

Typage et Analyse Statique

Cours 5

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Université Pierre et Marie Curie

année 2014-2015

Plan du cours

- ▶ génériques en Java
 - ▶ passage de Java 1.4 à 1.5
 - ▶ classes et méthodes paramétrées
 - ▶ typage et sous-typage
 - ▶ polymorphisme borné
- ▶ Panorama des langages

Génériques en Java 1.5

Introduction du polymorphisme paramétrique (même code quelque soit le type).

- ▶ But : manipuler des classes paramétrées
 - ▶ pour un code plus sûr
 - ▶ et plus lisible
- ▶ Contraintes :
 - ▶ utiliser la même machine virtuelle
 - ▶ être compatible ascendant (programmes 1.4 compilables)

Motivations

- ▶ typage statique générique pour
 - ▶ diminuer les tests dynamiques de types
 - ▶ écrire des structures de données génériques classiques et effectuer des calculs dessus
 - ▶ faciliter la lecture des programmes
- ▶ répondre aux critiques d'autres langages :
 - ▶ C++, Ada95, O'Caml, Haskell ...
- ▶ tenir compte de propositions d'extension :
 - ▶ Pizza, GJ, ...
- ▶ répondre à l'avance à C#

Contraintes

- ▶ compatible avec les versions antérieures :
 - ▶ du langage
 - ▶ des bibliothèques
 - ▶ de la machine abstraite
- ▶ cohabitation possible entre codes/bibliothèques antérieurs
- ▶ ne pas être coûteux si on ne s'en sert pas

Influences

- ▶ polymorphisme paramétrique (ML,)
- ▶ propositions Pizza et GJ :
 - ▶ Pizza : <http://pizzacompiler.sourceforge.net/>
 - ▶ GJ : : <http://homepages.inf.ed.ac.uk/wadler/pizza/gj/>
- ▶ Génériques pour C# et .NET :
<http://research.microsoft.com/projects/clrgen/>
- ▶ un bon livre :
 - ▶ Génériques et collections Java, Naftalin-Wadler, O'Reilly

Java 1.4 : API

vecteurs extensibles

```
1 java.util
2   Class ArrayList
```

hiérarchie de classes

```
1 java.lang.Object
2   java.util.AbstractCollection
3     java.util.AbstractList
4       java.util.ArrayList
```

principales méthodes

```
1 ArrayList(int initialCapacity)
2 void add(int index, Object element)
3 Object get(int index)
4 Object set(int index, Object element)
```

Java 1.4 : utilisation (UD.java)

```
1 import java.util.ArrayList;
2 class UD {
3     public static void main(String[] a) {
4         ArrayList all=new ArrayList(3);
5         all.add(0,new Integer(3));
6         all.add(1,"salut");
7         Integer x = (Integer)(all.get(0));
8         Integer y = (Integer)(all.get(1));        //
9         int res = x.intValue() + y.intValue();
10    }
11 }
```

► compilation : javac -source 1.4 UD.java

► exécution : runtimeException

```
$ java UD
```

```
Exception in thread "main"
```

```
java.lang.ClassCastException: java.lang.String
    at UD.main(UD.java:8)
```


Java 1.5 : API

vecteurs extensibles

```
1 java.util
2   Class ArrayList<E>
```

hiérarchie de classes

```
1 java.lang.Object
2   java.util.AbstractCollection<E>
3     java.util.AbstractList<E>
4       java.util.ArrayList<E>
```

principales méthodes

```
1 ArrayList(int initialCapacity)
2 void add(int index, E element)
3 E get(int index)
4 E set(int index, E element)
```

Java 1.5 : utilisation (UD.java)

⇒ warnings à la compilation

```
1 $ javac -source 1.5 UD.java
2 Note: UD.java uses unchecked or unsafe operations.
3 Note: Recompile with -Xlint:unchecked for details.
4 -bash-3.00$ javac -source 1.5 -Xlint:unchecked UD.java
5 UD.java:5: warning: [unchecked] unchecked call to add(int,E) as a member of ↵
      the raw type java.util.ArrayList
      all.add(0,new Integer(3));
          ^
6 UD.java:6: warning: [unchecked] unchecked call to add(int,E) as a member of ↵
      the raw type java.util.ArrayList
      all.add(1,"salut");
          ^
7
8
9
10
11 2 warnings
```

