**From:** TERRIEN Olivier ItecFccOsd   
**Sent:** Monday 22 August 2016 19:03  
**To:** YOUNSI Thin-Hinane ItecFccOsd; SABRI Abdessamad ItecFccOsd  
**Subject:** Pour vous aider dans vos prochains tests techniques

Voice une liste non exhaustive de ce que vous pourriez trouver en entretien technique à votre niveau (à mon sens).

Après si vous avez des questions, n’hésitez pas.

**Principe orienté Objet :**

Souvenez-vous, il y a 3 principes :

1. Héritage
2. Polymorphisme
3. Encapsulation

Le polymorphisme c’est, grosso modo, ce qui permet de définir plusieurs méthodes avec le même nom avec ou pas des signatures différentes. Ce n’est donc pas QUE de l’héritage (même si l’héritage induit forcément le polymorphisme). Quand ce n’est pas pour l’héritage, on peut très bien trouver dans une même classe deux méthodes « addition » : int addition(int, int) et float addition(float, float). Selon le type des paramètres qu’on passera, on n’appellera pas la même méthode. Et c’est le compilateur qui le saura.

L’héritage, c’est ce qui permet de spécialiser une structure en ajoutant des comportements ou en redéfinissant des comportements existants (par polymorphisme donc).

L’encapsulation, c’est ce qui permet de rassembler en un même endroit, les éléments de structure d’une classe ainsi que son comportement et en en exposant que son « interface » (ça ne signifie pas qu’on a créé une interface). Le cas d’école c’est l’attribut qu’on définit private et dont on expose ou pas les méthodes GET et SET qu’on met donc public.

**Interface / Abstract**

Un interface c’est une structure qui ne contient que des méthodes abstraites (ou default depuis java8) mais aucun attribut d’instance (par contre, on avoir des attributs de classes comme des constantes). Depuis Java 8, pour les besoins des expressions Lambda, on peut avoir également dans les interfaces des méthodes **default** qui possèdent donc des implémentations. Ces méthodes sont bien évidemment surchargeables.

Une classe abstraite est une classe qui contient (ou pas) des méthodes abstraites. C'est-à-dire que les méthodes n’ont pas encore de corps. On ne sait pas encore ce qu’elles vont faire.

La seule différence entre une classe abstraite et une interface, depuis Java8, c’est donc que les interfaces ne peuvent pas avoir d’attribut d’instance.

On peut instancier une interface ou classe abstraite (on parler donc de classe anonyme) à condition de fournir une implémentation aux méthodes abstraites

On peut implémenter plusieurs interfaces. Mais on ne peut hériter que d’une et une seule classe (qu’elle soit abstraite ou non d’ailleurs).

**Niveau de visibilité**

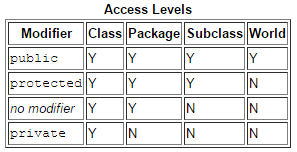
**Public** : tout le monde a accès à l’élément (classe, champs, méthodes)

**Protected** : C’est le plus compliqué (c’est pourquoi on pose souvent la question sur lui ). Il y a en fait 2 niveaux de visibilité dans le niveau « protected »

1. Le niveau lié à l’héritage : tous les enfants voient les élements protected des la chaine « parente ».
2. Le niveau « package » : les éléments qui ne sont pas enfant MAIS qui sont dans le même package, voient quand même les éléments « protected »

**Default** (c'est-à-dire qu’on ne met rien) : il s’agit d’une visibilité dite « package ». C’est à dire que seuls les éléments d’un même package peuvent les voir

**Private** : visiblement uniquement dans la class courante



**Final**

Final se met sur des classes, des attributs, des variables, des paramètres de méthode ou mêmes des méthodes  et **il a 2 sens différents selon où il est placé (ce qui est bien pourri)**

* Method : quand final est défini sur une méthode, cela signifie que la méthode ne pourra pas être surchargé dans les classes filles.
* Class : quand final est défini sur une classe, cela signifie que la classe ne pourra pas être dérivée (une classe finale signifie donc par défaut que TOUTES ses méthodes sont finales)
* Attribut, paramètres et variables (le résultat est le même) : quand final est placé devant un paramètre de méthode ou devant une variable ou devant un attribut, cela signifie que la première valeur affectée ne changera pas (*rien à voir avec l’héritage n’est-ce pas ? je vous avais dis que c’était pourri*)

**Static**

Le mot clef static signifie « de classe » qui est le contraire de « d’instance »

Il peut se positionner devant une méthode ou un attribut et a exactement la même signification :

* Une élement static sera disponible POUR TOUTES les instances d’une classe QUELLE QUE SOIT l’instance.

Pour accéder à un élément static on n’a donc PAS besoin d’instancier un objet. D’ailleurs, un élement static ne peut donc PAS être surchargé par une classe fille

**Static Final**

Un élement « static final » signifie donc qu’il est disponible sans avoir à instancier un objet ET que sa valeur ne changera pas quoiqu’il arrive.  il s’agit donc d’une constante. Dans ce cas, au moment de la compilation, si c’est simple, Java va remplacer l’appel de la constante par sa valeur.

**Synchronized**

Ce mot clef est lié à la capacité multithread de Java. Pour rappel un thread est un bout de programme qui peut s’exécuter en parallèle d’un autre (de manière asynchrone) et ce dans la même JVM.

