

La généricité paramétrique dans Java 1.5

Notes de cours et TP 2007- IUP GMI 3 et Master d'Informatique UMINM 202

Marianne Huchard

1 Préliminaires pour le TP

Pour vérifier la version de Java avec laquelle vous travaillez, utilisez la commande `java -version`

Dans l'environnement actuel, positionnez la variable `PATH` de manière à ce qu'elle vous oriente vers le JDK 1.5, avec (en bash)

```
export PATH=/usr/local/java/jdk1.5.0/bin :$PATH
```

Pensez à positionner également la variable `CLASSPATH` et à utiliser le mécanisme des packages pour être dans de bonnes conditions de travail.

Pour avoir de nombreux détails sur la compilation, utilisez le compilateur `javac` avec l'option `-Xlint`.

2 Introduction

Le polymorphisme paramétrique (ou généricité), qui autorise la définition d'algorithmes et de types complexes (classes, interfaces) paramétrés par des types, est une caractéristique fondamentale des langages auxquels il apporte une meilleure lisibilité et un meilleur niveau d'abstraction. La plupart des langages de programmation évolués en disposent (Eiffel, Ada, C++, Haskell, etc.) ainsi que le langage de modélisation UML. La figure 1 présente ainsi un concept d'association ou de paire, souvent présent dans les bibliothèques proposant des structures de données (des collections). Une paire représente un couple d'éléments dont le premier est une clef d'accès au second. Là encore, il est judicieux de déclarer un modèle d'association, paramétré par le type de la clef et le type de la valeur (appelés *first* et *second* dans le cas des paires) On peut parler d'abstraction *fonctionnelle* par analogie au paramétrage de fonctions ordinaires.

3 Intérêt du polymorphisme paramétrique

En l'absence de polymorphisme paramétrique, on le simule principalement :

- soit en ayant une unique copie du code utilisant un type universel, par exemple `Object` en Java (traduction homogène) ;
- soit en créant une copie spécialisée du code pour chaque situation (traduction hétérogène).

Par exemple, si l'on est intéressé par la définition d'une classe `Paire` suffisamment générale, une représentation homogène consistera à typer les deux composantes de la paire par `Object`. C'est le même principe qui était utilisé dans les versions antérieures de l'API Java.

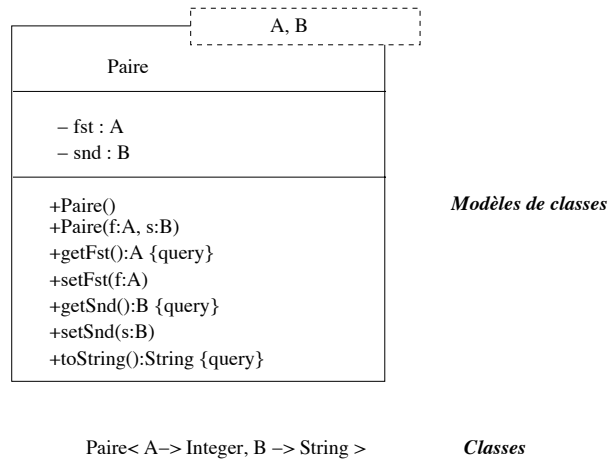


FIG. 1 – Un modèle de classe *Paire* et une instantiation notés en UML

```
public class Paire{
    private Object fst, snd;
    public Paire(Object f, Object s){fst=f; snd=s;}
    public Object getFst(){return fst;}
    ....}
```

Récupérer une valeur demandera souvent un typecast.

```
Paire p1 = new Paire("Paques",27);
String p1fst = (String)p1.getFst();
```

La traduction homogène a plusieurs inconvénients : les *cast* alourdissent le code et surtout ils ne sont vérifiés qu'à l'exécution. On peut les contrôler avec `instanceof` ou récupérer l'exception `ClassCastException` mais tout cela est bien tardif (le programme est en cours d'exploitation).