⇒ exception à l'exécution

```
1 $ java UD
2 Exception in thread "main"
3 java.lang.ClassCastException: java.lang.String
4 at UD.main(UD.java:8)
```

Java 1.5 : classe paramétrée (US.java)

```
1 import java.util.ArrayList;
2 class US {
3     public static void main(String[] a) {
4         ArrayList<Integer> all=
5             new ArrayList<Integer>(3);
6         all.add(0,new Integer(3));
7         all.add(1,"salut");
8         Integer x = (Integer)(all.get(0));
9         Integer y = (Integer)(all.get(1));
10        int res = x.intValue() + y.intValue();
11    } }
```

⇒ erreur à la compilation

```
1 US.java:7: cannot find symbol
2 symbol   : method add(int,java.lang.String)
3 location: class java.util.ArrayList<java.lang.Integer>
4     all.add(1,"salut");
5         ^
6 1 error
```

Exemples (1)

Classe et interface:

```
1 interface Comparable<E> {  
2     int compareTo(E e)  
3 }
```

```
1 public class H implements Comparable<H> {  
2     // ...  
3     H max(H e) {  
4         if (this.compareTo(e) > 0)  
5             {return this;}  
6         else {return e;}  
7     }
```

Exemples (2)

Structures de données génériques:

```
1  class Arbre<T> {
2      private T etiq;
3      private List<Arbre<T>> fils =
4          new ArrayList<Arbre<T>>();
5
6      public Arbre<T>(T etiq){this.etiq=etiq;}
7      public T getEtiq(){return etiq;}
8      public List<Arbre<T>> getFils(){return fils;}
9      public void ajouteFils(Arbre<T> f){fils.add(f);}
10     //...
11 }
```

Contraintes sur les variables de type

Une variable de type peut avoir une ou plusieurs bornes (séparées par &).

- ▶ borne supérieure : <T extends Object>
- ▶ combinée : &

```
1  static <T extends Clonable & Closable> T m(T x) {  
2      T y = x.clone();  
3      x.close();  
4  }
```

- ▶ récursive

```
1  public interface Orderable<T extends Orderable<T>> {  
2      Public boolean lessThan(T t);  
3  }
```

2 modèles:

- ▶ expansion de code (à la C++)
chaque instance d'une classe paramétrée a sa propre version de code
- ▶ le paramètre de type est remplacé par Object
un code unique
avec des transtypages sûrs

Limitations

- ▶ paramètre de type instancié par une classe ou une interface
pas par un type primitif
⇒ auto-boxing (voir transparents suivants)
- ▶ pas de manipulation du paramètre de type à l'exécution :
 - ▶ pas de new sur un type paramétré
 - ▶ pas d'héritage sur un type paramétré
 - ▶ pas de cast avec type paramétré (warning)
 - ▶ ni d'instanceof (erreur), ni de catch (erreur)
 - ▶ pas de type paramétré comme type des éléments d'un tableau :
car il faudrait garder le type du paramètre de type dans le
tableau pour vérifier les relations de sous-typage des tableaux.

Conversion de types explicite

opération de transtypage : $(\tau)expr$

indique que l'expression $expr$ doit être considérée de type τ

- ▶ types primitifs
 - ▶ sans perte d'information : octet vers entier
 - ▶ avec perte d'information : flottant vers entier
- ▶ types de valeurs allouées (objets ou tableaux)
 - ▶ même référence vue d'un autre type
 - ▶ selon la relation de sous-typage, ajoute ou non un test dynamique pour vérifier la correction du transtypage
 - ▶ si $\tau_{expr} \leq \tau$: code correct du point de vue des types

```
1 PointColore pc0 =  
2   new PointColore(2,3, "Bleu");  
3 String s0 = pc0.getCouleur();  
4 Point p2 = (Point)pc0;
```

