TP 01
Notions de bases
Prise en main des outils

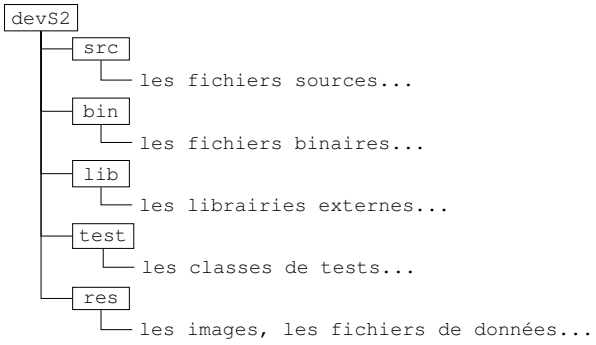
Exercice 1 : Prise en main de l'environnement de développement

Le but de cet exercice est de prendre en main votre outil de développement. Nous conseillons fortement *VSCodium* et de compiler dans le terminal intégré durant au moins les 4 premières semaines.

Q1. Commençons par mettre en place de l'environnement de travail.

Q1.1. Lancez votre outil de développement et définissez un répertoire `devS2` pour votre projet java. Tous les développements futurs devront être dans ce même répertoire.

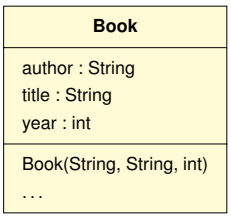
Q1.2. Créez les répertoires nécessaires pour que votre projet respecte l'arborescence suivante :



Q1.3. Ajoutez un répertoire `tp01` et placez-y votre terminal.

Q2. Écrivons votre première classe.

Q2.1. Créez une classe `Book` respectant les spécifications UML suivante :



On y retrouve donc 3 attributs et un constructeur complètement spécifié (dont la signature a autant de paramètres que l'objet n'a d'attributs.)

Q2.2. Ajoutez une méthode `toString` transformant une instance en une chaîne de caractères au format CSV, à savoir : `<author>;<title>;<year>`.

```
public String toString()
```

Q3. Il est temps de tester votre première classe.

Q3.1. Créez une classe `UseBook`, munie d'un programme principal `main` qui créé une instance de `Book` et l'affiche. On rappelle la signature incontournable d'une méthode `main` :

```
public static void main(String[] args)
```

Q3.2. Compilez ce programme au moyen de la commande `javac Book.java`. Normalement aucune erreur de compilation ne doit être observée.

Q3.3. Exécutez ce programme via la commande `java Book`.

Q4. Approfondissons un peu cette classe.

Q4.1. Écrivez au sein d'une classe `UseBook2` un programme principal `main` créant une bibliothèque de 5 livres (sous forme d'un tableau défini en extension).

Q4.2. Complétez votre `main` pour qu'il affiche toute la bibliothèque (à raison d'un livre par ligne).

Q4.3. Ajoutez à ce programme principal la recherche et l'affichage du livre le plus ancien de la bibliothèque.

Q4.4. En objet, il n'est pas convenable que du code d'une classe adresse directement les attributs d'une autre classe (`UseBook2` pour les attributs de `Book` en l'occurrence). Réécrivez votre code en le basant sur une méthode `isOldest` à ajouter à votre classe `Book`.)

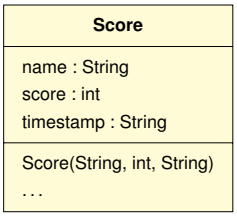
```
boolean isOldest(Book other)
```

Exercice 2 : Sauvegarde des scores.

On veut créer un système de gestion des scores, qui ne garde en mémoire qu'un nombre limité de scores.

Q1. Intéressons-nous d'abord à un score.

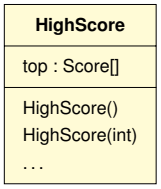
Q1.1. Créez une classe `Score`, répondant aux spécifications UML suivantes :



Q1.2. Ajoutez une méthode `toString` qui retourne un score en chaîne de caractères ; respectant la forme : `(<timestamp>) <name> = <score>`.

Q2. Concentrons-nous sur le panthéon des meilleurs scores.

Q2.1. Écrivez une classe `HighScore` permettant de sauvegarder différents scores à l'aide d'un tableau. La taille du tableau par défaut (si non spécifiée en paramètre du constructeur) est 100. Votre classe doit respecter la structure suivante :



Q2.2. Écrivez une méthode `getNbFreeSlot` retournant le nombre d'emplacement inoccupé dans le tableau des scores.

```
int getNbFreeSlot()
```

Q2.3. Ajoutez une méthode `toString` qui met en forme textuelle l'objet à raison d'un score par ligne. Seuls les emplacements occupés sont affichés. Voici un exemple d'un `HighScore` de taille par défaut, comportant 2 scores :

```
TOP SCORE:
(29/01) Alice = 300
(29/01) Bob = 800
**98 free slots**
```

Q3. Travaillons maintenant à une première méthode d'ajout de scores.

