TP Noté - Testing

Dans ce TP nous allons mettre à profil nos connaissances pour réaliser le test unitaire d’un système dit de « memory store ».

Le rendu se fait sur clef USB, dans un zip contenant le code source complet, sous forme « NOM.Prenom.zip ».

# But

Nous cherchons à créer un « memory store » sous Java. Un « memory store » est une zone de stockage de variables, ayant (ou non), une durée de validité dans le temps. Pour des raisons de simplicité, on commencera par s’occuper de la gestion clef-valeur sans se soucier d’une quelconque gestion de la validité dans le temps.

# Memory Store (4pts)

Avant de réaliser les tests unitaires, il convient de créer le système qui va réaliser le memory store.

Java prodigue un élément qui nous aidera grandement pour réaliser ce système : l’objet HashMap.

1. Créer un package « com.memory »
2. Créer une classe « MemoryStore »

Cette classe est notre memory store, avec les couples clef/valeur contenus dans le HashMap. Pour vous aider, nous prodiguons ici un patron simple à reproduire :

|  |
| --- |
| /\*\* \* The memory store will keep data inside the class, for a specific amout of time\*/**public** **class** MemoryStore { **private** **static** HashMap<String, Object> *mem*; /\*\* \* Add an object into the memory store \* **@param** obj The object to store \* **@return** The key stored \*/ **public** **static** String add(Object obj){} /\*\* \* Get a specific object regarding the key \* **@param** key The key to search \* **@return** The object retrieve, or null if nothing found \*/ **public** **static** Object get(String key){} /\*\* \* Delete an entry from the memory store \* **@param** key The key to delete \* **@return** The delete value result (true if the object has been found and deleted, false in other case) \*/ **public** **static** **boolean** delete(String key){} /\*\* \* Empty the memory store \*/ **public** **static** **void** erase(){} /\*\* \* This generate a random string of the given size \* **@param** length The output length to retrieve \* **@return** The generated string \*/ **private** **static** String generate(**int** length){ String chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890"; **int** charLength = chars.length(); String pass = ""; **for**(**int** x=0; x<length; ++x){ **int** i = (**int**) Math.*floor*(Math.*random*() \* charLength); pass += chars.charAt(i); } **return** pass; }} |

Voici quelques explications :

* Add : ajoute une entrée dans le hashMap, et renvoi la clef générée (utiliser generate pour créer cette clef).
* Get : renvoie l’objet correspondant à la clef fournie par add.
* Delete : supprime une clef stockée dans le HashMap à partir d’une clef donnée.
* Erase : supprime toutes les clefs (on obtient une HaspMap vide).
* Generate : le code complet est donné, génère une chaine de caractères de longueur X, aléatoire.

On vous demande, dans un premier temps, de coder l’intérieur de chaque fonction. En faisant attention aux « effets de bords ». Par exemple, lorsque l’on souhaite ajouter un objet au HashMap avec Add, la clef prévue ne doit pas déjà exister dans la liste des clefs existantes (dans le HashMap).

**Indice :**

* L’objet HashMap contient déjà la majorité des fonctions nécessaires pour réaliser une bonne partie de ces fonctions.

# Tests (10pts)

On vous demande de créer un package « test.memory » qui sera utilisé pour réaliser les tests unitaires.

Une fois ce package fait, vous créerez les tests unitaires. Pensez, entre chaque fonction de tests, à effacer le memory store.

Vous êtes libre de choisir les tests à faire, mais vous devez, en haut de chaque classe de tests, expliquer le choix des tests faits, et leurs implications. De même, vous devez expliquer pourquoi certains tests ne sont pas faits, et quelle est l’incidence sur la fiabilité du système (sans incidence, faible, critique…).

# Gestion de la validité (6pts)

On cherche à ajouter une fonctionnalité nécessaire à ce type système : un store temporaire. Ce type de système, au moment de l’ajout (fonction Add donc), créer un timer qui se lancera dans X secondes (spécifié par l’utilisateur), à cet instant, le timer appellera la fonction delete de notre memory store, avec la clef nécessaire pour la destruction. Ainsi, notre couple clef/valeur disparait au bout d’un certain temps, ce qui est fort pratique, par exemple, lorsqu’il s’agit de sessions (pour rappel une session à une durée de validité limitée dans le temps).

**Indice :**

1. Vous devez disposer d’une classe qui « extend TimerTask », et qui dispose d’un constructeur ayant en paramètre la clef à détruire.
2. Utiliser « new Timer. schedule(new TimerTask(), duration) ; » pour démarrer un timer, TimerTask étant une nouvelle instance de notre classe (cf indice 1).
3. Créer une fonction Add qui surcharge l’actuelle Add : elle dispose de deux arguments ; l’objet (comme actuellement), et la durée en millisecondes avant destruction. Cette fonction utilise alors (cf indice 2) pour déclencher la suppression.
4. Utilisez « Thread.sleep(milliseconds) » pour faire une pose à un endroit donné dans un programme.

On vous demande alors de réaliser un test unitaire ayant la possibilité de tester la suppression de la clef, avec un timeout (duration) fixé à 200ms.