

# Typage et Analyse Statique

## Cours 6

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel  
Master mention Informatique  
Sorbonne Université

année 2023-2024

# Plan du cours

- ▶ génériques en Java
  - ▶ passage de Java 1.4 à 1.5
  - ▶ classes et méthodes paramétrées
  - ▶ typage et sous-typage
  - ▶ polymorphisme borné
- ▶ *lambda*-expressions en Java 1.8
- ▶ fusion des modèles
  - ▶ fonctionnel et objet en Scala
  - ▶ comparaison avec OCaml

# 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, OCaml, 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
7 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");
          ^
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  }
```

# Implantation

## 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  
mais nécessite des transtypages sûrs

⇒ en quoi est-ce différent du polymorphisme paramétrique d'OCaml ?

# Limitations

- ▶ paramètre de type instancié par une classe ou une interface  
pas par un type primitif  
⇒ auto-boxing (voir cours 5)
  
- ▶ 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.

## 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

OK en 1.8

## 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

## types paramétrés :

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

## tableaux :

- ▶ covariance pour les tableaux :  
 $\tau_2[] \leq \tau_1[]$  ssi  $\tau_2 \leq \tau_1$

# 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;
```

## Typage et sous-typage (3)

- ▶ pas de création de tableaux paramétrés : erreur

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

- ▶ transtypage de tableau

```
1 public class CE<T> {  
2     T[] elts;  
3     public CE(){elts = (T[]) new Object[10];}  
4 }
```

```
$ javac -Xlint CE.java
```

```
CE.java:3: warning: [unchecked] unchecked cast  
    public CE(){elts = (T[]) new Object[10];}  
                          ^
```

```
    required: T[]
```

```
    found:    Object[]
```

```
where T is a type-variable:
```

```
    T extends Object declared in class CE
```

```
1 warning
```

# 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'execution
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:3: 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
int m(ArrayList<String> x) {return x.get(0).length();}
    ^
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  
pour la contra-variance

<?> 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
  - ▶ utilisation de méthodes qui ne prennent pas de paramètres modifiables du type du paramètre de type mais peuvent en retourner
- ▶ 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
  - ▶ utilisation de méthodes qui retournent pas de paramètres modifiables du type du paramètre de type mais peuvent en prendre

# Exemples

## exemple dans Collections:

```
1 public static <T> void copy(List<? super T> dest,  
2                             List<? extends T> src) {  
3     for(T elt : src){dest.add(elt);}  
4 }
```

- ▶ *src* est en lecture (co-variance comme pour le type du résultat pour un type fonctionnel) : utilisation de `T get(int)`
- ▶ *dst* est en écriture (contra-variance comme pour le type du paramètre dans un type fonctionnel), utilisation de `boolean add(T)`

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        // e.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 ( $\langle T \rangle$ ) :
  - ▶ 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*
- ▶ 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/>
- ▶ tutorial GJ :  
<http://www.cis.unisa.edu.au:80/~pizza/gj/Documents/>
- ▶ tutorial Java 1.8 :  
<https://docs.oracle.com/javase/tutorial/extra/generics/index.html>
- ▶ tutorial Pizza :  
<http://pizzacompiler.sourceforge.net/doc/tutorial.html>
- ▶ sur .NET et les generics C# :  
<http://research.microsoft.com/projects/clrgen/>

# Exemple génériques en Java (1)

retour sur la méthode copy :

```
1  class Test {
2
3      static <T> void copy( List<? extends T> src,
4                          List<? super T> dst) {
5          for (T element : src) { dst.add(element); }
6      }
7
8      public static void main(String[]a) {
9          List<Integer> li = new ArrayList<Integer>(10);
10         li.add(10); li.add(20);
11         List<Object> lo = new ArrayList<>();
12         lo.add("Hello");
13         // ok
14         copy(lo,lo);
15         copy(li,li);
16         copy(li,lo);
17         // ...
```

