Programmation Fonctionnelle

ITIG ou VSC. ITIG on peut demander auprès de l’université la version payante. INTILIG. Faire attention à la version de JAVA, il faut qu’elle soit stable. Il nous encourage à prendre la dernière version du 17.1.8.

Jenv qui permet de switcher la version de java. Il faut l’installer. Il nous faut également gradle et un terminal. Activer aussi la sauvegarde automatique des IDE.

# Qu’est-ce que c’est ?

La programmation fonctionnelle est une façon de programmer différente de ce qu’on a pu voir.

**Paradigme impératif** a été fait par java, pascal. En C, on a eu un paradigme impératif auquel on a ajouté les pointeurs.

**Le paradigme objet,** on est dans un paradigme où on a voulu représenter les objets. L’aspect objet en java nous fera penser le monde en terme objet.

**Paradigme de la programmation à base de contrainte** : Prolog. Paul est le père de Jean et Jean le père de Pierre. Donc Paul est le grand-père de Pierre.

**Paradigme fonctionne**l c’est pensé un programme par ce qu’on assemble. Depuis JAVA 8, le JAVA a intégré des éléments fonctionnels. La programmation fonctionnel est plus déclarative, ce qu’on veut que comment le faire.

**Les particularités de la fonctionnel :**

* Les fonctions sont totales, déterministes, absence de réflexion, d’exception.
  + ***Qu’est-ce qu’est la totalité*** ? Elle est totale lorsqu’elle est définie sur ce qu’elle prend en entrée. Il n’y a pas de valeur pour lequel le prog plante. On a une valeur qui est toujours retourner. On verra à la fin du cours le type Option qui rend ce qui n’est pas totale, totale.
  + ***Qu’est ce que déterministe***? Si on fourni les mêmes entrées on obtiendra les mêmes sorties. C’est vrai en ASQUEL. Dans JAVA ce n’est pas un langage qui n’est pas totalement fonctionnelle. On peut étendre le terme déterministe. Pour comprendre ce qui est déterministe, on prend le contraire. Si je fais une entrée sortie, je vous demande de taper deux fois un caractère, on aura alors pas forcément la même sortie pour les deux.
  + ***Qu’estce que pure***? La fonction prend ses entrées et fourni un résultat, elle n’est pas censée modifier autre chose. Mettre un point à l’étudiant, je modifie la mémoire, c’est un effet de bord. A partir du moment que je peux modifier la mémoire, ca va poser pb lorsqu’elle est collective.
  + ***Pas de réflexion***
  + ***Pas d’exception.***

Les avantages :

La programmation fonctionnelle est fortement typée. Tout l’idée de la programmation fonctionnelle c’est que les problèmes soient visibles devant le programmeur. On peut faire des accès à la mémoire sans que cela pose pb.

Une certaine rigueur. Il y a moins de base en fonctionnel qu’en impératif.

# Génériques

Héritage, implantation, transitage(cast), la surcharge(overload) et le masquage (@override = redéfinir une méthode sur un fils et de l’appeler avec le fils), énumération (peut implémenter une interface, c’est une classe dont les itérations sont limités), sous-classe, super-classe, classe mère, classe fille, hiérarchie d’héritage.

List<Integer> xs = new ArrayList<>();

Xs.add();

Int s ;

For (int I = 0; i<xs.size(); i++){ /for(Integer v: xs)

S+=xs.get(i); / s+=v;

}

JEP Café José Paumard, il prend un concept et il va nous parler de ce concept pendant 15 min en JAVA. JAVA8 in action Mario Reda. Effective JAVA joshua Blok

Boxing in boxing : on le reverra dans java Stream lorsqu’on aura des listes d’entier. Classe StreamInteger.

Class Nom<T1,T2,…,Tn>{}

Il faut que les types aient quelque chose en commun dans les génériques pour simplifier.

R=0 ;

For (A f : fs){

R+= f.size() ;

}

Return r

List<A> notes = new ArrayList<>(); //Java est un langage typé et va deviner que c’est le type A.

On peut aussi écrire new ArrayList() ; C’est du Raw type, java l’accepte mais va nous alerter. Déconseiller.

