Formation : INFO2

Rapport du TD n°04
Java Avancé

Guillaume CAU
18/10/2019

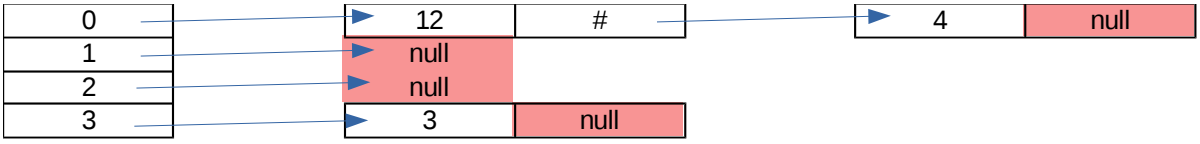
UNIVERSITÉ PARIS-EST
MARNE-LA-VALLÉE

Exercice 01 :

Question 01 :

Pour créer notre HashMap, nous avons besoin de « maillons ». Ces maillons contiendront l'information que l'on souhaite stocker dans notre HashMap.

Dans notre cas, nous voulons stocker des **int**. Le maillon *Entry* contiendra donc nécessairement un champ de type **int**.
Le maillon devra également stocker la référence vers le prochain maillon.
En effet, notre HashMap suivra la structure suivante :



Il est donc nécessaire de stocker la référence vers l'objet suivant pour construire la liste chaînée.

La classe *Entry* doit être **package-private**. En effet, les maillons ne sont utilisés que par la HashMap et non depuis l'extérieur.

L'accessibilité de ses champs doit être **private**. En effet, il n'est pas censé être possible de récupérer les valeur depuis l'extérieur de celui-ci. Il faut passer par les méthodes internes.

Question 02 :

Dans cette question, il n'est pas efficace d'utiliser l'opérateur **%** car le processeur effectue des divisions successives pour le calculer.
Il existe une méthode pour le remplacer qui peut être calculée en un cycle processeur. Il s'agit de l'opérateur logique **&**.
Grâce à celui-ci, nous allons être capable de récupérer la valeur des derniers *bits* de l'int ce qui correspondra au « multiple ».

Dans notre cas, il suffit de faire l'opération suivante : (int)&0x1

Si l'on souhaite utiliser 8 cases, on utilise l'opération suivante : (int)&0x7.

Plus général, il suffit de faire le **et logique** sur l'int avec la valeur souhaitée-1.

0	1	1	1	0	0	1	0	
					1	1	1	&
0	0	0	0	0	0	1	0	

Le masque dans l'exemple ci-dessous est 7. Pour obtenir une suite de 1, il faut soustraire 1 à la valeur souhaitée. De cette manière, nous pouvons récupérer la valeur du masque dynamiquement.

Question 03 :

Pour réussir à créer la méthode **add**, j'ai utilisé ce qu'on appelle une fonction de hashage. Une fonction de hashage est une fonction qui prend une entrée et qui renvoie une sortie de taille finie at avec les valeurs les plus différentes possible à la sortie.
La fonction de hashage utilisée ici est celle vue dans la question 2.

Question 04 :

La langage Java met à disposition des utilisateurs la possibilité de créer des classes internes. Il s'agit de classe qui sont définie à l'intérieur d'autres classes.

Dans notre cas, pour placer la classe en interne dans notre HashMap, nous devons la mettre en private puis en static.

Question 05 :

En Java, il existe une méthode qui est très utilisée. Il s'agit de la méthode ForEach. Cette méthode prend un ensemble d'élément en entrée puis exécute un certain code pour chaque élément.

En Java, il est également possible d'utiliser des méthodes Lambda, nous les avons vu dans les TPs précédents.
Java nous met à disposition l'interface fonctionnelle Consumer dans notre cas. Celle-ci prend une entréer en ne renvoie rien. Exactement le comportement recherché pour notre **forEach**.

Question 06 :

Pour créer la méthode contains, il faut parcourir toute la HashMap pour voir si l'élément n'est pas déjà inséré.

Si l'élément est déjà présent, il faut renvoyer **true**, sinon **false**.

Exercice 02 :

Question 01 :

Le java met à disposition des utilisateurs ce que l'on appelle des types paramétrés. Il s'agit en quelque-sorte de variable de type. Il est possible de dire : cette fonction renverra un objet de type **K**. Et ce type **K** peut prendre plusieurs valeurs comme : String/Integer/Map…

Dans notre cas, nous souhaitons créer un tableau d'objets contenant un type paramétré. Or, Java oublie les types paramétrés après la compilation. La machine virtuelle n'a donc pas moyens de savoir de quelles types sont les éléments du tableau. C'est donc le type Objet qui est mis par défaut.
Il est donc nécessaire de faire un **cast** avant de l'affecter à un champ ou une variable.

Question 02 :

Dans la méthode **contains**, le type le plus générique est Object.

Question 03 :

Pour implémenter la fonction d’augmentation de la taille du tableau, il faut copier le tableau précédent dans un tableau plus grand.
Pour améliorer les performances et éviter les collisions, il a été choisis d’augmenter la taille du tableau par 2 à chaque fois que celui-ci est à moitié remplis.
Le fait d’augmenter la taille de 2 en 2 permet de limiter l’appel à la fonction d’agrandissement plus il y a d’élément dans le programme.

Question 04 :