Une représentation hétérogène nécessitera de définir autant de copies de la classe que de couples de types possibles dans le contexte où l'on se trouve. Par exemple, si l'on a besoin d'une paire de chaînes de caractères et d'une paire comprenant un entier et une chaîne de caractères, on écrira le code suivant.

```
public class PaireStringString
{
    private String fst, snd;
    public Paire(String f, String s){fst=f; snd=s;}
    public String getFst(){return fst;}
    ....
}
```

```
public class PaireintString
{
    private int fst;
    private String snd;
    public Paire(int f, String s){fst=f; snd=s;}
    public int getFst(){return fst;}
    ....
}
```

La traduction hétérogène présente aussi des inconvénients : duplication excessive de code qui est source potentielle d'erreur lors de l'écriture ou de la modification du programme ; nécessité de prévoir toutes les combinaisons possibles de paramètres pour un programme donné.

En résumé, le polymorphisme paramétrique :

- évitera des duplications de code ;
- évitera des *cast* et des contrôles dynamiques au profit de contrôles effectués à la compilation ;
- facilitera l'écriture d'un code générique et réutilisable.

4 Historique de l'introduction du polymorphisme paramétrique dans Java

Une demande de modification de Java, ou *jsr* pour *java specification request* a été déposée par G. Bracha en 1999. Cette demande maintenant fort célèbre, la *jsr 014* [Bra99], décrivait les contraintes que devait respecter cette extension de Java, qui incluait notamment :

- Style Java
 - paramétrage des classes et des interfaces,
 - syntaxe proche de C++,
 - simplicité ;
- Environnement logiciel
 - compatibilité ascendante,
 - compatibilité avec l'API, y compris les exceptions,
 - les génériques sont des types comme les autres (*first class types*),
 - l'introspection doit fonctionner ;
- Tirer les leçons des études précédentes
 - Ada, Eiffel et Haskell proposent des contraintes sur les types passés en paramètres ;
- Environnement de développement
 - changer le compilateur est obligatoire,
 - la compilation séparée est préconisée,
 - les modifications doivent garantir l'efficacité de l'exécution,
 - la taille des fichiers de bytecode doit rester raisonnable,
 - la sécurité doit être respectée.

Après de nombreuses années de maturation autour de diverses propositions (Pizza [OW97], GJ [BOSW98], NextGen [CS98], MixGen [ABC03], Virtual Types [TT99], Parameterized Types [AFM97] et PolyJ [MBL97]), la généricité a finalement été ajoutée en Java sur la base du langage GJ (Generic Java) dans la version JDK 1.5 (également nommée *Tiger*). En Java 1.5, l'API, notamment les classes collections, mais aussi la classe `Class` et bien d'autres sont devenues des classes génériques. Un tutorial est proposé par Sun [Bra04].

5 Le paramétrage simple de classes et d'interfaces

5.1 Paramétrage de classe

En Java 1.5, Une classe générique admet des paramètres formels de type qui sont placés entre chevrons (comme en C++). Les paramètres formels de type sont valables

pour les variables et méthodes d'instances, mais pas pour les méthodes et variables de classes (`static`).

```
class Paire<A,B>
{
    private A fst;
    private B snd;
    public Paire(){ }
    public Paire(A f, B s){fst=f; snd=s;}
    public A getFst(){return fst;}
    public B getSnd(){return snd;}
    public void setFst(A a){fst=a;}
    public void setSnd(B b){snd=b;}
    public String toString(){return getFst()+''-''+getSnd();}
}
```

L'instanciation (ou invocation dans la terminologie Java 1.5) consiste à valuer les paramètres formels de type (on parlera alors de paramètres réels). Les paramètres réels ne peuvent pas être des types primitifs, mais ce défaut est compensé par l'une des nouveautés de Java 1.5, *l'autoboxing*, qui convertit de manière transparente les valeurs des types primitifs en instances.

```
Paire<Integer,String> p = new Paire<Integer,String>(9, ''plus grand
chiffre'');
Integer i=p.getFst();
String s=p.getSnd();
System.out.println(p);
```

5.2 Paramétrage d'interface

Il ne pose aucun problème spécifique. La définition d'une interface `Pile` pourrait se présenter comme suit.

```
interface Pile<A>
{
    boolean estVide();
    void empile(A a);
    A depile();
    Integer nbElements();
    A sommet();
}
```

