Faculté des Sciences
Aix*Marseille Université

Année universitaire

Site : ☐ Luminy ☐ St-Charles ☐ St-Jérôme ☐ Cht-Gombert ☐ Aix-Montperrin ☐ Aubagne-SATIS
Sujet de : ☐ 1er semestre ☐ 2ème semestre ☐ Session 2 Durée de l'épreuve :
Examen de : L2 Nom du diplôme : Licence d'informatique
Code du module : SIN3U02 Libellé du module : Programmation 2
Calculatrices autorisées : NON Documents autorisés : OUI

---

1 Consignes

- Téléchargez l'archive `examenSession1.zip` au lien suivant : `https://ametice.univ-amu.fr/mod/resource/view.php?id=1249695`. Décompressez l'archive et décompresser là. Vous devriez obtenir un répertoire `examenSession1` ayant un *package* `cluster`. Ouvrer le projet avec IntelliJ grâce à l'option `open` en sélectionnant le dossier `examenSession1`. Toutes les classes que vous aurez à créer devront être placées dans le package `cluster`. Ajoutez le projet en tant que projet gradle. Normalement, c'est fait automatiquement en sélectionnant `Use Gradle wrapper task configuration` à la création du projet mais vous pouvez faire manuellement en cliquant droit sur le fichier `build.gradle` et en sélectionnant `add as a gradle project`.
- Ne passez pas trop de temps sur une question. Si vous restez bloqué (code ne compilant pas ou erreur d'exécution que vous n'arrivez pas à compiler) et n'arrivez pas à corriger le problème, passez à la question suivante.
- Toutes les variables de type `List` seront à initialiser avec des objets de type `ArrayList`.
- Il n'y aucune indication dans le sujet sur l'utilisation des mot-clés `final`, `public`, `private` ou `protected`. Pour chaque attributs ou méthodes, ce sera à vous de choisir d'utiliser ou non ces mot-clés.
- le sujet liste les classes, méthodes et attributs à créer mais vous avez le droit de rajouter des classes, des attributs et des méthodes afin d'améliorer la lisibilité du code. Vous serez évalué pas seulement sur le fait que votre code soit correct (fait ce qui est demandé) mais aussi sur le fait qu'il est bien écrit (rappel : toujours coder en anglais).
- Pour cet examen, vous n'avez accès qu'au cours `http://pageperso.lif.univ-mrs.fr/~arnaud.labourel/` et à la documentation de l'API de java : `https://docs.oracle.com/en/java/javase/11/`.
- Lorsque vous avez fini de composer, sélectionnez votre répertoire `examenSession1` contenant votre projet et compressez-le (clic droit puis compresser...) puis déposez l'archive sur le cours AMETICE programmation 2 au lien suivant `https://ametice.univ-amu.fr/mod/assign/view.php?id=1249691`

2 Sujet : Gestion de tâches dans une ferme de machines

Vous allez travailler dans cet examen à la gestion d'une ferme de nœuds (*nodes*) représentant des ordinateurs qui mettent à disposition de la mémoire (*memory*) exprimée en Mégaoctets (Mo) qui peut être consommées par des tâches (*job*). L'objectif est de proposer un ordonnanceur (*scheduler*) qui place un ensemble de tâches sur les nœuds de manière à maximiser l'utilisation de la mémoire.

3 La tâche de calcul : classe `Job` (2 points)

La classe `Job` va nous permettre de représenter une tâche de calcul qui a une quantité de mémoire à consommer sur un nœud. La classe `Job` contiendra les attributs, méthodes et constructeurs suivants :

- un attribut `int memory` : la quantité de mémoire requise par la tâche exprimée en Mo.
- un attribut `int id` : l'identifiant de la tâche. La première tâche créée devra avoir un `id` égal à `0`, la deuxième tâche créée devra avoir un `id` égal à 1, la troisième tâche créée devra avoir un `id` égal à 2 et ainsi de suite.
- un attribut `static int jobCount` : le nombre total de tâches qui ont été créées (mis à jour à chaque appel du constructeur).