- ▶ si $\tau_{expr} \not\leq \tau$: nécessite l'ajout d'un test dynamique

```
1 String s1 = ((PointColore) p2).getCouleur();
```

Conversion de types implicite

- ▶ en cas d'affectation ou de passage de paramètre, si la valeur passée est d'un sous-type de la valeur attendue :

```
1 Point p0 = new PointColore();
```

- ▶ pour certains opérateurs : arithmétiques ou de concaténation des chaînes de caractères

```
1 int x = 3.14 + 4;  
2 String s0 = pc0 + " " + p0;
```

- ▶ dans le cas où il faut transformer la valeur d'un type primitif vers sa valeur «équivalente» de sa classe associée (classe encapsulante ou *wrapper*) : autoboxing

Simplification des déclarations

Depuis Java 1.7, il y a de l'inférence de types pour éviter de répéter des types pour les classes paramétrées ; on utilise l'opérateur <> pour l'instanciation des classes :

```
1 List<String> l = new ArrayList<>();
2 Map<String, Integer> m1 = new HashMap<>();
3 Map<String, List<String>> m2 = new HashMap<>();
```

Le compilateur Java déduit les types des constructeurs si cela est possible, sinon déclenche une erreur à la compilation.

```
1 ArrayList<String> al = new ArrayList<>();
2 al.add('hello');
3 al.addAll(new ArrayList<>);
```

⇒ erreur ligne 3

Méthodes paramétrées

Possibilité de définir le paramètre de type au niveau d'une méthode (devant son type de retour) :

```
1 class G {
2     public static <T> void arrayToList (T[] a, List<T> l) {
3         for ( T elt : a ) { l.add(elt); }
4     }
5     public static void main(String[] args) {
6         ArrayList<String> als = new ArrayList<String>();
7         arrayToList(args,als);
8         System.out.println("==> " + als);
9     }
10 }
```

```
1 $ java G un deux trois
2 ==> [un, deux, trois]
```

Intérêt :

- ▶ Si la généricité d'une méthode est indépendante de celui de la classe de définition, alors il n'est pas dangereux de le lier localement en indiquant les paramètres de types de la méthode.

Méthodes paramétrées : autre exemple

```
1 import java.util.*;
2 interface Comparator<T> {
3     public int compare(T x, T y); }
4 class ByteComparator implements Comparator<Byte> {
5     public int compare (Byte x, Byte y) { return (x - y);} }
6
7 class Collections {
8     public static <T> T max(Collection<T> col,
9                             Comparator<T> cmp) {
10         Iterator<T> it = col.iterator();
11         T elt = it.next();
12         while (it.hasNext()) {
13             T elt2 = it.next();
14             if (cmp.compare(elt,elt2) < 0 ) elt = elt2;
15         }
16         return elt;
17     }
18 }
```

Méthodes paramétrées en OCaml

En OCaml <http://caml.inria.fr/pub/docs/manual-ocaml/manual005.html> :

```
1 # class intlist (l : int list) =
2   object
3     method empty = (l = [])
4     method fold : 'a. ('a -> int -> 'a) -> 'a -> 'a =
5       fun f accu -> List.fold_left f accu l
6     end;;
7
8 # let l = new intlist [1; 2; 3];;
9 val l : intlist = <obj>
10
11 # l#fold (fun x y -> x+y) 0;;
12 - : int = 6
13
14 # l#fold (fun s x -> s ^ string_of_int x ^ " ") "";;
15 - : string = "1 2 3 "
```

Typage et sous-typage

- ▶ un type paramétré $\tau_2 \langle T_2 \rangle \leq \tau_1 \langle T_1 \rangle$ ssi :
 - ▶ $\tau_2 \leq \tau_1$
 - ▶ et $T_2 = T_1$
- ▶ un type paramétré $\tau \langle T \rangle \leq \tau$
- ▶ un type paramétré $\tau \langle T \rangle \leq \text{Object}$
- ▶ $\tau_2 \langle T_2 \rangle$ n'est pas sous-type de $\tau_1 \langle T_1 \rangle$ si $T_2 \neq T_1$