Question 1 *Proposez une classe `PileListe` qui implémente l'interface `Pile`. Utilisez une liste chaînée (`LinkedList`) pour stocker les éléments de la pile. Ecrivez un petit programme qui crée et manipule des piles. Si vous deviez définir des exceptions associées à la pile (ex. pile vide), feriez-vous des classes génériques dans tous les cas ? Dans certains cas ? Jamais ?*

6 Le paramétrage de méthodes

On peut paramétrer les méthodes (méthodes d'instance ou de classe) avec des types différents de ceux de la classe générique concernée. Ci-dessous on voit ainsi le paramétrage

de la méthode de classe `copieFstTab` dont le rôle est de copier la première composante d'une paire dans un tableau à un certain indice, puis le paramétrage de la méthode d'instance `memeFst` qui compare les deux premières composantes de deux paires dont la deuxième composante n'est pas forcément de même type.

```
class Paire<A,B>
{
    .....
    public static<X,Y> void copieFstTab
        (Paire<X,Y> p,
         X[] tableau, int i)
    {tableau[i]=p.getFst();}

    .....
    public <C> boolean memeFst(Paire<A,C> p)
    {return p.getFst()==getFst();}
}
```

Voici une utilisation de ces méthodes.

```
Paire<Integer,String> p5 = new Paire<Integer,String>(9,
                                                    "plus grand chiffre");
Integer[] tab=new Integer[2];
Paire.copieFstTab(p5,tab,0);
Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
System.out.println(p5.memeFst(p2));
```

7 Le principe de l'effacement de type

En C++ l'instanciation des génériques s'appuie sur une copie du code avec remplacement des types formels par les types réels (traduction hétérogène). La compilation des génériques en Java 1.5 se base au contraire sur un procédé d'effacement de type encore appelé *type erasure* (traduction homogène). Toutes les informations de type placées entre chevrons sont effacées ; les variables de types restantes sont remplacées par la borne supérieure (`Object` s'il n'y a pas de contrainte) ; des *cast* sont insérés si nécessaire (quand le code résultant n'est pas correctement typé). Le code présenté ci-dessus pour la classe `paire` sera à peu près traduit comme le code suivant.

```
class Paire
{
    private Object fst;
    private Object snd;
    public Paire(){}
    public Paire(Object f, Object s){fst=f; snd=s;}
    public Object getFst(){return fst;}
    public Object getSnd(){return snd;}
    public void setFst(Object a){fst=a;}
    public void setSnd(Object b){snd=b;}
    public String toString(){return getFst()+''-''+getSnd();}
}
```

```
Paire p = new Paire(9, 'plus grand chiffre');
Integer i=(Integer)p.getFst();
```

Ce principe a des conséquences non négligeables : à l'exécution, il n'existe en fait qu'une classe qui est partagée par toutes les instanciations. Cela explique en particulier que les variables de type paramétrant une classe ne portent pas sur les méthodes et variables statiques.

Question 2 *Pour les variables `p2` et `p5` de la section précédente, testez `p2.getClass()==p5.getClass()`; Conclusion ?*

Question 3 *Ajoutez à la classe générique `Paire` la variable de classe `static Integer nbInstances=0`; . Incrémentez-la à chaque création de paire. Créez quelques paires avec différents types et examinez l'évolution de `nbInstances`. Obtenez-vous le même comportement qu'en C++ ?*

Une autre conséquence du fait que les paramètres de type n'existent pas à l'exécution est qu'il est déconseillé (voire interdit) de les utiliser dans le contexte de vérification de type (`instanceOf`) ou de coercition (`cast`).

Une classe générique peut par ailleurs être utilisée dans le code source sans ses paramètres, par exemple `Paire p7=new Paire()`; . On appelle ceci un type brut (*raw type*). Sauf contexte particulier, il est préférable de ne pas les utiliser car le compilateur n'effectue pas de vérifications sérieuses. Cela sert essentiellement à faire interopérer ancien code (Java 1.4) et nouveau code.

