POLYTECH DIJON

TRAVAUX DIRIGÉS

Développement d'applications Web

JAVA – Classes & Multithreading

OBJECTIFS

Le but de ce TD est de manipuler la syntaxe Java permettant de créer des classes et de commencer à toucher au multithreading

Notions abordées :

- Séquences d'échappement
- Classe & Interface
- Multithreading

> **Note**
> *Vos enseignants sont là pour répondre à vos questions. Alors posez des questions ! :-)*

> **Attention**
> *Les séquences d'échappement utilisées dans ce sujet nécessitent un système Linux, MacOS ou Windows 10+.*

I. CONSOLE

I.A. Affichage de base

Lors du TD précédent, vous avez vu comment afficher de l'information dans la console à l'aide de `System.out.println`. Commençons par voir s'il vous reste quelque chose en mémoire…

- Créez un nouveau projet à l'aide de votre IDE.
- Ecrivez un programme qui affiche le message « Chargement en cours… ».
- Testez le bon fonctionnement.

Le code suivant permet de faire une pause de 2 secondes dans le programme :

```
Thread.sleep(2000);
```

- Intégrez la séquence suivante à votre programme :
  1. Afficher le message « Chargement en cours… »
  2. Attendre 3 secondes
  3. Afficher le message « Chargement terminé. »
- Testez le bon fonctionnement.

Ok, c'est pas mal, mais pendant le chargement, l'utilisateur ne sait pas si le programme fait quelque chose ou si tout est planté. Pour ne pas laisser de place au doute, vous allez animer l'affichage des pointillés pour indiquer à l'utilisateur que tout va bien.

- Modifiez le programme précédent pour afficher la séquence suivante :
  1. « . »
  2. « .. »
  3. « … »
  4. « . »
  5. …

  Chaque étape de la séquence sera espacée de 500ms et la séquence se terminera au bout de 3s.
- Testez le bon fonctionnement.

Bien, ça bouge, l'utilisateur comprend que le programme fait le job mais niveau affichage, c'est un peu déroutant tous ces points. L'idéal serait de toujours écrire les pointillés au même emplacement dans la console. L'animation informerait bien l'utilisateur et ne remplira pas l'écran de points si le chargement dure longtemps.

I.B. Séquences d'échappement

Il existe des caractères spéciaux, appelés séquences d'échappement, qui, au lieu d'afficher une lettre ou un chiffre, permettent de manipuler le comportement de la console.

Vous en connaissez normalement quelques-uns :

- Que fait le caractère spécial `\t` ?
- Quelles différences y a-t-il entre `\r` et `\n` ?

Il existe un grand nombre de séquences d'échappement. Par exemple, `\033[2J` efface la console et `\033[5;3H` place le curseur de la console à la quatrième colonne de la sixième ligne.

- Modifiez le programme précédent pour que la séquence de pointillés s'affiche toujours à la fin du message « Chargement en cours ».
- Testez le bon fonctionnement.

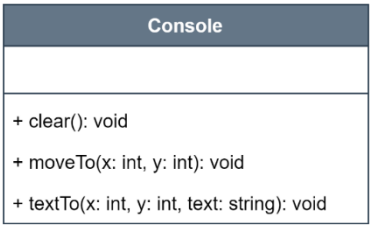
> **Attention**
> *C'est évident mais… Si vous n'écrivez qu'un caractère, un seul caractère sera affiché dans la console, et donc un seul caractère sera remplacé si vous écrivez à un endroit qui contient déjà du texte.*

I.C. Une classe pour les gouverner tous !

Les séquences d'échappement sont extrêmement pratiques dans le cas présent. Cependant, à chaque fois que l'on a besoin de de modifier la position du curseur, il faut se souvenir des bonnes valeurs à utiliser.

Nous allons donc créer une classe qui va se charger de contrôler la console pour nous. De cette façon, lorsque nous aurons besoin de contrôler la position du curseur dans d'autres projets, nous n'aurons qu'à utiliser ce module sans avoir à nous replonger dans la documentation des séquences d'échappement.

➢ Créez la classe suivante :



- La méthode **`clear`** efface le contenu de la console.
- La méthode **`moveTo`** positionne le curseur de la console aux coordonnées fournies.
- La méthode **`textTo`** affiche le texte fourni en paramètre aux coordonnées indiquées.