## Exemple génériques en Java (2)

```
1 // 18: pas ok
2 // 19:    copy (lo,li);
3     }
4 }
```

```
1 Test.java:19: error: method copy in class Test cannot be applied
2   to given types;
3     copy (lo,li);
4     ^
5   required: List<? extends T>,List<? super T>
6   found: List<Object>,List<Integer>
7   reason: inferred type does not conform to upper bound(s)
8     inferred: Object
9     upper bound(s): Integer, Object
10  where T is a type-variable:
11    T extends Object declared in method <T>copy(List<? extends T>,List<? ←
12    super T>)
1 error
```

# Introduction des Lambda-expressions

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

- ▶ des lambda-expressions (notées  $\lambda$ -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'rifier si un e'tudiant
4 && (et.getAge() <= 23) // a entre 16 et 23 ans
```

```
1 (Etudiant et) -> {
2   int age = et.getAge();
3   System.out.println(age);
4   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  $\lambda$ -expression, qui devra 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  $\lambda$ -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 la place 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 donnés, 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 mousePressed(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) ->
3     getGraphics().drawString("salut le monde",e.getX(),e.getY());
4 addMouseListener(clic) ;
```

## Composition de calculs et de l'ordre supérieur

On peut ainsi définir de nouveaux itérateurs basés 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  $g(f\ x)$

```
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 $\lambda$ -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 possèdent des méthodes utilisant des  $\lambda$ -expressions comme map, filter et reduce.

```
1 T reduce(T identity,  
2     BinaryOperator<T> accumulator)
```

## 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.equals(""))) // sauf les ""
8         .reduce("", (res, n) -> res + ", " + n) ; // les concatener tous
9 }
```

# le langage Scala

- ▶ “Scalable language” : langage pouvant passer à l'échelle mêmes concepts pour les composants de différentes tailles
- ▶ cherche à unifier et généraliser les concepts de la programmation fonctionnelle et objets
- ▶ introduit des mixins pour la modularité
- ▶ typé statiquement (ADT, GADT, ...)
- ▶ développé à l'EPFL (Lausanne)
- ▶ cible la JVM (interopérabilité avec Java) ou .NET

ces transparents s'inspirent de la présentation de Scala à POPL 2006 (Ordesky) et du cours de programmation avancée de Giuseppe Castagna (lui même s'inspirant de DA-OC - O'Reilly 2000).

# Complémentarité des modèles fonctionnel et objets

Unification programmation fonctionnelle et objet pour le passage à l'échelle :

- ▶ programmation fonctionnelle : grain fin, et il est facile de construire de vrais programmes concis grâce aux
  - ▶ fonctions d'ordre supérieur
  - ▶ types algébriques (types sommes) et filtrage par motifs
  - ▶ polymorphisme paramétrique
- ▶ programmation objet : gros grain, mais il est facile de spécialiser et d'étendre des systèmes complexes avec
  - ▶ héritage, sous-typage et liaison tardive
  - ▶ polymorphisme de sous-typage
  - ▶ classes comme abstractions partielles

## Scala (2)

Scala est objet et fonctionnel

- ▶ avec un modèle objet uniforme :  
tout est objet, y compris les valeurs fonctionnelles
- ▶ avec du filtrage par motif
- ▶ des fonctions d'ordre supérieur
- ▶ des nouveaux "traits" pour abstraire et composer les programmes

## Classes et traits

- ▶ traits (comme une interface Java avec possibilité de définir des méthodes)

```
1 trait Vector {  
2   def norm() : Double  
3   def isOrigin (): Boolean = (this.norm == 0)  
4 }
```

- ▶ classes

```
1 class Point(a: Int, b: Int) extends Vector {  
2   var x: Int = a  
3   var y: Int = b  
4   def norm(): Double = Math.sqrt(Math.pow(x,2) + Math.pow(y,2))  
5   def erase(): Point = { x = 0; y = 0; return this }  
6   def move(dx: Int): Point = new Point(x+dx,y)
```

- ▶ exécution

```
1 scala> new Point(1,1).isOrigin  
2 res0: Boolean = false
```

# Equivalence

## ▶ points avec coordonnées cartésiennes

