- attention aux termes.
- réponses peu précises.

satisfaisant - car la partie présentation application ratée.

Compte Rendu du TD numéro 1 de l'équipe Eirb'reteau

Pour le 23 septembre

Coordinateur : Victor Dury
Tandem 1 : Reda Boudjeltia, Lionel Adotevi
Tandem 2 : Aurélien Nizet, Pierre Gaulon
TD du 19 septembre 2014

Ce TD étant une séance de prise en main des divers outils de développement, nous l'avons parcouru entièrement pendant la séance. Cependant, il nous restait la partie supplémentaire à démarrer, c'est à dire la présentation de l'application que nous aurons à développer pendant les séances suivantes. Pour ces travaux, nous avons décidé de fonctionner en deux tandems, comme conseillé, mais chaque personne de l'équipe a parcouru l'ensemble des étapes de ce TD, puisqu'il s'agit de documentation.

1 Synthèse sur le travail de l'équipe

Cette partie résume les différentes découvertes de notre groupe pendant ce travail de recherche et de documentation sur la prise en main de l'environnement et du langage Java.

⇒ la dernière version du langage ?

1.1 Le kit de Développement de Java

1.1.1 La compilation

Le suffixe du fichier généré par le compilateur après compilation d'un fichier *.java* est *.class*.

1.1.2 L'exécution

Si l'on essaye d'exécuter la commande *java* avec comme argument ExecutionExemple.class, l'exécuteur va essayer de trouver la classe nommée *class* dans le répertoire ExecutionExemple, d'où l'erreur : impossible de trouver ou charger la classe principale ExecutionExemple.class.

la machine virtuelle.

pourquoi une classe qui s'apelle class.

1.1.3 Vérifier la version de java

Pour vérifier la version de la JVM (Java Virtual Machine) utilisée, il faut écrire sur le terminal la commande *java – version*.
Pour vérifier la version du compilateur utilisée, il faut écrire sur le terminal la commande *javac – version*.
Il faut récupérer la distribution JRE (Java Runtime Environment) lorsque l'on doit exécuter une application Java sur sa machine. Certaines commandes Java ne se trouvent pas dans la distribution JRE, telles que : *java*, *javac*, *javadoc*, car elles appartiennent à Java Developpement Kit, et permettent d'exécuter, de compiler ou de créer la documentation sur du code Java.

développer des applications java.

faux

1.2 Boutez Vos Neurones

1.2.1 La classe String

La classe *String* appartient à la hiérarchie de paquetages java.lang.
Pour obtenir le nombre de caractères d'une chaîne x, on peut utiliser la méthode *length*() sur l'objet x, comme ceci : $x.length$. Pour obtenir une sous-chaîne de la chaine x, on peut utiliser la méthode *substring*() sur x. Par exemple, $x.length(3, 5)$ va nous donner la sous-chaîne de x de son indice 3 à son indice 5. Les fonctions *hashCode*() ou *toString* sont des méthodes de *String* héritées de la classe *Object*.

L'objectif de l'interface Comparable est de définir un ordre total entre deux objets de même classe. Comparable<String> va donc définir un ordre total entre deux objets de classe *String*. Le lien "est-un" entre cette interface et la classe *String* indique que les méthodes de l'interface Comparable sont héritées à la classe *String*. Ainsi, on peut comparer deux chaînes. La méthode de Comparable, nommée *compareTo*(), a pour prototype *intcompareTo(String)*. Cependant, l'usage de chevrons indique que cette méthode peut être utilisée par d'autres objets que ceux de classe *String*. C'est ce que l'on appelle la surcharge de méthode.

syntaxique mécanisme de substitution de type à la compilation

faux

faux

Il existe une autre classe représentant des chaînes que *String*, nommée *StringBuffer*. Son nom complet est *java.lang.StringBuffer*, et elle permet de représenter des chaînes non constantes. La question se pose donc : pourquoi aurions-nous besoin d'avoir des encapsulations différentes pour les chaînes constantes et non constantes ?

- Tout d'abord, le partage d'objets. Il est impossible de partager une chaîne constants et une chaîne non constante, puisque la référence de cette dernière peut changer en fonction de la mémoire occupée par celle-ci (particulièrement si le début de la chaîne est modifié). Il est donc nécessaire d'avoir deux formes différentes pour ces chaînes.
- Ensuite, la gestion de l'espace. En effet, dès la déclaration d'une chaîne constante, on connaît sa taille et donc on peut allouer directement de la mémoire. Alors que pour une chaîne non constante, on va devoir allouer, ou libérer de la mémoire à chaque fois que la chaîne est modifiée. Ces deux traitements devant être différents, cela pousse à avoir deux classes différentes.
- Enfin, la modification des chaînes. Il est évident qu'afin de pouvoir modifier une chaîne, elle doit être non constante.

ok

chaîne constante car ne sera pas modifié ⇒ partage de valeur.