8 Polymorphisme paramétrique, sous-typage et spécialisation

Les deux mécanismes d'abstraction (généricité paramétrique et héritage) se mélangent comme le montrent les exemples ci-dessous.

Sous-typage des classes pour un paramètre fixé `PileListe<String>` est bien, comme attendu, un sous type de `Pile<String>`.

On peut par exemple écrire `Pile<String> pi=new PileListe<String>()`;

Sous-typage des paramètres D'un point de vue théorique, `Pile<String>` n'est pas un sous-type de `Pile<Object>` puisque certaines opérations admises sur une `Pile<Object>`, telles que `empile(Object o)`, ne sont pas correctes pour une `Pile<String>`. C'est ainsi que l'entend Java 1.5 également.

`PileListe<Object> pi=new PileListe<String>()` provoquera donc une erreur de compilation.

Combinaisons de dérivations et d'instanciations

Classe générique dérivée d'une classe non générique.

```
class Graphe{}
class GrapheEtiquete<TypeEtiq> extends Graphe{}
```

Classe générique dérivée d'une classe générique.

```
class TableHash<TK,TV> extends Dictionnaire<TK,TV>{}
```

Classe dérivée d'une instantiation d'une classe générique.

```
class Agenda extends Dictionnaire<Date,String>{}
```

9 Le paramétrage contraint (ou borné)

9.1 Contrainte extends

Il est souvent utile de poser des contraintes sur les types passés en paramètres :

- lorsque ceux-ci doivent fournir certains services (méthodes, attributs) ;
- plus généralement, pour exprimer qu'ils correspondent à une certaine abstraction.

Ainsi, dans la perspective de munir la classe `Paire<A,B>` d'une méthode de saisie, on veut imposer que les types `A` et `B` disposent préalablement d'une telle méthode. Java 1.5 nous offre la possibilité de le faire grâce à des bornes placées sur les paramètres de type. On définit tout d'abord un type (ici une interface) pour représenter l'abstraction « objet que l'on peut saisir ».

```
interface Saisissable
{
    public void saisie(ConsoleStandard c) throws IOException;
}
```

La classe `PaireSaisissable` peut être écrite avec des paramètres contraints à être des sous-types de `Saisissable`. Vous noterez que `PaireSaisissable` elle-même est également sous-type de `Saisissable`, il n'y a aucune obligation à cela, c'est simplement pour la logique de l'exemple.

```
class PaireSaisissable<A extends Saisissable, B extends Saisissable>
    implements Saisissable
{
    private A fst;
    private B snd;
    public PaireSaisissable(A f, B s){fst=f; snd=s;}
    public A getFst(){return fst;}
    public B getSnd(){return snd;}
    public void setFst(A a){fst=a;}
    public void setSnd(B b){snd=b;}
    public String toString(){return getFst()+''-''+getSnd();}

    public void saisie(ConsoleStandard c) throws IOException
    {
        System.out.print("Valeur first:"); fst.saisie(c);
        System.out.print("Valeur second:"); snd.saisie(c);
    }
}
```

Pour l'utiliser, il faut disposer de types concrets qui implémentent l'interface `Saisissable`. Par exemple, on peut proposer une classe qui enveloppe `String` et qui dispose de `saisie`.

```

class StringSaisissable implements Saisissable
{
    private String s;
    public StringSaisissable(String s){this.s=s;}
    public void saisie(ConsoleStandard c) throws IOException
    {s=c.lireChaine();}
    public String toString(){return s;}
}

```

Dans un programme principal on peut maintenant effectuer des saisies d'instances de paires.

```

ConsoleStandard c = new ConsoleStandard();
StringSaisissable s1 = new StringSaisissable("");
StringSaisissable s2 = new StringSaisissable("");
PaireSaisissable<StringSaisissable,StringSaisissable> mp
    = new PaireSaisissable<StringSaisissable,StringSaisissable>(s1,s2);
mp.saisie(c);

```

9.2 Contraintes multiples

On peut poser plusieurs bornes sur un paramètre de type. Par exemple, on peut contraindre les composantes de la classe *Paire* à être à la fois *Saisissables* et *Sérialisables*. Le caractère & sert à combiner des bornes.

```

class Paire<A extends Saisissable & Serializable,
           B extends Saisissable & Serializable>
{.....}