— une constructeur `public Job(int memory)` : qui initialise les attributs `memory` et `id` de la tâche. Il devra lever une exception de type `IllegalArgumentException` si on essaie de créer un `Job` avec une mémoire inférieure ou égale à 0.

— une méthode `int getMemory()` qui renvoie la mémoire de la tâche.
— une méthode `static void getJobCount()` qui renvoie le nombre de nœuds créés depuis le dernier appel à `resetJobCount()`.
— une méthode `static int resetJobCount()` : remet à 0 l'attribut `jobCount`. Un `Job` créé après un appel à `resetJobCount` devra avoir un `id` égal à `0`, la deuxième tâche créée devra avoir un `id` égal à 1, la troisième tâche créée devra avoir un `id` égal à 2 et ainsi de suite.
— une méthode `int getId()` : qui renvoie l'`id` du processus.
— une méthode `String toString()` : renvoie un chaîne de caractère représentant la tâche. La chaîne contiendra la chaîne `"Job"`, suivi de l'identifiant de la tâche, suivi de la quantité de mémoire de la tâche entre parenthèses, le tout séparé par des espaces. Par exemple, pour une tâche avec 1000Mo de mémoire et un identifiant égal à 0, un appel à `toString` devra renvoyer la chaîne de caractères `"Job 0 (100Mo)"`.

Créer la classe `Job` avec les attributs et méthodes demandées.

4 Tester la classe `Job` : classe `TestJob` (3 points)

Vous trouverez dans le répertoire `test/java/cluster` une classe de test `TestJob`. Cette classe ne contient qu'un test pour la méthode `toString`. Vous devez donc rajouter des tests pour les autres méthodes. Pour compiler et lancer des tests sur votre programme, il faut passer par l'onglet gradle à droite et cliquer deux fois sur `examenSession1` -> `Tasks` -> `verification` -> `test`.
Vou allez devoir rajouter des méthodes de tests afin de tester les comportements suivants :

— un nœud créé juste après un appel à `resetJobCount()` a un `id` égal à 0.
— un nœud créé avec une certaine mémoire a bien cette mémoire selon l'accesseur (*getter*) correspondant.
— un appel à `getJobCount()` donne bien le nombre de nœuds créés depuis le dernier appel à `resetJobCount()`.

Rajouter les tests demandés en utilisant JUnit ou assertJ.

5 Les nœuds de la ferme de calcul : classe `Node` (2 points)

Une ferme de calcul se compose d'un ensemble de nœuds (*nodes*) auxquels on va affecter des tâches utilisant de la mémoire. Chaque nœud a une capacité de mémoire (quantité de mémoire maximale que peut fournir le nœud) et une quantité de mémoire disponible (égal à la capacité en mémoire du nœud moins la mémoire utilisée par les tâches affectées au nœud). La quantité de mémoire disponible prend donc la valeur 0 quand il n'y a plus de mémoire disponible et la valeur de la capacité quand aucune tâche n'est affectée au nœud. La classe `Node` contiendra les attributs, méthodes et constructeurs suivants :

— un attribut `String name` : le nom du nœud.
— un attribut `int memoryCapacity` : la capacité de mémoire du nœud.
— un attribut `int availableMemory` : la quantité de mémoire disponible dans le nœud.
— un attribut `List<Job> assignedJobs` : la liste des tâches affectées au nœud.
— un constructeur `Node(String name, int memoryCapacity)`qui crée un nœud avec le nom et la capacité de mémoire spécifiés. Le nœud créé n'a aucune tâche affectée.
— une méthode `boolean canAccept(Job job)` renvoie `true` si le nœud a assez de mémoire disponible pour accepter le `job`.
— une méthode `boolean canHandle(Job job)` renvoie `true` si le nœud a une capacité de mémoire suffisante pour gérer le `job`, c'est-à-dire qu'il a une capacité de mémoire supérieure ou égale à la mémoire requise pour le `job`.
— une méthode `void accept(Job job)` : affecte le `job` au nœud et met à jour la quantité de mémoire disponible en conséquence.
— une méthode `int usedMemory()` : renvoie la mémoire consommée par les tâches affectées au Nœud.