Il n'existe pas de lien "est-un" avec l'interface Comparable car il est impossible de comparer un objet constant et un objet non constant. L'instruction qui permet d'obtenir une chaîne non constante à partir d'une chaîne constante est :

StringBuffer(*String str*). L'instruction inverse est *String*(*StringBuffer str*).

1.2.2 La source de *ExecutionExemple.java*

Le mot-clef *final* permet de rendre une variable constante. Cela entraîne une variable constante si cette variable est d'un type de base (int, boolean ...), et une référence constante si la variable est d'un autre type. L'objet alors référencé peut quant à lui être modifié, tant que sa référence ne change pas.

Dans le fichier *ExecutionExemple.java*, les conventions de codages utilisées sont les suivantes :

- **chaîne** représente une variable (de classe String), car elle est écrite en minuscules.
- **EXCLAMATION** représente une variable (de classe String) constante (utilisation du mot clef *final*), car elle est écrite en majuscules.
- **System** représente une classe, car elle commence par une majuscule.
- **out** représente une variable de classe System, car elle est écrite en minuscules.
- **println()** représente une méthode, car il y a des parenthèses.

Dans l'expression **System.out.println(EXCLAMATION)**, l'envoi du message println se fait de la classe String vers la classe java.io.PrintStream. Les définitions de l'identificateur println sont différenciées par le nombre et le type des paramètres. Selon le type de paramètre utilisé dans cette méthode, le compilateur va décider si il doit utiliser telle ou telle autre définition de la méthode println(). Le traitement qui effectue l'opérateur + sur les instances de cette classe est une concaténation. Elle construit une variable de classe Stringbuffer, puis insère la première variable, puis la deuxième à la suite (append), et enfin la convertit en variable de classe String grâce à *String*(*StringBufferstr*).

1.2.3 Les chaînes de caractères littérales

Le code qui respecte la spécification sur les chaînes littérales est le 2ème, car la fonction de comparaison renvoie faux lorsque l'on compare deux chaînes identiques d'adresses différentes, et renvoie vrai lorsque l'on utilise le *intern*. Le premier code est une comparaison de références, superficielle, car on compare deux éléments partageant la même adresse. Il faut espérer que l'enchaînement des opérations générées par le compilateur ne soit pas celui-ci, car il y aurait alors un problème de mémoire. En effet, on utilise 3 instances pour une seule variable.

1.2.4 La source libre de String.java

Dans les commentaires de la source libre, on a les valeurs de retour des méthodes, en plus de leur type, et une description plus complète des paramètres attendus dans ces méthodes, par rapport à la documentation de la classe. Ces commentaires décrivent de manière explicite le fonctionnement de la fonction. Une chaîne de caractères est stockée un tableau de caractères, l'attribut value. En plus de cela cette chaîne stocke un attribut offset, un attribut count (le nombre de caractères), et un attribut hash.

La méthode *equals*() vérifie si les références sont les mêmes, et sinon vérifie en profondeur les deux tableaux de caractères. Le comparateur == vérifie

seulement les références. Le mot clef *native* marque une méthode qui va être implémentée en d'autres langages, par exemple du C ou C++.

1.3 Un coup d'oeil sur le monde java

Les concepteurs du langage Java sont Patrick Naughton et James Gosling. Le nom interne donné à Java SE 8 est *Spider*. Le nom donné à la JVM est *HotSpot*. Il faut télécharger Java EE (Enterprise Edition) pour les entreprises, Java FX pour les interfaces graphiques, et Java ME (Micro Edition) pour les Applications Embarquées. La machine virtuelle utilisée pour le développement des applications Java sur les téléphones android est *KVM* (Kilobyte Virtual Machine), développée par la société *OVA*.

1.4 Présentation de l'application

1.4.1 Le version de départ

Le problème avec la sortie d'un passager, c'est que l'application ne libère pas sa place dans le tableau de l'autobus contenant tous les passagers, mais la place du dernier passager arrivé, qui lui n'est pas forcément sorti. En effet, chaque passager qui entre dans l'autobus est inscrit dans son tableau de passagers (sa référence est notée dans la case *nombre_assis* + *nombre_debout* − 1). Si, par exemple, il y a 10 passagers (donc des indices 0 à 9 dans le tableau), et que le passager référencé à la case 4 (le 5ème passager) décide de descendre à un arrêt, et qu'un nouveau passager monte, le tableau va remplir la case numéro 8 (le 9ème passager) avec l'adresse du nouveau. On perd alors l'information sur l'ancien 9ème passager, alors qu'on garde l'information sur le 5ème passager, qui lui n'est plus dans l'autobus. Notamment, lors du prochain arrêt, on va demander au passager qui est sorti si il veut sortir à cet arrêt, et on ne va rien demander à l'ancien 9ème passager, qui est alors dans une dimension parallèle.