On ne peut pas avoir plus de threads à un instant T que de cœurs (des cœurs logiques pour être plus précis) sur un processeur. **Seul l’OS gère le moment où un thread s’exécute** et peut parfois recourir à des arrêts de thread pour en lancer d’autres.

Le premier thread qui exécutera une methode **synchronized** posera un verrou sur l’instance « this ». tous les autres threads qui chercheront à exécuter la même méthode ou une autre methode synchronized du même objet, seront mis en attente. Une fois qu’il aura fini d’exécuter la méthode, le verrou sera relâché et un autre thread pourra donc se mettre à travailler.

En gros, c’est pour mettre des goulots d’étranglement sur les thread quand, par exemple, les threads utilisent une même variable qui peut être modifiée par toutes.

Il est possible d’utiliser synchronized comme s’il s’agissait d’une méthode mais ce n’en est pas un :

*synchronized(my\_lock) {*

*}*

C’est exactement la même chose que précédemment sauf que là on dit quelle est l’instance qu’on veut utiliser comme verrou (ce n’est pas forcément this, mais ça peut)

**Garbage collector :**

Le garbage collector est le principe introduit par java pour éviter qu’on ait à coder la gestion mémoire des objets (ce qui n’est pas le cas en C++ et c’est pourquoi tous les codeurs C++ n’ont plus de cheveux sur la tête… enfin c’est aussi parce qu’ils sont vieux).

Souvenez-vous, il y a deux gros emplacements mémoires dans un programme :

* La pile (=stack)
* Le tas (=heap)

Comme son nom l’indique, la stack sert à empiler (et dépiler donc) les appels de méthodes, les références et les types primitifs

Le heap c’est l’endroit où vivent les objets  **c’est donc lui que le garbage collector va nettoyer (et uniquement lui)**

* **Le Garbage Collector est comme une femme : on propose et c’est lui qui dispose. En gros, on ne peut pas le contrôler. On peut juste lui demander de nettoyer la mémoire, mais après il fait ce qu’il veut et quand il veut.**

Le passage des paramètres de type object dans une méthode se fait toujours par « copie de référence » (si c’est un type primitif, c’est uniquement par valeur puisque les primitifs ne vivent pas sur le heap).

Une référence c’est en gros la variable de la pile qui sait où trouver l’objet sur le heap (en gros c’est un integer qui contient l’adresse mémoire de l’objet stocké sur le heap)

**Un objet sera donc éligible à la « garbage collection » dès lors que plus aucune référence ne pointera sur lui. Je dis bien « éligible à la garbage collection » et non pas « supprimé de la mémoire ». La suppression de la mémoire est un processus complexe qui est géré uniquement par le garbage collector. Eligible signifie donc uniquement que le garbage collector sait qu’il peut supprimer l’objet de la mémoire mais il va peut être le faire plus tard**

**Générique**

Les génériques sont identifiés par les caractères ‘<’ et ‘>’. Par exemple : public class List**<T>**

Ca signifie que va dire au compilateur quel type d’élément on veut utiliser à la place (si on mot List<T extends Person> on specifie juste qu’on va restreindre T à la hiérarchie de Person dont Person elle-même). Ca évitera donc de se poser la question du type de l’élément au moment d’exécuter le programme (ce qui est coûteux).

Par exemple : List<Person> a = new ArrayList<Person>()  ça signifie qu’on définit une liste qui va accepter que des objects de type Person ET tous les enfants de Person (puisqu’un enfant de Person est aussi une Person).

On peut définir ses propres classes ou méthodes génériques.

**Enum**

Une enum est une classe particulière qui fournit des éléments typés fortement. C’est très utile pour gérer des éléments constants. En plus, il y a une gestion mémoire particulière et optimisée pour les énums.

**Introspection / Reflection**

C’est un moyen de lire la définition d’une classe ou d’un objet au moment de l’exécution du programme.

Les annotations son souvent utilisées par reflection.

Il faut utiliser ce moyen avec **parcimonie**.

**Sérialization**

La sérialisation c’est la capacité qu’a un objet à sortir de la JVM (par exemple pour transiter sur le réseau dans le cadre de RMI par exemple). Un objet sérialisable pourra être stockée sur un disque ou envoyé sur le réseau ET pourra ainsi être reconstruit à l’identique.

Parfois, il ne sera pas nécessaire de reconstuire tout l’objet parce que certains champs découlent des autres ou parce que le stockage de tel ou tel champs est trop volumineux.

Déclarer un attribut « transient » permet d’éviter la sérialization de cet attribut.

**WebService VS RMI**

RMI signifie « Remote Method Invocation ». C’est un protocol de communication entre 2 JVM (uniquement java donc). En gros, avec RMI, une JVM est capable d’appeler la méthode d’une autre JVM (en envoyant donc des objets sérializables). RMI est vieux et avec les avancés dans les webservices, il est carrément obsolète.

WebService utilise le protocol http. C’est parfait pour **l’interopérabilité** (par exemple, un serveur Java et un client C#).

Regards,

**Olivier TERRIEN**

Technical Leader

ITEC/FCC/OSD - (Antalis, SLA, B3S, CBS, TDA)

Tel: +33 (0)1 589 **84 018**