— une méthode `String toString()` : renvoie une chaîne de caractères représentant le nœud. La chaîne contiendra la chaîne `"Node"`, suivi du nom du nœud, suivi de la quantité de mémoire utilisée sur la capacité du nœud entre parenthèses, le tout séparé par des espaces. Par exemple, pour un nœud ayant pour nom `Calcul` avec 4000Mo de mémoire utilisée et une capacité de 10000Mo, un appel à `toString` devra renvoyer la chaîne de caractères `"Node Calcul (4000/10000Mo)"`.

— une méthode `void printJobs()` : affiche le nœud et toutes les tâches qui lui ont été affectées, chacun sur une ligne différente. L'exemple suivant vous indique un exemple de ce qui est demandé.

```
package cluster;

public class Main {
        public static void main(String[] args) throws Exception{
                Job job1 = new Job(1000);
                Job job2 = new Job(3000);
                Node node = new Node("Calcul", 10000);
                node.accept(job1);
                node.accept(job2);
                node.printJobs();
        }
}
```

L'exécution du code ci-dessus devra donner l'affichage suivant :

```
Node Calcul (4000/10000Mo)
Job 0 (1000Mo)
Job 1 (3000Mo)
```

Créer la classe ***`Node`*** *avec les attributs et méthodes demandées.*

6 Exception `NotEnoughMemoryException` (2 points)

Il y a pour le moment un problème si on appelle la méthode `accept(Job job)` de la classe `Node` alors que le nœud n'a pas assez de mémoire. Pour éviter cela, vous allez rajouter du code pour déclencher une exception `NotEnoughMemoryException` si le nœud ne dispose pas de la mémoire suffisante pour accepter la tâche fournie. Le but est d'obtenir la levée d'une `Exception` de type `NotEnoughMemoryException` qui aura un message dépendant de la situation (pas assez de mémoire totale ou pas assez de mémoire restante) et qui utilisera des appels à `toString()` sur la tâche et le nœud concerné.
Par exemple, l'affectation d'une tâche nécessitant 1000Mo sur un nœud ayant 10Mo devra lever une exception ayant le message suivant :

```
Node Calcul (0/10Mo) has not enough total memory to handle Job 0 (1000Mo).
```

L'affectation d'une tâche nécessitant 1000Mo sur un nœud ayant 1000Mo de capacité mais aucune mémoire restante devra lever une exception ayant le message suivant :

```
Node Calcul (1000/1000Mo) has not enough remaining memory to handle Job 1 (1000Mo).
```

Créer une classe ***`NotEnoughMemoryException`*** *et modifier le code de la méthode* ***`void accept(Job job)`*** *de la classe* ***`Node`*** *afin qu'elle lève une exception en cas de mémoire insuffisante.*

7 Ordonnanceur : interface `Scheduler` (2 points)

Afin d'affecter des tâches aux nœuds, on va utiliser des ordonnanceurs (*scheduler*). Pour cela on va définir une interface. Cette interface sera générique (type paramétré) avec un seul paramètre de type que l'on nommera `J`. Le type `J` représentera un `Job` et vous devez donc ajouter dans la définition de l'interface le code nécessaire pour forcer le fait que `J` sera une extension de `Job`. L'interface n'aura qu'une seule méthode `List<J> scheduleJobs(List<J> jobs, List<Node> nodes)` qui essaye d'affecter les tâches `jobs` (en utilisant la méthode `accept` sur le nœud) aux nœuds `nodes` et qui renvoie la liste des tâches qu'elle n'a pas affectées (faute de disponibilité des ressources en fonction des tâches déjà affectées).
Créer l'interface ***`Scheduler`*** *avec la définition de méthode demandée.*

