Cours MOOC EPFL d'introduction à la programmation orientée objet, illustré en Java

Deuxième devoir (noté)
Constructeurs

J. Sam & J.-C. Chappelier

du 5 novembre au 23 novembre 2015

Ce devoir comprend deux exercices à rendre.

1 Exercice 1 — Boîte de Petri

Le but de cet exercice est de simuler de façon très basique une « boîte de Petri » peuplée « d'organismes cellulaires ».

1.1 Description

Télécharger le programme fourni sur le site du cours [1] et le compléter.

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `SimulationPetri.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/`;

1. `https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/SimulationPetri.java`

3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `SimulationPetri.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Le code fourni crée une boîte de Petri et y ajoute un certain nombre d'organismes. Il affiche son contenu avant et après en avoir simulé l'« évolution ».

Les définitions des classes `Petri` (la boîte) et `Cellule` (les organismes) manquent et il vous est demandé de les fournir.

1.1.1 La classe `Cellule`

Une cellule est caractérisée par :
- un nom (`nom` de type `string`) ;
- une taille en microns (`taille` de type `double`) ;
- un niveau d'énergie (`energie` de type `int`) ;
- et une couleur (`couleur` de type `string`).

Respectez strictement les noms demandés pour ces attributs.

Les méthodes qui sont spécifiques à cette classe et font partie de son interface d'utilisation sont :
- des constructeurs conformes à la méthode `main` fourni, avec l'ordre suivant pour les paramètres lorsqu'il y en a : le nom, la taille, le niveau d'énergie et la couleur ;
  le constructeur par défaut initialisera le nom à « `Pyrobacculum` », la taille à 10, le niveau d'énergie à 5 et la couleur à « `verte` » ;
- une méthode `affiche` permettant d'afficher les caractéristiques de la cellule ; l'affichage respectera scrupuleusement le format suivant :
  `<nom>, taille = <taille> microns, énergie = <energie>, couleur =`

suivi d'un saut de ligne ;
`<nom>` est le nom de la cellule, `<taille>` sa taille, `<energie>` son niveau d'énergie et `<couleur` sa couleur ;

— une méthode `division` sans paramètre permettant à la cellule de se diviser ;
la méthode `division` retournera une nouvelle `Cellule` issue de la division ; cette dernière sera construite par copie de la cellule d'origine ; la copie subira ensuite une mutation de sa couleur ;
la division a un coût énergétique pour la cellule d'origine qui perd un point d'énergie après sa division.

Vous simulerez la mutation de façon très simple en modifiant la couleur (une seule fois) : de « `verte` » elle passera à « `bleue` », de « `bleue` » à « `rouge` », de « `rouge` » à « `rose bonbon` », de « `violet` » à « `verte` ». Pour toute autre couleur, la mutation ajoutera simplement « `fluo` » à cette couleur (pour simplifier, sans aucun autre traitement sur la chaîne de caractères). Par exemple si la couleur d'origine est « `jaune` », la mutation de la couleur donnera « `jaune fluo` ».

1.1.2 La classe `Petri`

Une boîte de Petri sera caractérisée par sa population : un tableau dynamique (`ArrayList`) de `Cellule`. Les méthodes de son interface d'utilisation sont :

— une méthode `ajouter` permettant d'ajouter à la population de la boîte une cellule passée en paramètre ; la bactérie sera toujours ajoutée en fin de tableau ;

— une méthode `affiche` affichant le contenu de la boîte ; cette méthode se contentera d'afficher toutes ses cellules selon le format d'affichage précédement décrit pour les cellules (et sans répétition de code) ; cette méthode parcourera l'ensemble de la première à la dernière cellule, dans l'ordre de l'ajout ;

— une méthode `evolue` permettant de faire évoluer la population de la boîte ; cette méthode ne retourne rien et est sans paramètres ;
faire évoluer le contenu de la boîte consiste à faire se diviser toutes les cellules qui y sont présentes à l'origine (avant l'appel à la méthode d'évolution) ;
les cellules issues de la division seront ajoutées à la boîte ; toujours en fin d'ensemble (mais ne seront pas elles-mêmes divisées) ;
une fois que toutes les cellules de la population *d'origine* se sont divisées, il faudra supprimer celles dont le niveau d'énergie est inférieur ou égal à zéro ;
pour préserver la cohérence du contenu de la boîte, vous ne supprimerez

des cellules qu'après avoir terminé toutes les divisions.