Q3.1. Écrivez une méthode `addFirstFreeSlot` qui ajoute un score donné à la première place disponible (en commençant à remplir par le début du tableau). La méthode retourne `true` si l'ajout est réalisé, `false` si le tableau est plein et qu'un ajout est impossible.

```
boolean addFirstFreeSlot(Score newScore)
```

Q3.2. Créez une classe `UseHighScore` munie d'une méthode principale qui joue le scénario suivant :

— création d'un `HighScore` de taille par défaut
— ajout de Alice avec un score de 300 le 29/01
— ajout de Bob avec un score de 800 le 29/01

— affiche le pantheon des scores.

Q3.3. Créez une classe `UseHighScore2` munie d'une unique méthode `main` qui reçoit la taille du `HighScore` le scénario suivant :

— création d'un `HighScore` de taille fournit en argument du `main`
— ajout de Alice avec un score de 300 le 29/01
— ajout de Bob avec un score de 800 le 29/01
— ajout de Alice avec un score de 42 le 30/01
— ajout de Alice avec un score de 650 le 31/01

Effectuez différents tests en faisant varier la taille du `HighScore`. Les scores en surnombre seront simplement perdus. Notez que la transformation d'une chaîne de caractères en entier se fait via l'instruction suivante :

```
int param = Integer.parseInt("10");
```

Q4. Concevons une seconde méthode d'ajout des scores. On souhaite conserver les scores du plus important au plus faible. Si un nouveau score plus important que certains de ceux présents est ajouté et qu'il n'y a plus de place disponible, le plus faible disparaît. Peu importe le joueur qui réalise les scores. S'il reste des emplacements disponibles, on insère le nouveau score de manière à conserver le tableau trié en décalant les scores plus faibles.

Q4.1. Ajoutez à votre classe `HighScore` une méthode `shifting` qui décale les cases à partir d'un indice donné en paramètre.

```
void shifting(int idx)
```

Q4.2. Complétez votre classe par la méthode d'ajout trié `addHighestFirst`.

```
boolean addHighestFirst(Score newScore)
```

Q4.3. Écrivez une classe `UseHighScore3` dotée uniquement d'une méthode principale qui joue le scénario suivant :

— création d'un `HighScore` de taille 3
— ajout de Alice avec un score de 300 le 29/01
— ajout de Bob avec un score de 800 le 29/01
— ajout de Carl avec un score de 750 le 30/01
— ajout de Alice avec un score de 650 le 31/01
— ajout de Bob avec un score de 430 le 31/01

Vérifiez le bon déroulement du scénario en affichant l'état du `HighScore` entre chaque étape. Votre programme doit mener à l'affichage :

```
TOP SCORE:
(29/01) Bob = 800
(30/01) Carl = 750
(31/01) Alice = 650
**0 free slots**
```

Q5. Concevons maintenant une dernière méthode d'ajout de scores. On souhaite désormais ne conserver que le meilleur score de chaque joueur. Si le joueur n'est pas dans le `HighScore`, il est ajouté dans le premier emplacement libre. S'il est déjà présent, on compare son ancien score et le nouveau : son score n'est remplacé que si le nouveau est plus élevé.

Q5.1. Ajoutez à la classe `Score` une méthode `isSamePlayer` qui détermine si un score donné provient du même joueur.

```
boolean isSamePlayer(Score other)
```

Q5.2. Ajoutez à la classe `Score` une méthode `isHigher` qui détermine si un score est plus élevé qu'un autre passé en paramètre.

```
boolean isHigher(Score other)
```

Q5.3. Ajoutez à votre classe `HighScore` la méthode `addOneSlotPerPlayer` qui ajoute le nouveau score si le joueur n'est pas déjà présent et qu'il reste un emplacement libre, ou si le joueur est présent et que son nouveau score est plus élevé que l'ancien.

`boolean addOneSlotPerPlayer(Score newScore)`

Q5.4. Créez une classe `UseHighScore4` munie d'une unique méthode `main` qui joue le scénario suivant :

—

— création d'un `HighScore` de taille 3
— ajout de Alice avec un score de 300 le 29/01
— ajout de Bob avec un score de 800 le 29/01
— ajout de Alice avec un score de 650 le 31/01
— ajout de Carl avec un score de 750 le 30/01
— ajout de Bob avec un score de 430 le 31/01

Vérifiez le bon déroulement du scénario en affichant l'état du `HighScore` entre chaque étape. Alice ne doit jamais obtenir deux slots. Votre programme doit mener à l'affichage :

```
TOP SCORE:
(31/01) Alice = 650
(29/01) Bob = 800
(30/01) Carl = 750
**0 free slots**
```