> **Astuce**
> *La fonction* **`printf`** *que vous avez déjà vue en C permet de créer des séquences d'échappement dynamiques intégrant des variables).*

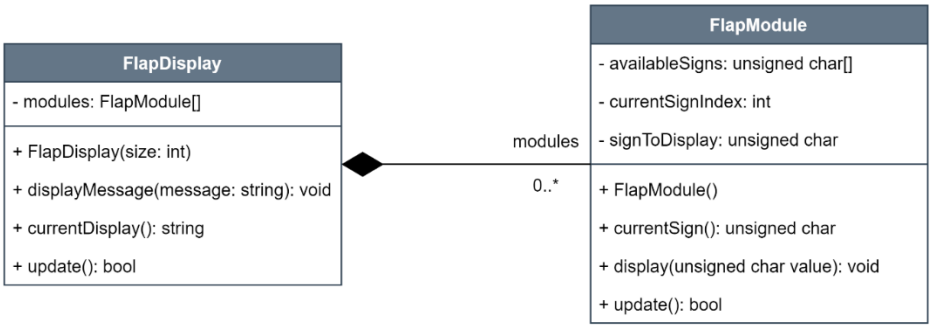
➢ Testez le bon fonctionnement de votre classe avec un programme qui teste le comportement des trois méthodes.

II. AFFICHEUR A PALETTES

Si nous vous parlons d'affichage dans les gares ou les aéroports, de rotation de lettres, et du cliquetis sous-jacent, il est possible que certains d'entre vous n'aient pas la référence. Dans ce cas, pas de souci, voici de quoi nous parlons : https://www.oatfoundry.com/blog/riche-histoire-dafficheurs-a-palettes-planche-solari/

Le principe est simple : un panneau d'affichage est constitué de modules adjacents. Chaque module peut afficher une lettre parmi un ensemble disponible à un instant T. Pour passer à une autre lettre, le module effectue une rotation, découvrant une à une les lettres de son ensemble jusqu'à afficher le caractère souhaité.

La modélisation ci-dessous va nous permettre de reproduire cette mécanique dans la console :



II.A. FlapModule

➢ Créez la classe **`FlapModule`** à partir des informations suivantes :

Description des attributs

- **`availableSigns`** sera un **`ArrayList<Character>`** contenant tous les symboles pouvant être affichés par le module.
- **`currentSignIndex`** indique l'indice du symbole actuellement affiché parmi les éléments de **`availableSigns`**.
- **`signToDisplay`** est le caractère que l'on souhaite voir afficher sur le module.

Description des méthodes

- **`FlapModule`** est le constructeur par défaut de la classe. Il définira la liste des symboles disponibles : les lettres de l'alphabets en majuscule + le caractère **`\u2591`**. **`signToDisplay`** sera initialisé avec le caractère **`\u2591`** et **`currentSignIndex`** avec l'indice de ce caractère.
- **`currentSign`** retournera le symbole actuellement affiché
- **`display`** définit le caractère que l'on souhaite voir apparaître sur le module. Si le caractère demandé n'est pas présent dans **`availableSigns`**, le caractère **`\u2591`** sera à afficher.
- **`update`** affichera le caractère suivant si le caractère affiché n'est pas le caractère souhaité. Si une modification du caractère affiché est faite, update retourne **`true`**, sinon elle retournera **`false`**.

➢ Dans votre **`main`**, créez une instance de **`FlapModule`** et testez son fonctionnement en affichant le caractère courant du module sur une position fixe de la console (la classe Console précédemment réalisée pourra sûrement vous être utile). La rotation des symboles doit se faire toutes les 100ms.

➢ Pourquoi utilisons nous un **`ArryList<Character>`** plutôt qu'un **`ArrayList<char>`** ?

II.B. FlapDisplay

➢ Créez la classe **`FlapDisplay`** à partir de la modélisation précédente et des informations suivantes :

Description des attributs

- **`modules`** stockera les modules constituant le panneau d'affichage. On utilisera un **`ArrayList<FlapModule>`** ici.

Description des méthodes

- `FlapDisplay` est le constructeur de la classe. Il prend en paramètre la taille de l'afficheur, c'est-à-dire le nombre de modules placés côte à côte.
- `displayMessage` permettra de définir le message à afficher sur l'afficheur.
- `currentDisplay` retournera une chaine de caractères représentant l'état actuel de chacun des mocules.
- `update` demandera une mise à jour des modules. Si tous les modules sont à jour, `update` renverra `false`. Si au moins un module a changé d'état, elle renverra `true`.