8 Ordonnanceur aléatoire : classe `RandomScheduler` (2 points)

Vous allez maintenant créer une classe `RandomScheduler` qui implémentera `Scheduler<Job>`.
Un `RandomScheduler` tente d'affecter les tâches au hasard sur les nœuds. Pour chacune des tâches à affecter, un nœud est pioché au hasard et on essaie d'y affecter la tâche. Si le nœud n'accepte pas la tâche, celle-ci est ajoutée à la liste des tâches non affectées. Pour tirer un nœud au hasard, vous pourrez utiliser une instance de `Random` avec la méthode `int nextInt(int bound)` qui renvoie un entier aléatoire entre `0` et `bound - 1` inclus.
Créer la classe `RandomScheduler` implémentant `Scheduler<Job>`.

9 Classe `PriorityJob` (2 points)

Dans certains cas, des tâches sont plus importantes que d'autres. Afin de prendre cela en compte, vous allez créer une classe `PriorityJob` qui étendra `Job` et qui implémentera `Comparable<PriorityJob>`. Cette classe rajoutera l'attribut, le constructeur et la méthode suivants :

- un attribut `int priority` représentant la priorité de la tâche. Plus une tâche est prioritaire et plus sa priorité est élevée.
- un constructeur `PriorityJob(int memory, int priority)` qui crée un `PriorityJob` avec la mémoire et la priorité spécifiées.
- une méthode `int compareTo(PriorityJob job)` qui renvoie :
  - un entier négatif si `this` a une priorité supérieure à celle de l'argument `job`,
  - `0` si `this` et `job` ont la même priorité,
  - un entier positif si `this` a une priorité inférieure à celle de l'argument `job`,

Créer la classe `PriorityJob` étendant `Job`.

10 Ordonnanceur avec priorité : classe `PriorityScheduler` (2 points)

Nous allez maintenant créer une classe `PriorityScheduler` qui implémentera `Scheduler<PriorityJob>`.
L'idée derrière un `PriorityScheduler` est de trier les taches par priorité. Pour trier, vous pouvez utiliser la méthode `static void sort(List<T> list)` de `Collections` qui trie une liste d'éléments pouvant se comparer à eux-même. Si vous avez implémenter correctement méthode `compareTo` de la classe `PriorityJob`, un appel à `sort` va trier la liste des tâches en mettant les tâches les plus prioritaire en premier.
Une fois les tâches triées, le `PriorityScheduler` va tenter d'affecter les tâches dans cet ordre. Pour cela, il va tenter de l'affecter à tous les nœuds. Dès qu'il trouve un nœud acceptant la tâche, le `PriorityScheduler` passe à la tâche suivante. Si jamais aucun nœud ne convient, la tâche est rajoutée à la liste des tâches non-affectées.
Créer la classe `PriorityScheduler` implémentant `Scheduler<Job>`.

11 Classe `Controller` (3 points)

Un contrôleur (*controller*) permet d'organiser la répartition des différentes tâches sur les nœuds de la ferme. Pour cela il faut préalablement ajouter les nœuds gérés et soumettre les tâches à prendre en compte. Comme `Scheduler`, la classe `controller` définira un type paramétré avec un paramètre de type `J` qui devra être une extension de `Job`. La classe `Controller` devra avoir les méthodes suivantes :

- `void addNode(Node node)` : ajoute un nœud.
- `Controller(String name, Scheduler<J> scheduler)` : construit un contrôleur avec un nom et un ordonnanceur.
- `void submitJob(J job)` : ajoute une tâche non-affectée au contrôleur. Cette méthode devra lever une `IllegalArgumentException` si aucun nœud du contrôleur n'a assez de capacité de mémoire pour gérer la tâche.
- `void scheduleJobs()` : affecte (en utilisant le `Scheduler` donné au constructeur) les tâches non-encore affectées. Toutes les tâches qui ne sont pas affectées par le `Scheduler` sont conservées dans les taches non affectées du contrôleur.
- `void printNodesAndNonScheduledJobs()` : affiche les nœuds et les tâches non affectées.

Créer la classe `Controller` avec les méthodes demandées et les attributs qui vous semblent nécessaires.