Typage et sous-typage (1)

- ▶ effacement de type

raw type : type paramétré sans paramètre (compatibilité)

- ▶ `Type<A>` vers `RawType`
- ▶ `RawType` vers `Type<A>` : warning

```
1 ArrayList<String> als = new ArrayList<String>(10);
2
3 ArrayList al = als; // ok
4
5 als = al; // warning a' la compilation
6 als = new ArrayList(); // idem
```


Typage et sous-typage (2)

- ▶ pas de sous-typage direct sur les paramètres de types : erreur voir polymorphisme borné

```
1 ArrayList<String> as = new ArrayList<String>(3);
2 ArrayList<Object> ao = as;
3 ...
4 incompatible types
5 found   : java.util.ArrayList<java.lang.String>
6 required: java.util.ArrayList<java.lang.Object>
7     ArrayList<Object> ao = as;
```

- ▶ pas de création de tableaux paramétrés : erreur

```
1 A[] aa = new A[10];
```

Compatibilité ascendante

- ▶ Tous les anciens programmes Java tournent.
- ▶ pas d'information sur les paramètres de type dans la représentation des objets
- ▶ pas d'introspection ou de test de types sur le paramètre de type

Warning à la compilation

```
1 javac -Xlint:unchecked UD.java
```

Danger : voir de 2 manières une même structure

```
1 ArrayList<String> als = new ArrayList<String>(10);
2 ArrayList al = als;
3
4 als.add(0,"Salut");
5 // als.add(1,new Integer(4)); erreur compilation
6
7 al.add(1,new Integer(4)); // warning a' la compilation
8
9 String s = als.get(1); // erreur a' l'exe'cution
10 // Exception in thread "main"
11 // java.lang.ClassCastException:
12 //java.lang.Integer cannot be cast to java.lang.String
```

pas de *unchecked warning*

⇒ pas de `RuntimeException` *ClassCastException*

Exemple : QueueD.java (1)

```
1 import java.util.ArrayList;
2
3 class Vide extends Exception {}
4 class Pleine extends Exception {}
5
6 class QueueD {
7     int taille, longueur;
8     ArrayList q;
9     int tete, fin;
10
11     QueueD(int n) {taille = n; q = new ArrayList(n);}
12
13     void entrer(Object x) throws Pleine {
14         if (longueur < taille) { q.add(fin++ % taille,x); longueur++;}
15         else throw new Pleine();
16     }
17
18     Object partir() throws Vide {
19         if (longueur > 0) { longueur--; return q.get(tete++ % taille);}
20         else throw new Vide();
21     }
22 }
```

Exemple : Queue.java (2)

```
1
2  class Vide extends Exception {}
3  class Pleine extends Exception {}
4
5  class Queue<A> {
6      int taille, longueur;
7      A[] q;
8      int tete, fin;
9
10     Queue(int n) {taille = n; q = new A[n];}
11
12     void entrer(A x) throws Pleine {
13         if (longueur < taille) { q[fin++ % taille] = x; longueur++;}
14         else throw new Pleine();
15     }
16
17     A partir() {
18         if (longueur > 0) { longueur--; return q[tete++ % taille];}
19         else throw new Vide();
20     }
21 }
```

Exemple : QueueS.java (3)

```
1 import java.util.ArrayList;
2
3 class Vide extends Exception {}
4 class Pleine extends Exception {}
5
6 class QueueS<A> {
7     int taille, longueur;
8     ArrayList<A> q;
9     int tete, fin;
10
11     QueueS(int n) {taille = n; q = new ArrayList<A>(n);}
12
13     void entrer(A x) throws Pleine {
14         if (longueur < taille) { q.add(fin++ % taille, x); longueur++;}
15         else throw new Pleine();
16     }
17
18     A partir() throws Vide {
19         if (longueur > 0) { longueur--; return q.get(tete++ % taille);}
20         else throw new Vide();
21     }
22 }
```

Exemple : compilation (4)