Indication Pour supprimer la valeur à la position `i` dans un `ArrayList v`, vous pouvez utiliser les instructions suivantes :

```
last = v.size()-1;
Collections.swap(v, i, last);
v.remove(last);
```

1.2 Exemples de déroulement

L'exemple de déroulement ci-dessous correspond au programme principal fourni.

```
Population avant évolution :
Pyrobacculum, taille = 10.0 microns, énergie = 5, couleur = verte
PiliIV, taille = 4.0 microns, énergie = 1, couleur = jaune
Population après évolution :
Pyrobacculum, taille = 10.0 microns, énergie = 4, couleur = verte
PiliIV, taille = 4.0 microns, énergie = 1, couleur = jaune fluo
Pyrobacculum, taille = 10.0 microns, énergie = 5, couleur = bleue
```

2 Exercice 2 — Plan d'étude

Le but de cet exercice est de programmer un certain nombre de fonctionnalités permettant à un étudiant d'établir son emploi du temps en choisissant des cours parmi ceux proposés par un plan d'étude.

Chaque cours offre des activités (séances de cours ex-cathedra ou d'exercices) qui ont un horaire.

Il s'agit essentiellement d'aider l'étudiant à choisir des cours du plan d'étude en garantissant qu'il n'y ait pas de conflit sur les horaires des activités.

2.1 Description

Télécharger le programme fourni sur le site du cours [2] et le compléter.

2. `https://d396qusza40orc.cloudfront.net/intropoojava/assignments-data/Planning.java`

ATTENTION : vous ne devez modifier ni le début ni la fin du programme, juste ajouter vos propres lignes à l'endroit indiqué. Il est donc primordial de respecter la procédure suivante (les points 1 et 3 concernent spécifiquement les utilisateurs d'Eclipse) :

1. désactiver le formatage automatique dans Eclipse :
   `Window > Preferences > Java > Editor > Save Actions`
   (et décocher l'option de reformatage si elle est cochée)
2. sauvegarder le fichier téléchargé sous le nom `Planning.java` (avec une majuscule, notamment). Si vous travaillez avec Eclipse vous ferez cette sauvegarde à l'emplacement `[dossierDuProjetPourCetExercice]/src/` ;
3. rafraîchir le projet Eclipse où est stocké le fichier (clic droit sur le projet > refresh) pour qu'il le prenne en compte ;
4. écrire le code à fournir entre ces deux commentaires :

```
/*****************************************
 * Completez le programme a partir d'ici.
 *****************************************/

/*****************************************
 * Ne rien modifier apres cette ligne.
 *****************************************/
```

5. sauvegarder et tester son programme pour être sûr(e) qu'il fonctionne correctement, par exemple avec les valeurs données plus bas ;
6. rendre le fichier modifié (toujours `Planning.java`) dans « OUTPUT submission » (et non pas dans « Additional ! »).

Le code fourni met à disposition une classe `Time` permettant de modéliser un horaire comme étant un jour (de type `String`) et une heure de début (de type `double`). Cette classe met de plus à disposition une fonction statique `printTime` qui permet d'afficher un `double` en tant qu'heure. Pour invoquer une méthode statique d'une classe en Java, on préfixe simplement la méthode du nom de la classe suivie d'un point (*e.g.* `Time.printTime(10.5)`, voir aussi `Time.print()`).

Notez que la partie à compléter comporte déjà les lignes d'affichage impliquant des accents. A vous bien sûr de les placer au bon endroit. Le but est simplement de fournir les caractères accentués (certains claviers pouvant être dépourvus de facilités à cet égard).