```
1 class Point(a: Int, b: Int) {  
2   var x: Int = a  
3   var y: Int = b  
4   def norm(): Double = Math.sqrt(Math.pow(x,2) + Math.pow(y,2))  
5   def erase(): Point = { x = 0; y = 0; return this }  
6   def move(dx: Int): Point = new Point(x+dx,y)  
7   def isOrigin(): Boolean = (this.norm() == 0)  
8 }
```

## ▶ points avec coordonnées polaires

```
1 class PolarPoint(norm: Double, theta: Double) extends Vector {  
2   var norm: Double = norm  
3   var theta: Double = theta  
4   def norm(): Double = return norm  
5   def erase(): PolarPoint = { norm = 0 ; return this }  
6 }
```

Là où un objet de type Vector est attendu, on pourra utiliser une instance de PolarPoint ou de la première définition de Point mais pas de la seconde.

# Héritage : Point et ColPoint (1)

## ► déclaration Point :

```
1  class Point(a: Int, b: Int) {  
2      var x: Int = a  
3      var y: Int = b  
4  
5      def norm(): Double = Math.sqrt(Math.pow(x,2) + Math.pow(y,2))  
6      def erase(): Point = { x = 0; y = 0; return this }  
7      def move(dx: Int): Point = new Point(x+dx,y)  
8      def isOrigin(): Boolean = (this.norm() == 0)  
9  }
```

## Héritage : Point et ColPoint (2)

### ► héritage :

```
1  class ColPoint(u: Int, v: Int, c: String) extends Point(u, v) {
2  val color: String = c // variable d'instance non-mutable
3  def isWhite(): Boolean = c == "white"
4
5  override def norm(): Double = {
6    if (this.isWhite()) return 0
7    else return Math.sqrt(Math.pow(x,2)+Math.pow(y,2)) }
8
9  override def move(dx: Int): ColPoint=new ColPoint(x+dx,y,"red")
10 }
```

### ► ajout de isWhite, héritage de erase, isOrigin, redéfinition de move, norm.

```
1  scala> new ColPoint( 1, 1, "white").isOrigin()
2  val res3: Boolean = true
```

la liaison tardive de norm dans isOrigin

# Classe abstraite

type pour les entiers naturels :

▶ en OCaml

```
1   type nat = Zero | Succ of nat
```

▶ en Scala

```
1   abstract class Nat {  
2     def isZero : Boolean  
3     def pred : Nat  
4     def succ : Nat = new Succ(this)  
5     def + (x : Nat): Nat = if (x.isZero) this else succ + x.pred  
6     def - (x : Nat): Nat = if (x.isZero) this else pred - x.pred  
7   }  
8   class Succ(n: Nat) extends Nat {  
9     def isZero : Boolean = false;  
10    def pred : Nat = n  
11  }  
12  object Zero extends Nat {  
13    def isZero : Boolean = true;  
14    def pred : Nat = throw new Error("error : Zero.pred")  
15  }
```

## Fonctions d'ordre supérieur

- ▶ les fonctions sont des valeurs
- ▶ les fonctions peuvent être anonymes, curriifiées, imbriquées
- ▶ peuvent être implantées comme des méthodes de classes Scala :

```
1 matrix exists (row => row forall (0 ==))
```

- ▶ `matrix` pourrait être de type `List[List[int]]` en utilisant la classe `List` :