▶ QueueD :

```
1 $ javac -Xlint:unchecked QueueD.java
2 QueueD.java:14: warning: [unchecked] unchecked call to add(int,E) as a ←
   member of the raw type java.util.ArrayList
3     if (longueur < taille) { q.add(fin++ % taille,x); longueur++;}
4                             ^
5 1 warning
```

▶ Queue :

```
1 $ javac -Xlint:unchecked Queue.java
2 Queue.java:10: generic array creation
3     Queue(int n) {taille = n; q = new A[n];}
4                             ^
5 1 error
```

▶ QueueS :

```
1 $ javac -Xlint:unchecked QueueS.java
```

Exemple : exécution (5)

► UQS.java

```
1 import java.util.ArrayList;
2 class UQS {
3     public static void main(String[] a) {
4         QueueS<Integer> q = new QueueS<Integer>(3);
5         try { q.entrer(new Integer(3));
6             q.entrer(new Integer(4));
7             Integer x = q.partir();
8             Integer y = q.partir();
9             int res = x.intValue() + y.intValue();
10            System.out.println(res);
11        } catch (Exception e) {System.out.println(e.toString());}
12    } }
```

► compilation :

```
1 $ javac -Xlint:unchecked UQS.java
```

► exécution

```
1 $ java UQS
2 7
```


Nouvelles erreurs à la compilation (1)

du à l'effacement du paramètre de type

- ▶ en cas de redéfinition
- ▶ en cas de surcharge

Exemple de surcharge:

```
1 class K {  
2     int m(ArrayList<Integer> x) {return x.get(0).intValue();}  
3     int m(ArrayList<String> x) {return x.get(0).length();}  
4 }
```

```
$ javac K.java
```

```
K.java:7: error: name clash: m(ArrayList<String>)  
    and m(ArrayList<Integer>) have the same erasure  
    int m(ArrayList<String> x) {return x.get(0).length();}  
        ^
```

```
1 error
```

Nouvelles erreurs à la compilation (2)

Exemple de redéfinition:

```
1 import java.util.*;
2
3 class K1 {
4     int m(ArrayList<Integer> x) {
5         return x.get(0).intValue(); }
6 }
```

```
1 import java.util.*;
2
3 class K2 extends K1 {
4     int m(ArrayList<String> x) {return x.get(0).length();}
5 }
```

```
javac K1.java K2.java
K2.java:3: name clash: m(java.util.ArrayList<java.lang.String>)
in K2 and m(java.util.ArrayList<java.lang.Integer>)
in K1 have the same erasure, yet neither overrides the other
class K2 extends K1 {
^
1 error
```

Polymorphisme borné (1)

Sous-typage sur le paramètre de type en définissant une inconnue de type, notée ?, sur laquelle portera les différentes contraintes de typage.

```
1 ArrayList<?> al = als;
```

- ▶ C borne supérieure : ? extends C
? une inconnue de type sous-type de C pour la covariance
- ▶ C borne inférieure : ? super C
? une inconnue de type sur-type de C

<?> indique <? extends Object>

Polymorphisme borné (2)

On récupère la relation de sous-typage grâce aux bornes et au sens de la variance :

- ▶ co-variance : `List<? extends Number>` sera sous-type de `List<T>` où `T` est sous-type de `Number`
utilisée pour la lecture d'une valeur
- ▶ contra-variance : `List<? super Number>` sera sous-type de `List<T>` où `T` est sur-type de `Number`
utilisée pour l'écriture d'une valeur

exemple dans Collections:

```
1 public static <T> void copy(List<? super T> dest,  
2                          List<? extends T> src)
```

On peut indiquer plusieurs bornes aux inconnues de type à l'aide de &.