Si **private,** l’instance de la classe. **Protected** les classes filles**. Public** c’est tout le monde. **Rien**

**Static :** il y a un espace mémoire qui est commun à toutes les instances de la classe. Si je mets static devant une méthode alors elle devient commune à toutes les instances de la classe. Les paramètres de généricité parlent au niveau de l’instance.

**Final**: on ne peut pas réaffecter. Une fois qu’on l’a initialisé on ne peut pas changer la valeur. Au sens héritage, pour une méthode cela nous interdit de la override.

## Typage Borne :

<T extends Number>, On ne peut pas mettre trois classes. Classe en premier puis l’interface <T extends C & I1 & I2>.

Si T extends Comparable cad que ce sont des éléments qui peuvent être triés. En java les collections sont forcément homogènes, il n’y a pas de collection hétérogène. Le joker permet de définir n’importe qu’elle classe dans le type générique. Cela permet de fixer le fait d’être en dessous de quelque chose ? extends MyClasse. A l’inverse : ? super MyClasse. C’est lié à la collection. Quand la collection doit contenir des objets de type T elle va appeler les méthodes de type T.

List<B> y=… ;

List <A> x = y ; // Cela ne marche pas, sauf List<A> x= new ArrayList<B>() ;

La convariance c’est dans la même direction vers le haut. A contra-variance c’est l’inverse. L’invariance c’est java qui ne sait pas faire.

## Type erasure : effacement type.

Problème de pont, on a dupliqué une méthode que le compilateur a créé.

Semaine prochaine du 1er oct QCM en programmation fonctionnelle.

# Les interfaces fonctionnelles

Une interface ne définit qu’un contrat contrairement à une classe abstraite. C’est un agrément entre quelque chose et la chose qui respecte le contrat. Une interface à un problématique de typage. On ne met pas de code dans un interface avant JAVA9.

Une classe peut avoir le comportement d’un étudiant et celui d’une étudiant boursier.

JAVA 9 et 8 ont ramené la possibilité de définir une méthode par default.

A <I> :

M()-> B

Default N()-> B{n ! m()}

Possibilité d’avoir du code avec default dans une interface.

* Qu’est-ce qu’une interface fonctionnelle ?

C’est une interface qui peut avoir des définitions par default. Quelque chose qui a qu’une méthode à définir en dessous. On peut mettre autant de méthode qu’on veut dans une interface. Une méthode pas encore définie.

* Pourquoi c’est 1 ? Cette idée de se dire que j’ai un truc dans java, ce truc est associé à la notion d’interface fonctionnelle et à la méthode qui manque.

Comment on utilisait avant cette modification les interface ?

@FunctionalInterface On veut que le compilateur nous avertisse si on s’est planté.

Il faut que z soit static pour faire : A v = B :: z ; Cela permet d’éviter de créer un objet B et de pouvoir l’utiliser directement avec l’instant. On peut faire si pas static (new B()) ::z

Crochet dans une interface graphique est un morceau de code.

# Lambda Calculs/ Lambda fonction

En mathématiques vous connaissez les fonctions : f :N->N x->x+1. G : N\*N->N x,y->x\*y

On va ajouter la lambda pour pouvoir faire la même chose.

Attention : (x,y)->x\*y est différent de x->y->x\*y => f(3,4) et g(3)(4)

LambdaParameters : Pour définir un lambda on définit les paramètres, on liste les paramètres. En java le + n’est pas défini que sur les entiers, il est défini sur Number aussi. On peut lister un ensemble de variable en donnant les types ou pas, on fait alors confiance au compilateur.

LambdaBody : En java le switch est une expression, un valeur que l’on peut prendre ne paramètre, que l’on peut retourner.

Exemple 1 :

? v = x -> {int y =x\*z ; int z=y\*3 ; return z ;}

Exemple 2 :

@FunctionInterface interface A{ int foo(int x); }

A v = z-> (z\*3)+99

Il a défini une interface fonctionnelle et il a utilisé un Lambda. Si on a une méthode qui a le type de retour de la Lambda alors on peut faire : return z -> (z\*3)+99

Attention au return, il faut que le type de retour de la Lambda soit compatible avec le type de retour. Il faut que ce qui se trouve à droite de la flèche soit compatible avec l’interface fonctionnel.

Return Null c’est un Object qui est retourné. Si on ne met rien pour foo c’est un retour de rien.