La définition des classes `Activity` (activité liée à un cours), `Course` (cours), `StudyPlan` (plan d'étude), et `Schedule` (emploi du temps) manquent et il vous est demandé de les fournir.

2.1.1 La classe `Activity`

Une activité, comme une séance d'exercices, est caractérisée par :
- le lieu où elle se déroule (par exemple un nom de salle, de type `String`) ;
- une durée en heures (un `double`) ;
- et un horaire (de type `Time`).

Aucun de ces attributs ne devra être modifiable après son initalisation.

Les méthodes qui sont spécifiques à cette classe et font partie de son interface d'utilisation sont :
- un constructeur conforme au `main` fourni, avec l'ordre suivant pour les paramètres : le lieu, le jour, l'heure et la durée ;
- les « getters » `getLocation`, `getTime` et `getDuration` retournant respectivement : le lieu de l'activité, son horaire et sa durée ;
- une méthode `conflicts` prenant en argument une activité ; cette méthode retourne `true` si l'activité courante est en conflit horaire avec l'activité passée en paramètre, et `false` sinon ;
  deux activités sont en conflit si elles ont lieu le même jour et que leurs plages horaires se chevauchent ; deux activités telles que l'heure de fin de l'une est identique à l'heure de début de l'autre ne sont pas en conflit ;
- une méthode `print` affichant les caractéristiques de l'activité en respectant scrupuleusement le format suivant :
  `le <horaire> en <lieu>, durée <duree>`
  où `<horaire>` est l'horaire de l'activité dans le format d'affichage produit par la méthode fournie `print()` de `Time`, `<lieu>` est le lieu de l'activité et `<duree>` sa durée affichée selon le format produit par la fonction fournie `printTime`.

2.1.2 La classe `Course`

Un cours a pour caractéristiques :
- un identificateur unique (de type `String`) ;
- un nom (de type `String`), comme `"Introduction à la programmation"` ;
- une activité correspondant au cours ex-cathedra hebdomadaire ;
- une activité correspondant à la séance d'exercices hebdomadaire ;
- et le nombre de crédits qui y est associé (un `int`).

Aucun de ces attributs ne devra être modifiable après son initalisation.

Les méthodes qui sont spécifiques à cette classe et font partie de son interface d'utilisation sont :
- un constructeur conforme au `main` fourni, avec l'ordre suivant pour les

paramètres : l'identificateur, le nom, l'activité liée à la séance ex-cathedra, l'activité liée à la séance d'exercices et le nombre de crédits ; le constructeur affichera le message « `Nouveau cours : <id>` », suivi d'un saut de ligne ; `id` est l'identificateur du cours ;
- les « getters » `getId`, `getTitle` et `getCredits` retournant respectivement l'identificateur unique du cours, son nom et son nombre de crédits ;
- une méthode `workload` retournant la charge de travail hebdomadaire à consacrer au cours : il s'agit de la somme de la durée de ses activités ;
- une méthode `conflicts` prenant en paramètre une `Activity` et retournant `true` si au moins une activité du cours est en conflit avec l'activité passée en paramètre ; et `false` sinon ;
- une méthode `conflicts` prenant en paramètre un `Course` ; cette méthode retourne `true` si au moins une activité du cours passé en paramètre est en conflit avec le cours courant ; et `false` sinon ;
- une méthode `print` affichant

les caractéristiques du cours respectant scrupuleusement le format suivant :

```
<id>: <nom> - cours <activite_excathedra>, exercices <activite_exercices>. crédits : <credits>
```

sans saut de ligne à la fin ; `<id>` est l'identificateur du cours, `<nom>` est son nom, `<activite_excathedra>` est l'activité associée à la séance ex-cathedra, affichée selon le format prévu par la méthode `print` de `Activity`, et `<activite_exercices>` est l'activité associée à la séance d'exercices, également affichée selon le format fourni par la méthode `print` de `Activity` et `<credits>` est le nombre de crédits du cours.

2.1.3 La classe `StudyPlan`

La classe `StudyPlan` modélise un plan d'études proposant un ensemble de cours.