Polymorphisme borné (3)

exemple de wikipédia :

```
1 import java.util.*;
2
3 class Ext {
4     public static void main(String[] args) {
5
6         List<? extends Number> c =
7             new ArrayList<Integer>(); // Read-only,
8 // c.add(new Integer(3)); // ? Aucune connaissance des sous-classes
9         Number n = c.get(0);
10 // c.add(n); // n n'est pas compatible avec ?
11         List<? super Integer> d = new ArrayList<Number>(); // Write-only,
12         d.add(new Integer(3)); // ok
13 // Integer i = d.iterator().next(); // erreur ? Comme Object
14         Object o = d.iterator().next(); // ok
15         List<?> e = new ArrayList<Integer>();
16         System.out.println(" e.size() : " + e.size());
17 // c.add(new Integer(5)); // erreur de compil
18     }
19 }
```

Polymorphisme borné (4)

API:

```
1 ArrayList(Collection<? extends E> c)
2   Constructs a list containing the elements of
3   the specified collection, in the order
4   they are returned by the collection's
5   iterator.
```

inférence de types pour la détermination des inconnues.

Exemple

```
1  import java.util.*;
2  class A<T> {
3      T x;
4      void set_x(T x) {this.x = x;}
5      T get_x() {return x;} }
6
7  class B<T> extends A<T>{
8      T y;
9      void set_y(T y) {this.y = y;}
10     T get_y() {return y;} }
11
12  class H {
13     public static void main(String[] a) {
14         A<? extends B> v1 = new B<B>();
15         A<? extends A> v2 = v1;
16         A<? extends Object> v3 = v2;
17
18         A<? super A> v4 = new B<Object>();
19         A<? super B> v5 = v4;
20         A<?> v6 = v5;
21     }
22 }
```


Variances

- ▶ tableaux : *covariance* avec informations de types dans les valeurs

`S[]` est sous-type de `T[]`, si `S` est sous-type de `T`

- ▶ instance de classes paramétrées :

- ▶ *covariance*

`List<S>` est sous-type de `List<? extends T>`
si `S` est sous-type de `T`

- ▶ *contravariance*

`List<S>` est sous-type de `List<? super T>`
si `S` est sur-type de `T`

Exemple 2 (1)

```
1 import java.util.ArrayList;
2
3 class Vide extends Exception {}
4 class Pleine extends Exception {}
5
6 public class QueueSW<A> {
7     int taille, longueur;
8     ArrayList<A> q;
9     int tete, fin;
10
11     QueueSW(int n) {taille = n; q = new ArrayList<A>(n);}
12
13     void entrer(A x) throws Pleine {
14         if (longueur < taille) { q.add(fin++ % taille, x); longueur++;}
15         else throw new Pleine();
16     }
17
18     A partir() throws Vide {
19         if (longueur > 0) { longueur--; return q.get(tete++ % taille);}
20         else throw new Vide();
21     }
22 }
```

Exemple 2 (2)

```
1  class K {
2      public static void main(String[] a){
3          try {
4              QueueSW<Number> q1 = new QueueSW<Number>(5);
5              q1.entrer(new Integer(3));
6              q1.entrer((Number) new Integer(2));
7              q1.entrer(new Double (2.2));
8
9              // q1.entrer((Object) new Integer(4));
10
11             Object o = q1.partir();
12             Number n = q1.partir();
13
14             // Integer x = q1.partir();
15
16         } catch (Exception e) {}
17     }
18 }
```

Modèles et Génériques : Composite et Visiteur (Partiel L3)

```
1  abstract class Visiteur<T> {
2      public abstract T visite(CteV c);
3      public abstract T visite(AddV a);
4      public abstract T visite(MultV m);
5  }
6  abstract class ExprArV {
7      public abstract <T> T accepte(Visiteur<T> v);
8  }
9  class AddV extends OpBinV {
10     public AddV(ExprArV fg, ExprArV fd){
11         this.fg = fg; this.fd = fd;
12     }
13     public <T> T accepte(Visiteur<T> v){
14         return v.visite(this);
15     }
16 }
17 class VisiteurEval extends Visiteur<Integer> {
18     public Integer visite(AddV a){
19         Integer i1,i2;
20         i1=a.sous_expr_g().accepte(this);
21         i2=a.sous_expr_d().accepte(this);
22         return (i1 + i2);
23     }
24 }
25 // A COMPLETER...
```

Compilation (1)

inférence de types: sur les inconnues ?