```

9.3 Contrainte récursive

Le paramètre de type peut réapparaître dans la contrainte. Par exemple, un ensemble ordonné est paramétré par le type T des éléments qui sont comparables avec des éléments du même type T.

```

interface Comparable<A>{boolean infStrict(A a);}
class orderedSet<A extends Comparable<A>>{}

```

9.4 Discussion sur le polymorphisme contraint

Exprimer les contraintes par des types (interfaces ou classes) présente :

- l'avantage de réifier les contraintes et de définir explicitement les abstractions ;
- deux inconvénients, celui de devoir définir autant de classes/d'interfaces que de contraintes et celui de devoir ajouter de nombreuses classes enveloppes pour adapter les types concrets à la contrainte.

L'une des propositions d'introduction de la généricité paramétrique (PolyJ) qui n'a pas été retenue, offrait une solution alternative intéressante, qui consistait à exprimer les contraintes par des clauses *where*. Ces clauses regroupent la liste des méthodes que doit posséder le type passé en paramètre.

```

class MaPaireSaisissable[A,B]
    where A {void saisie(ConsoleStandard c)}
        B {void saisie(ConsoleStandard c)}

```

L'avantage principal est que les types existant déjà n'ont pas besoin de classes enveloppes pour pouvoir servir de paramètre réel. L'inconvénient, qui apparaît sur cet exemple, est que les signatures peuvent être répétées. Une solution mixte incluant les deux possibilités aurait été certainement très souple à utiliser.

10 Le paramétrage par des jokers (*wildcards*)

Nous avons vu que la spécialisation des paramètres réels de type n'implique pas le sous-typage entre classes paramétrées (`Collection<Integer>` spécialise mais n'est pas un sous-type de `Collection<Object>`). Il existe cependant un super-type de toutes les instanciations d'une classe générique, qui s'écrit comme une instanciation dans laquelle les paramètres de type sont remplacés par le caractère joker `?`, par exemple, `Collection<?>`.

10.1 Utilisations simples du caractère joker

Ce super-type peut servir à typer certaines expressions, notamment des variables locales ou des attributs, ainsi qu'à simplifier ou préciser des écritures.

Voici tout d'abord un exemple où cela sert à typer une variable (mais ensuite tout n'est pas possible sur la variable).

```
Paire<?,?> p3 = new Paire<Integer, String>();
```

Question 4 Testez sur `p3` les instructions suivantes et interprétez les résultats obtenus.

```
p3.setFst(12);
System.out.println(p3);
```

Et voilà un exemple où le caractère joker sert à simplifier le code. Dans la première version de la méthode d'affichage, le nom des paramètres formels de type ne sert pas, il peut donc être évité.

```
public static<A,B> void affiche(Paire<A,B> p)
{
    System.out.println(p.getFst()+" "+p.getSnd());
}
public static void affiche(Paire<?,?> p)
{
    System.out.println(p.getFst()+" "+p.getSnd());
}
```

Question 5 Testez les deux méthodes d'affichage.

10.2 Jokers et contraintes (extends et super)

Les jokers peuvent être utilisés également avec des contraintes. Vous noterez dans les exemples proposés ici que l'utilisation des contraintes permet d'élargir le cadre d'utilisation des méthodes.

Contrainte extends On peut écrire deux variantes pour une méthode qui récupère le premier élément d'une liste et l'utilise comme valeur pour la première composante d'une paire. Il suffit que la liste soit paramétrée par le même premier paramètre de type que la paire ou par un de ses sous-types. La version avec `joker` est plus lisible, elle permet de ne pas faire mention explicite du paramètre de la liste qui ne sert que dans la signature.

```
public void prendListFst(List<? extends A> c)
{setFst(c.get(0));}

public <X extends A> void prendListFst(List<X> c)
{setFst(c.get(0));}
```

Toutes ces instructions sont alors possibles avec l'une ou l'autre des versions.

```
Paire<Object,String> p6 = new Paire<Object,String>();
List<Object> lo = new LinkedList<Object>();
List<Integer> li = new LinkedList<Integer>();
lo.add(new Integer(6)); li.add(6);
p6.prendListFst(lo);    p6.prendListFst(li);
```

Notez que l'instruction `p6.prendListFst(li);` n'aurait pas été possible avec une signature plus stricte de la méthode, c'est-à-dire de la forme `void prendListFst(List<A> c)`.