# Fonctions

## Dictionnaire de fonction présent dans l’API JAVA

Il manque une méthode dans les interfaces suivantes car ce sont des interfaces fonctionnelles.

Ce sera toujours : IF v= lambda

Le lambda va permettre de définir cette méthode qui manque.

Package java.util.function

Si une fonction prend T et ne renvoie rien alors elle fait un effet de bord, elle modifie la mémoire, ou alors elle réalise un affichage. Consumer

()->T : Cela sert à générer des valeurs. Supplier. On a un moyen d’obtenir des valeurs à la queue Leleu.

Le prédicat : T->B Cela sert à dire si la valeur est vraie.

Bipredicate : e,d -> e.note>= d.note

## Exemples

Supplier<Integer> foo = ()-> 4;

Integer v = foo.get();

DRY : don’t repeat yourself

Dans le consommateur, celle qui reste à définir est accept.

Quand on fait le compose, on prend l’entrée, on passe à la fonction en paramètre et ca nous fait notre résultat.

Compose : Function<T,R>\* Function< ? super V, ? extends T> -> Function<V,R>

On applique le résultat de la fonction sur la fonction passée en paramètre dans andThen.

AndThen : Function<T,R>\* Function< ? super R, ? extends V> -> Function<T,V>

Si j’utilise equals entre objet de type E ou Set<E>ou Map<E,…> alors j’ajoute equal et hashcode dans R.

Fold(xs,magique) => (N,+,O)

Exercice 4 :

Q1 :

* Version avec lambda :

JEP Café : présente les erreurs.

# Collections & Tables & Flux

Il faut distinguer les listes (collection) et les tables (map). On a deux hiérarchies et on va retrouver la notion d’itération.

Il faut choisir la bonne collection, c’est se donner moins de travail. Exemple : liste et ensemble.

On retrouve l’interface iterable et iterator. Lorsqu’on fait list<T> ts =… et for (T t : ts){} Ce qui est caché là-dedans est un itérateur. Le vrai code java qui est compilé est Iterator<T> it = ts.iterator() ; while(it.hasNext()){ T t = it.next() ;}

On va pouvoir passer des fonctions dans ces itérateurs.

C<T> c.forEach( ???) L’indice change à chaque tour de boucle, la donnée courante. Il faudrait alors un algo de l’élément courant. On lui passe un truc qui prend T et qui va faire quelque chose avec le T. T->(), c’est un consommateur Consumer<T>. C’est en réalité d’un Consumer< ? super T> et non U super T car c’est pas autorisé.

Donc on peut avoir une collection et faire c.foreach(t->…) et i.forEachRemaining(t->…)

Le foreach utilise un itérateur sur une structure itérable. On parcourt l’intégralité de la collection. Dans le foreachRemaining c’est ce qu’il reste et cela utilise un itérateur qui a pu déjà travailler.

Dans un hashMap pour faire Map<E,List<Double>) notes ; Il faut définir dans Etudiant hashcode et équals.

Listof(note) ; notes.get(bob).add(note) ; cela peut ne pas marcher car il peut ne pas y avoir de liste de notes.

If(notes.hasKey(bob)) {

Notes.put(bob,new ArrayList<>());

} notes.get(bob).add(note);

List<Double> notes\_bob = notes.computeIfAbsent(bob, k-> new ArrayList<>()) ;

Notes\_bob.get(bob).add(note);

Soit ma map avait bob et j’ai rien fait sinon elle n’avait pas bob et j’ai fait new ArrayList<>().

Notes.computeIfAbsent(bob, k->new ArrayList<>()).add(note);

# Stream

C’est système dynamique qui est construit au-dessus d’une collection au sens large. Particularité des Stream c’est le côté paresseux.

On appelle ça l’opération paresseuse. Range(1…) appliquer(x->x+1) donner(4). Ne marche pas lorsqu’on veut faire la moyenne. Les streams sont paresseux mais cela peut s’annuler. Lazy.

Eager pour l’inverse de lazy.

Opération création : ?-> S<T>

Opération intermédiaire : S<T>->S<U>

Opération terminale : S<T>-> U

List<E> es= ….

Es.stream() la liste est dans stream maintenant.

Cochez le bon bout de code ou alors expliquer ce que fait le code. Bien regarder les documents du prof.