```
1 class List[+T] {  
2   def isEmpty : boolean;  
3   def head: T  
4   def tail : List[T]  
5   def exists(p : T => Boolean): Boolean =  
6     !isEmpty && (p(head) || (tail exists p))  
7     ...  
8 }
```

## Scala(3)

L'idée est d'unifier les concepts de programmation fonctionnelle et de programmation par objets

1. Types algébriques (types sommes ou ADT) avec les hiérarchies de classes + filtrage par motifs
2. Fonctions avec les objets
3. Modules avec les objets

# 1ère unification : ADT et filtrage par motifs

La plupart des langages fonctionnels ont des types de données algébriques et du filtrage par motifs

⇒ manipulation concise et canonique des structures de données

Critiques du monde objet :

- ▶ les "ADT" ne sont pas extensibles
- ▶ cela va à l'encontre du modèle de données objets
- ▶ le filtrage casse l'encapsulation

néanmoins en OCaml les types sommes peuvent être extensibles  
( type  $t = ..+$  mais on perd la détection de l'exhaustivité du filtrage,

# Filtrage par motifs en Scala

Introduction des "case class" indiquant une classe pouvant apparaître dans un motif.

- ▶ modèle uniforme : un "ADT" sera aussi une classe (classique)
- ▶ un motif permet d'accéder aux paramètres du constructeur de la classe
- ▶ peut aussi s'appliquer directement à une déclaration objet (qui doit alors être sérialisable).

## Exemple 1 : évaluateur d'expressions

### ▶ classe abstraite : Expression

```
1 abstract class Expression
2 case object X extends Expression
3 case class Const(value: Int) extends Expression
4 case class Add(left: Expression, right: Expression) extends Expression
5 case class Mult(left: Expression, right: Expression) extends Expression
6 case class Neg(expr: Expression) extends Expression
```

### ▶ évaluation d'une expression avec un environnement

```
1 def eval(expression: Expression, xValue: Int): Int = expression match{
2   case X => xValue
3   case Const(cst) => cst
4   case Add(left, right) => eval(left, xValue) + eval(right, xValue)
5   case Mult(left, right) => eval(left, xValue) * eval(right, xValue)
6   case Neg(expr) => - eval(expr, xValue) }
```

### ▶ exécution

```
1 val expr = Add(Const(1), Mult(Const(2), Mult(X, X)))
2 assert(eval(expr, 3) == 19)
```

exemple extrait du “Kerflyn’s Blog”

## Exemple 2 : évaluateur de termes avec GADT (1)

définition des types avec GADT

- ▶ **But:** préciser le typage sur les paramètres de types
  - ▶ les contraintes sur les paramètres de type peuvent changer en fonction des constructeurs
  - ▶ les variables de types sont :
    - ▶ existentielles quand elles sont en position argument d'un constructeur
- ▶ type en Ocaml :

```
1 type 'a term =  
2   | Lit : int -> int term  
3   | Succ : int -> int term  
4   | IsZero : int term -> bool term  
5   | If : bool term * 'a term * 'a term -> 'a term
```

- ▶ code en OCaml :

```
1 let rec eval : type a. a expr -> a = function e -> match e with  
2   | Lit i -> i  
3   | Succ n -> (eval n) + 1  
4   | IsZero t -> eval t == 0  
5   | If (c,t1,t2) -> if (eval c) then (eval t1) else (eval t2)
```

## Exemple 2 : évaluateur de termes avec GADT (2)

► classes en Scala :

```
1 class Term [T]
2 case class Lit (x : int) extends Term[int]
3 case class Succ (t : Term[int]) extends Term[int]
4 case class IsZero (t : Term[int]) extends Term[Boolean]
5 case class If [T] (c : Term[Boolean],
6                   t1 : Term[T],
7                   t2 : Term[T]) extends Term[T]
```

le modifieur de classe case autorise ensuite le filtrage

► code en Scala :

```
1 def eval[T](t : Term[T]): T = t match {
2   case Lit(n) => n
3   case Succ(u) => eval(u) + 1
4   case IsZero(u) => eval(u) == 0
5   case If(c, t1, t2) => eval(if(eval(c)) t1 else t2)
```

## 2ème unification : fonctions sont des objets

- ▶ Si les fonctions sont des valeurs et les valeurs des objets alors les fonctions sont des objets
- ▶ le type fonctionnel  $S \Rightarrow T$  est équivalent à `scala.Function1[S,T]` où `Function1` est une classe abstraite :

```
1 abstract class Function1[-S,+T]{def apply(x:S):T}
```

- ▶ les fonctions sont vues comme des objets avec une méthode `apply`
- ▶ la fonction anonyme "incrément"