- ▶ capture (liaison) d'un ? avec un paramètre de type (`<T>`) :
 - ▶ liaison unique (en dehors du type de retour)
 - ▶ et le type paramétré n'est pas argument d'un autre type paramétré
- ▶ aide à l'inférence en indiquant explicitement les paramètres de type de retour

Compilation (2)

- ▶ code compatible 1.4
- ▶ pas de changement de machine virtuelle
- ▶ remplace les types paramétrés par les types sans paramètres :
 - ▶ pas d'information du paramètre de type à l'exécution
 - ▶ ajoute des tests de typage dynamiques (warning)
 - ▶ ne va pas plus vite
- ▶ limite les possibilités de *debug*

Compilation (3)

- ▶ papiers sur Pizza pour les techniques de compilation :
 - ▶ monomorphisation : code spécialisé pour chaque paramètre de type instancié
 - ▶ tests de typage dynamiques : code plus compact mais plus lent
- ▶ papiers sur le CLR de .NET modifié pour intégrer les *generics* de C#

⇒ la compatibilité Java coûte cher.

Autres lectures (1)

- ▶ livre de Wadler (O'reilly) : Génériques et collections en Java
- ▶ cours de Forax (Mlv) :
<http://www-igm.univ-mlv.fr/~forax/ens/java-avance/cours/pdf/>
- ▶ cours de Barthélémy (Cnam) :
<http://deptinfo.cnam.fr/~barthe/00/typage-genericite.pdf>
- ▶ tutorial GJ :
<http://www.cis.unisa.edu.au:80/~pizza/gj/Documents/>
- ▶ tutorial Java 1.5 :
<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>
- ▶ tutorial Pizza :
<http://pizzacompiler.sourceforge.net/doc/tutorial.html>
- ▶ sur .NET et les generics C# :
<http://research.microsoft.com/projects/clrgen/>

Introduction des Lambda-expressions'

Pour répondre à ces inconvénients, Java 1.8 introduit les traits de la programmation fonctionnelle :

- ▶ des lambda-expressions (notées λ -expressions)
- ▶ passage de paramètres fonctionnels
- ▶ composition de calculs
- ▶ retour d'une valeur fonctionnelle d'une fonction ou d'une méthode
- ▶ typage des lambda-expressions via des interfaces fonctionnelles le tout dans le cadre typé à la Java :
 - ▶ interface fonctionnelle
 - ▶ API Stream

Création d'une valeur fonctionnelle

Syntaxe:

(parametres) -> corps_de_la_fonction

- ▶ si le corps de la fonction est une expression
- ▶ si le corps de la fonction est une suite d'instructions :
les encadrer par des accolades, et utiliser l'instruction return

```
1 (n) -> n +1 // fonction successeur
2 (a,b) -> a * b // fonction multiplication
3 (Etudiant et) -> (et.getAge() >= 16) // ve'rififier si un e'tudiant
4 && (et.getAge() <= 23) // a entre 16 et 23 ans
```

```
1 (Etudiant et) -> { int age = et.getAge();
2 System.out.println(age);
3 return (age >= 16) && (age <= 23); }
```

Comme pour les classes locales ou anonymes, les lambda-expressions peuvent capturer des variables.

Interface fonctionnelle

On parle d'interface fonctionnelle pour les interfaces ne contenant qu'une seule déclaration de méthode. De nombreuses interfaces fonctionnelles sont définies dans `java.util.function`.

```
1 public interface Predicate<T> {
2     boolean test (T t) ;
3 }
4 public interface Function<T,R> {
5     R apply(T t) ;
6 }
```

On associera une telle interface comme type d'une λ -expression, qui se devra de respecter la signature de la méthode déclarée. Il n'y a pas de type fonctionnel directement manipulable.

```
1 Predicate<Etudiant> estJeune =
2     (Etudiant et) ->
3     (et.getAge() >= 16) && (et.getAge() <= 23) ;
```

Une λ -expression peut être considérée comme une implantation anonyme d'une telle interface. Elle se doit d'être compatible au niveau des types (paramètres formels et type de retour)).