Contrainte super Dans certains cas, plutôt que de vouloir donner une borne supérieure au type, on peut vouloir le borner inférieurement.

Question 6 *Nous vous proposons de mener une étude symétrique à l'étude précédente avec une méthode `copieFstColl`, une variante de la méthode `copieFstTab`, qui écrit le premier composant d'une paire dans une collection. Une version simple serait comme suit.*

```
public void copieFstColl(Collection<A> c)
{c.add(getFst());}
```

Ecrivez cette méthode dans la classe `Paire` et testez les instructions suivantes.

```
Paire<Integer,Integer> p2 = new Paire<Integer,Integer>(9,10);
Collection<Object> co = new LinkedList<Object>();
Collection<Integer> ci = new LinkedList<Integer>();
p2.copieFstColl(co); // ne devrait pas fonctionner !!!!!
p2.copieFstColl(ci);
```

L'instruction `p2.copieFstColl(co);` n'est pas compilable en l'état actuel des choses. Pourtant, elle ne pose pas de problème de sémantique : un entier peut être inséré dans une collection d'objets. En vous inspirant de ce que nous avons fait pour la méthode `prendListFst`, utilisez le mot-clef `super` pour rendre possible cette instruction. Essayez une forme avec et une forme sans `joker`.

Notons pour conclure cette partie que `super` et `extends` ne peuvent être utilisés ensemble, et que ces deux mots-clefs ne sont utilisables pour les déclarations de variables et d'attributs que lorsqu'ils sont composés avec le `joker`, comme dans `List<? extends Number> l;` par exemple.

10.3 Capture de joker

Question 7 Soit la classe suivante [Bra04].

```
public class TestInferenceCapture
{
    public static <T> List<T> unmodifiableList(List<T> liste){return liste;}
}
```

Tentez votre chance avec les instructions suivantes.

```
LinkedList<Integer> li=null;
List<Integer> lli=unmodifiableList(li);
List<?> ll=new LinkedList<String>();
List<?> lll = unmodifiableList(ll);
```

Même question avec la méthode suivante, dans laquelle on utilise les jokers, et donc dans laquelle on ne vérifie pas que les types sont les mêmes dans la liste passée en paramètre et dans la liste retournée.

```
public static List<?> unmodifiableList(List<?> liste){return liste;}
```

Ce que vous venez d'expérimenter avec l'appel `unmodifiableList(ll)` ; et la première version de la méthode (sans joker) est appelé « capture de joker ». Le type réel correspondant au joker dans l'invocation (ici `String`, le type dit *capturé*) va être associé au type `T` lors de l'invocation de la méthode.

11 Problèmes

11.1 Paramétrage contraint - Couples de toutes sortes

Question P1.1 Définissez une interface `Mâle` et une interface `Femelle`. Définissez, en spécialisant la classe `Paire`, la classe générique `CoupleConventionnel` qui représente les couples (ordonnés) constitués d'un mâle et d'une femelle. Instanciez-la pour créer des couples de chats et de chattes. Remarquez que l'on peut ainsi créer des couples constitués d'un dauphin et d'une chatte.

Question P1.2 Proposez, en dérivant la classe `Paire`, une classe générique `CoupleEspèce` pour représenter les couples constitués de deux membres de la même espèce. Instanciez-la. Vous ne devez plus pouvoir créer des couples constitués d'un dauphin (cétacé) et d'un chat (félin).

Question P1.3 Proposez une classe générique `CoupleFertile` pour représenter les couples constitués d'un mâle et d'une femelle de la même espèce. Instanciez-la pour créer des couples de chats et de chattes. Vous ne devez plus pouvoir créer des couples constitués de deux mâles, même s'ils appartiennent à une même espèce.