Elle est caractérisée par un tableau dynamique (`ArrayList`) de `Course`. Les méthodes qui sont spécifiques à cette classe et font partie de son interface d'utilisation sont :

- un constructeur initialisant l'ensemble de cours à une tableau vide ;
- un constructeur de copie créant un `StudyPlan` à partir d'un autre `StudyPlan` ; sa liste de cours sera une copie de celle du `StudyPlan` passé en paramètre ;
- la méthode `addCourse` prenant en paramètre un `Course`, permettant d'ajouter un cours à l'ensemble des cours du plan d'étude ;
- une méthode `conflicts` prenant en paramètre un identificateur de cours

(un `String`) et une sélection de cours (sous la forme d'un `ArrayList` d'identificateurs de cours sous la forme de `String`) ;
cette méthode retournera `true` si le cours identifié par le premier paramètre est en conflit avec un cours de la sélection ; et `false` s'il n'y aucun conflit ;
la méthode `conflicts` retournera `false` si le cours identifié par le premier paramètre ne fait pas partie du plan d'étude (aucun des cours du plan d'étude n'a cet identificateur) ;
les cours de la sélection qui ne font pas partie du plan d'étude seront ignorés ;

— une méthode `print` prenant en paramètre un identificateur de cours et affichant le cours du plan d'étude ayant cet identificateur ;
le format d'affichage sera celui prévu par la méthode `print` de `Course` ;
s'il n'existe aucun cours dans le plan d'étude avec l'identificateur donné, la méthode `print` ne fera rien ;

— une méthode `credits` prenant en paramètre un identificateur de cours (`String` et retournant le nombre de crédits de ce cours s'il est dans le plan d'étude ; dans le cas contraire, la méthode retournera zéro ;

— une méthode `workload` prenant en paramètre un identificateur de cours et retournant la charge de travail hebdomadaire de ce cours s'il est dans le plan d'étude ; dans le cas contraire, la méthode retournera zéro ;

— une méthode `printCourseSuggestions` prenant en paramètre une sélection de cours (donnée sous la forme d'un `ArrayList` d'identificateurs de cours) ; cette méthode affichera l'ensemble des cours du plan d'étude sans conflit avec la sélection proposée ;
les cours compatibles seront affichés selon le format prévu par la méthode `print` de `Course` suivi d'un saut de ligne ; les cours de la sélection qui ne font pas partie du plan d'étude seront ignorés ; si aucun cours du plan d'étude n'est compatible avec la sélection de cours proposée, la méthode affichera le message :
`Aucun cours n'est compatible avec la sélection de cours.`

Pour prémunir votre code contre les failles d'encapsulation, votre classe `Course` ne devra fournir aucune méthode permettant d'en modifier les attributs.

Vous veillerez à bien modulariser les traitements. Une méthode permettant de trouver, parmi les cours du plan d'étude, celui ayant un identificateur donné sera certainement la bienvenue. Cette méthode ne fera pas partie de l'interface d'utilisation publique de la classe.

2.1.4 La classe `Schedule`

L'emploi du temps d'un étudiant est modélisé au moyen d'un objet de type `Schedule`. Cette classe est caractérisée par :

- l'ensemble des cours choisis par l'étudiant (un `ArrayList` d'identificateurs de cours) ;
- Le plan d'étude dans lequel il a choisi ses cours.

Les méthodes qui sont spécifiques à cette classe et font partie de son interface d'utilisation sont :