Référencement de méthode

Si on désire utiliser une méthode statique existante à l'aplac d'une fonction, il est alors possible de la référencer par la notation `::`. Les références sur les méthodes d'instance ou de classe pour un objet ou une classe données, utilisent la même notation.

```
String::valueOf      x -> String.valueOf(x)
Object::toString    x -> x.toString()
x::toString         () -> x.toString()
ArrayList::new      () -> new ArrayList<>()
```

Utilisation d'une valeur fonctionnelle

- ▶ fonction réflexe pour l'interface graphique, ici l'interface `MouseListener` ne possède qu'un seul champ : `void pousePressed(MouseEvent e)`

```
1 MouseListener clic = (MouseEvent e) -> {
2     int x = e.getX();
3     int y = e.getY();
4     System.out.println("x = " + x + " y = "+y) ;
5     getGraphics().drawString("salut le monde", x, y)
6 } ;
7 addMouseListener(clic) ;
```

- ▶ et de manière plus concise :

```
1 MouseListener clic =
2     (e) -> getGraphics().drawString("salut le
3     monde",e.getX(),e.getY());
4 addMouseListener(clic) ;
```

Composition de calculs et de l'ordre supérieur

On peut ainsi définir de nouveaux itérateurs basé sur les fonctions :

- ▶ `map` qui applique une fonction aux éléments d'une liste et retourne la liste des résultats
- ▶ `filter` qui retourne les éléments d'une liste qui vérifient un prédicat
- ▶ `compose` qui compose deux fonctions g et f et calcule $g(fx)$

```
1 public <T, R> List<R> map(List<T> l, Function<T, R> f) ;  
2 public <T> List<T> filter(List<T> l, Predicate <T>pred) ;  
3 public <R,S,T> T compose(Function <S,T> g, Function<R,S> f, <R> x) ;
```

```
1 public <T> List<T> filter(List<T> l, Predicate<T> pred) {  
2     List<T> r = new ArrayList<T>();  
3     for (T e : l) {     if (pred.test(e)) r.add(e);     }  
4     return r;  
5 }
```

```
1 Predicate<Etudiant> estJeune =  
2     (et) -> ((et.getAge() >=16) && (et.getAge() <= 23)) ;  
3 List<Etudiant> le2 = filter(le,estJeune);
```

Un exemple complet d'utilisation de λ -expressions

On cherche à récupérer la liste des noms des étudiants jeunes d'une liste d'étudiants, sans tenir compte des étudiants avec un nom vide.

```
1 public List <String> getNomsEtudiantsJeunes(List<Etudiant> etudiants) {  
2     List<Etudiant> l = filter(etudiants, (et) ->  
3         (et.getAge() >= 16) && (et.getAge() <= 23)) ;  
4     List<Etudiant> l2 = map(l, (p) -> p.getNom()) ;  
5     List<Etudiant> l3 = filter(l2, (n) -> (! (n.getNom().equals("")))) ;  
6     return l3.reduce("", (r,n) -> r + "," + n);  
7 }
```

La classe Etudiant possède les méthodes getAge et getNom.
Les collections (voir cours 8) possèdent des méthodes utilisant des λ -expressions comme map, filter et reduce.

Api Stream

L'Api Stream pour les flux facilite la composition fonctionnelle de calculs, et permet de les paralléliser facilement.

Quelques signatures de méthodes :

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
Stream<T> filter(Predicate<? super T> predicate)
void forEach(Consumer<? super T> consumer)
T reduce(T identity, BinaryOperator<T> accumulator)
...
```

```
1 public String getNomsEtudiantsJeunes(List<Etudiant> etudiants) {
2     return etudiants // tous les etudiants
3         .stream() // dans un flux
4         .filter((Etudiant et) -> // filtrer les jeunes
5             (et.getAge() >= 16) && (et.getAge() <= 23))
6         .map(p -> p.getNom()) // recuperer les noms
7         .filter(n -> !(n.getNom().equals(""))) // sauf les ""
8         .reduce("", (res, n) -> res + ", " + n) // les concatener tous
9     }
```