11.2 Fonctions partielles

Nous vous proposons d'étudier en Java 1.5 la modélisation et l'implémentation d'une classe représentant les fonctions partielles définies en extension et dont le domaine et le codomaine sont finis.

Pour donner un exemple, une fonction partielle particulière serait déterminée par :

- un ensemble de départ, $E = \{Sylvie, Astrild, Nestor, Geraldine, Hector\}$
- un ensemble d'arrivée, $F = \{13, 12, 9, 8\}$
- une partie G du produit cartésien $E \times F$ telle que pour tout $x \in E$, il existe au plus un $y \in F$ avec $(x, y) \in G$,
par ex. $G = \{(Sylvie, 12), (Astrild, 13), (Nestor, 9), (Geraldine, 12)\}$

La fonction est totale si pour tout $x \in E$, il existe un $y \in F$ avec $(x, y) \in G$, ce qui n'est pas le cas dans l'exemple ci-dessus.

Nous supposons que E et F sont des ensembles de valeurs de types références (donc des valeurs de sous-classes d'`Object` et non de types primitifs), que l'on peut noter respectivement TE et TF .

Question P2.1 Proposez la partie d'une classe paramétrée représentant les fonctions partielles comprenant :

- la déclaration de la classe, paramètres, etc.,
- les attributs nécessaires à la représentation mémoire (il est conseillé d'utiliser les classes paramétrées `Set` et `HashMap` dont les principales fonctionnalités sont disponibles en ligne),
- un constructeur,
- quelques méthodes simples pour créer une fonction, savoir si elle est totale.

Question P2.2 Instanciez cette classe paramétrée pour créer la fonction donnée dans l'exemple ci-dessus.

Question P2.3 Dérivez de la définition de la classe fonction une classe permettant de représenter des fonctions arithmétiques (les ensembles de départ et d'arrivée contiennent obligatoirement des instances de `Number`).

Question P2.4 Pour implémenter la composition de fonctions, il y a deux possibilités théoriquement :

- écrire une fonction (méthode `static`) qui prenne deux fonctions en arguments, par ex. $f : E \rightarrow F$ et $g : F \rightarrow G$ et qui retourne la composée de f et de g , soit $h : E \rightarrow G$, avec $h(x) = g(f(x))$
- écrire une méthode qui prend donc seulement $g : F \rightarrow G$ comme argument et retourne la composée de `this` (supposée de $E \rightarrow F$) et de g , soit $h : E \rightarrow G$, avec $h(x) = g(f(x))$

Etudier ces deux implémentations.

Références

- [ABC03] Allen, Bannet, and Cartwright. A First-Class Approach to Genericity. In *OOPSLA 2003*, 2003.
- [AFM97] Ole Agesen, Stephen N. Freund, , and John C. Mitchell. Adding Type Parameterization to the Java Language. In *OOPSLA 97*, 1997.

- [BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past : Adding Genericity to the Java Programming Language. In *OOPSLA 98*, Vancouver, October 1998.
- [Bra99] Gilad Bracha. JSR 14 : Add Generic Types To The Java™ Programming Language. <http://jcp.org/en/jsr/all>, 1999.
- [Bra04] Gilad Bracha. Generics in the Java Programming Language. <http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>, 2004.
- [CS98] Cartwright and Steele. Compatible Genericity with Runtime Types for the Java programming language. In *OOPSLA 98*, Vancouver, October 1998. <http://www.cs.rice.edu/javaplt/nextgen/>.
- [MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized Types for Java. In *ACM POPL '97*, Paris, France, January 1997. <http://www.pmg.csail.mit.edu/polyj/>.
- [OW97] Martin Odersky and Philip Wadler. Pizza into Java : Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, Paris, France, january 1997.
- [TT99] Kresten Krab Thorup and Mads Torgersen. Unifying genericity : Combining the benefits of virtual types and parameterized classes. In Springer-Verlag, editor, *ECOOP 1999*, number 1628 in LNCS, pages 186–204, Lisbon, Portugal, June 1999.