Une solution serait de mettre en place un compteur de passagers rencontrés dans la structure d'autobus. Ce compteur serait alors indépendant du nombre de passagers dans le bus, mais dépendant du nombre de passagers créés dans l'application. Chaque passager aurait donc un identifiant unique, l'état du compteur lors de sa création. Le premier problème est que ce compteur ne pourra pas dépasser *MAX* (initialement à 100), donc on ne poura pas créer plus de 100 passagers. Le deuxième problème sera lors de l'annonce d'un nouvel arrêt. L'autobus demandera à tous les passagers, même ceux déjà sortis, si ils veulent sortir (fonction *ps.std_nouvel_arret*), ce qui n'est pas très efficace. Une autre solution serait de créer 4 tableaux supplémentaires, 2 de taille max_assis et 2 de taille max_debout. Il s'agirait pour chaque catégorie (assis ou debout) d'un masque des places disponibles, défini par un tableau de booléens annonçant si telle place est disponible ou non, et le tableau d'adresses de passagers correspondant, référençant son occupant. Ainsi, lors de la sortie d'un passager, on passe le masque de sa place (assise ou debout, au même indice que le passager) à 0, cette place devient disponible pour un autre. On pourra y écrire par la suite. Plusieurs problème sont alors soulevés. Il s'agit d'une refonte de la structure d'autobus, puisque le nombre de passagers assis et debout deviennent obsolètes, et que l'on doit ajouter 2 tableaux supplémentaires (les masques), et diviser le tableau de passagers en deux (assis et debout). Cela utilise donc un espace mémoire

supplémentaire conséquent. De plus, l'ajout d'un passager, et la recherche de place disponible se fera alors en temps linéaire, et non plus constant.

1.4.2 Le portage en langage Java

La classe JaugeNaturel va servir à représenter le nombre de places assises et debout du transport disponibles. En C, cela réprésente le nombre de places disponibles et de places occupées (*assis*, *debout*, *max_assis* et *max_debout*), ainsi que les fonctions de demande de disponibilité *_atb_a_place_assise* et *_atb_a_place_debout*.

La classe EtatPassager va servir à réprésenter l'état d'un passager (assis, debout, ou dehors). En C, cela représente les deux booléens *assis* et *debout*, ainsi que les fonctions de demande d'état *ps_std_est_dehors*, *ps_std_est_assis* et *ps_std_est_debout*, et les fonctions internes de changement d'état *_ps_std_accepter_sortie*, *_ps_std_accepter_assise* et *_ps_std_accepter_debout*.

2 Commentaires

2.1 Commentaire de Pierre

Étant donné que cette séance était particulière, et concernait principalement de la documentation, mon commentaire en sera d'autant plus bref. Même si on a déjà touché du doigt ce sujet l'an dernier avec l'abstraction, et l'encapsulation, la notion de programmation orientée objet (et le langage Java) est nouvelle pour moi. C'est pourquoi je pense que cette séance a été une bonne chose pour se mettre dans le bain. Maintenant, je pense qu'il est plus important de pratiquer pour assimiler cette notion et apprendre à l'utiliser, d'autant plus qu'on part de bases connues, d'un code en C.

2.2 Commentaire de Lionel

Ayant fait du java durant mon stage cet été, ce projet n'a fait que confirmer mes acquis, et dans certains cas, faire la lumière sur certains points obscurs. Ce projet m'a notamment permis d'apprendre des choses que j'avais négligé tel que les Stringbuffers, la compilation ou l'execution, avec des outils comme javac et java que je ne connaissais pas car je travaillais sous NetBeans.

2.3 Commentaire d'Aurélien

Ce premier projet m'a permis de découvrir le java de manière globale, notamment la familiarisation avec la documentation. Il m'a également permis de revoir le vocabulaire déjà appris en cours.

2.4 Commentaire de Victor

Ce projet a été instructif de deux manières pour ma part. Tout d'abord, entrer dans le monde du java a été très intéressant. J'ai compris que c'est un langage qui va me permettre de faire beaucoup de choses une fois maîtrisé, même s'il reste beaucoup à apprendre. Je n'en avais jamais fait auparavant, et le fait d'être coordinateur m'a demandé de m'intéresser de tout ; je suis donc content de partir sur de bonnes bases. Secondement, en tant que coordinateur, j'ai pu apercevoir les qualités et les défauts de mes coéquipiers. Malgrès quelques manques de concentration de temps en temps, j'ai pu observer que tous étaient de bons programmeurs.

2.5 Commentaire de Reda

Je ne connaissais pas le langage java avant cette année, et ce premier projet m'a permis de confirmer ce que j'avais vu en cours. J'ai bien aimer le fait de travailler en équipe.