- un constructeur initialisant le plan d'étude au moyen d'une copie d'un `StudyPlan` passé en paramètre ;
- un constructeur de copie initialisant le plan d'étude au moyen d'une copie du plan d'étude du `Schedule` passé en paramètre ; et l'ensemble des identifiants de cours au moyen d'une copie de l'ensemble d'identifiants du `Schedule` passé en paramètre ;
- une méthode `addCourse` retournant un booléen et prenant en paramètre une identificateur de cours ; cet identificateur sera ajouté à l'ensemble des cours de l'emploi du temps s'il n'est pas en conflit avec les cours déjà choisis, auquel cas l'identificateur sera inséré en fin d'ensemble ; La méthode `add_course` retournera `true` si le cours a pu être ajouté et `false` sinon ;
- une méthode `computeDailyWorkload` retournant la charge de travail quotidienne moyenne associée aux cours de l'emploi du temps ; cette charge sera la somme des charges de travail associées aux cours de l'emploi du temps divisée par 5 (5 étant le nombre de jours travaillés) ;
- une méthode `computeTotalCredits` retournant le nombre de crédits rapportés par les cours de l'emploi du temps ; il s'agit de la somme des crédits de ces cours ;
- et une méthode `print` affichant l'emploi du temps selon le format suivant (voir aussi l'exemple de déroulement plus bas) :

  ```
  <cours_choisis>
  Total de crédits   : <credits>
  Charge journalière : <charge>
  Suggestions :
  <sugestions_cours>
  ```

  - `<cours_choisis>` est l'ensemble des cours de l'emploi du temps, chacun affiché selon le format prévu par la méthode `print` de la classe `StudyPlan` et suivi d'un saut de ligne ;
  - `<credits>` est le nombre de crédits de l'emploi du temps (tel que calculé par la méthode `computeTotalCredits` de la classe `Schedule` ;

— <charge> est la charge de travail associée à l'emploi du temps (telle que calculée par la méthode `computeDailyWorkload` de la classe `Schedule`;
— <suggestions_cours> est l'ensemble des cours qu'il est encore possible de choisir dans le plan d'étude, selon le format prévu par la méthode `printCourseSuggestions` de la classe `StudyPlan`.

Les classes `StudyPlan` et `Schedule` ne doivent pas comporter de méthodes publiques donnant accès aux tableaux qu'elles ont en attributs.

2.2 Exemples de déroulement

L'exemple de déroulement ci-dessous correspond au programme principal fourni. Le caractère '\' ne figure pas dans la sortie. Il n'y a pas de saut de ligne dans la sortie à la place de '\'.

```
L'activité physicsLecture a lieu le lundi à 09:15 en Central Hall, \
durée 01:45.
L'activité historyLecture a lieu le lundi à 10:15 en North Hall, \
durée 01:45.
physicsLecture est en conflit avec historyLecture.

Nouveau cours : PHY-101
Nouveau cours : HIS-101
Nouveau cours : ECN-214
Emploi du temps 1 :
ECN-214: Finance - cours le vendredi à 14:00 en South Hall, durée 02:00, \
exercices le vendredi à 16:00 en Central 105, durée 01:00. crédits : 3
Total de crédits   : 3
Charge journalière : 00:36
Suggestions :
PHY-101: Physique - cours le lundi à 09:15 en Central Hall, durée 01:45, \
exercices le lundi à 14:00 en Central 101, durée 02:00. crédits : 4
HIS-101: Histoire - cours le lundi à 10:15 en North Hall, durée 01:45, \
exercices le mardi à 09:00 en East 201, durée 02:00. crédits : 4

Emploi du temps 2 :
ECN-214: Finance - cours le vendredi à 14:00 en South Hall, durée 02:00, \
exercices le vendredi à 16:00 en Central 105, durée 01:00. crédits : 3
HIS-101: Histoire - cours le lundi à 10:15 en North Hall, durée 01:45, \
exercices le mardi à 09:00 en East 201, durée 02:00. crédits : 4
Total de crédits   : 7
Charge journalière : 01:21
```

```
Suggestions :
Aucun cours n'est compatible avec la sélection de cours.

Emploi du temps 3 :
PHY-101: Physique - cours le lundi à 09:15 en Central Hall, durée 01:45, \
exercices le lundi à 14:00 en Central 101, durée 02:00. crédits : 4
Total de crédits   : 4
Charge journalière : 00:45
Suggestions :
ECN-214: Finance - cours le vendredi à 14:00 en South Hall, durée 02:00, \
exercices le vendredi à 16:00 en Central 105, durée 01:00. crédits : 3
```