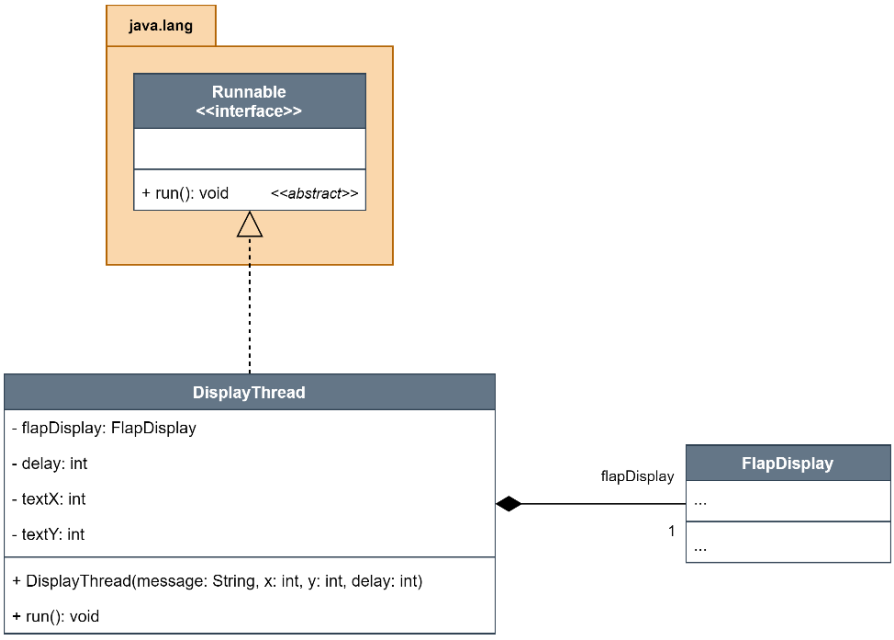
➢ Dans votre `main`, créez une instance de `FlapDisplay` et tester l'affichage de plusieurs messages successifs.

II.C. Multi-affichages

Nous allons à présent mettre en place plusieurs afficheurs. Mais chacun aura sa propre vitesse de rafraichissement. Pour cela, nous allons utiliser le multithreading. Chaque afficheur sera géré par son propre thread. Vous n'avez pas encore eu le cours sur le sujet ? Comme si vous lisiez vos cours…

En JAVA, le multithreading est simple à mettre en place.

➢ Créez la classe DisplayThread ci-dessous :



> **Note**
> *L'interface `Runnable` est fournie par JAVA. Inutile donc de la recoder.*

- La classe `DisplayThread` implémente l'interface `Runnable` (notez la modélisation UML). La syntaxe pour implémenter une interface est la suivante :

```
public class MaClasse implements MonInterface
{
    ...
}
```

- La classe **`DisplayThread`** doit redéfinir la méthode run de l'interface **`Runnable`**. Pour cela, vous devez ajouter le décorateur **`@Override`** avant le nom de la fonction à redéfinir :

```
@Override
public void MethodeARedefinir()
{
    ...
}
```

- Le constructeur de **`DisplayThread`** initialisera son instance de **`FlapDisplay`** avec le message fourni en paramètre.
- La méthode **`run`** de **`DisplayThread`** affichera le texte généré par **`flapDisplay`** aux coordonnées **`(textX ; textY)`** et mettra à jour, toutes les **`delay`** millisecondes, l'afficheur jusqu'à ce que ce dernier ne change plus.

➢ Dans votre **`main`**, démarrez plus thread d'affichage avec la syntaxe suivante :

```
Thread monThread = new Thread(new RunnableClass());
monThread.start();
```

Astuce
Il est possible que votre programme s'arrête avant que les afficheurs aient fini de générer le texte à afficher. Il est possible de forcer le programme principal à attendre la fin de l'exécution des threads en cours :

```
CountDownLatch latch = new CountDownLatch(nombre_threads);

//Démarrage des threads
…

try {
    latch.await();
}
catch(Exception e) { }
```

*En passant **`Latch`** à vos threads, ces derniers pourront décrémenter le compteur lorsqu'ils terminent leur tâche.*