) giving 20
Thread no 11 executes combinator(20, 200) giving 220
verbose parallel map/reduce gives: 220
```

- Notez l'ordre plutôt imprévisible dans lequel sont exécutés les appels au filter (`selector`) puis les dépendances entre les autres fonctions.

BY NC ND

Activités à réaliser avant le prochain TME

Remarque importante

La lecture de code en CPS est une activité aussi importante pour acquérir les concepts et les techniques de programmation présentées que la préparation des examens dans d'autres UE. N'oubliez pas que l'évaluation du projet porte entre autres choses sur la qualité de votre code. De plus, il y a beaucoup de technicité à acquérir dans l'étude des exemples fournis, des compétences recherchées sur la marché du travail.

1. Récupérer, bien lire, puis essayer les exemples proposés dans le cours.
2. (Re)Lire la seconde partie du cahier des charges du projet pour faire le lien avec les concepts introduits dans ce cours.
3. Lire la documentation Javadoc des packages `java.util.function` et `java.util.stream`.
4. Récupérer l'exemple des flux, la classe `TestStreams`, à lire, comprendre et exécuter en changeant les nombres de valeurs dans la source de données et de *threads* dans le `ForkJoinPool`.

BY NC ND