```
1 x: int => x + 1
```

correspond à :

```
1 new Function1[int,int] { def apply(x:int): int = x + 1}
```

le type de la classe abstraite `Function1` est co-variant sur le résultat et contra-variant sur le paramètre.

# Spécialisation de fonctions

Parce que  $\Rightarrow$  (définition de fonction) est une classe, elle peut être spécialisée dans une sous-classe. On obtient la spécialisation de fonctions.

par exemple sur les tableaux :

- ▶ fonctions mutables sur des intervalles d'entiers

```
1 class Array[A](length : int) extends (int => A) {  
2   def length : int = ...  
3   def apply(i : int): A = ...  
4   def update(i : int, x : A): unit = ...  
5   def elements : Iterator[A] = ...  
6 }
```

- ▶ appels :

```
1 a.update(i, a.apply(i) * 2) // pour a(i) = a(i) * 2
```

# Fonctions partielles

On trouve des fonctions définies seulement sur une partie de leur domaine.

- ▶ il est possible de demander à la méthode `isDefinedAt` si une fonction partielle est définie pour une valeur donnée

```
1  abstract class PartialFunction[-A, +B] extends (A => B) {  
2      def isDefinedAt(x : A): Boolean  
3  }
```

- ▶ le filtrage par motif est considéré comme une fonction, comme il peut avoir une extension des cas, l'exhaustivité du filtrage ne peut pas être vérifiée, et donc le bloc du filtrage est considéré comme une fonction partielle.
- ▶ on retrouve les variance habituelles : contra-variant sur le paramètre et co-variant sur le résultat

# Composant

- ▶ un composant est une partie de programme, à combiner avec d'autres parties dans des applications plus importantes.
- ▶ *Exigence* : Les composants doivent être réutilisables.
- ▶ Pour être réutilisable dans de nouveaux contextes, un composant a besoin d'interfaces en décrivant les services fournis ainsi que les services requis.

Un composant doit faire référence à d'autres composants non pas par des liens en dur, mais uniquement par ses interfaces requises.

Une autre façon d'exprimer cela est : Toutes les références d'un composant à d'autres composants doivent se faire via ses membres ou ses paramètres.

En particulier, il ne doit pas y avoir de données ou de méthodes statiques globales auxquelles les autres composants accèdent directement.

# Foncteurs et abstractions

|                 |                           |
|-----------------|---------------------------|
| Composant       | ≡ Foncteurs ou Structure  |
| Interface       | ≡ Signature               |
| Composant exigé | ≡ paramètre du foncteur   |
| Composition     | ≡ application du foncteur |

En Ocaml :

```
1 type comparison = Less | Equal | Greater;;  
2 module type ORDERED_TYPE = sig  
3   type t  
4   val compare: t -> t -> comparison  
5   end;;
```

# Foncteur Set

```
1 module Set (Elt: ORDERED_TYPE) = struct
2   type element = Elt.t type set = element list let empty = []
3   let rec add x s =
4     match s with
5     [] -> [x]
6     | hd::tl ->
7       match Elt.compare x hd with
8       Equal -> s
9       | Less -> x :: s
10      | Greater -> hd :: add x tl
11   let rec member x s =
12     match s with [] -> false
13     | hd::tl ->
14       match Elt.compare x hd with
15       Equal -> true (* x belongs to s *)
16       | Less -> false (* x is smaller than all elmts of s *)
17       | Greater -> member x tl
18   end ;;
```

## signature inférée

La signature inférée est :

```
1   module Set :  
2   functor (Elt : ORDERED_TYPE) ->  
3   sig  
4   type element = Elt.t  
5   type set = element list  
6   val empty : 'a list  
7   val add : Elt.t -> Elt.t list -> Elt.t list val member : Elt.t -> Elt.t list ↔  
      -> bool  
8   end
```

# application d'un foncteur

## ▶ Chaînes ordonnées

```
1  module OrderedString =
2    struct
3      type t = string
4      let compare x y =
5          if x = y then Equal
6          else if x < y then Less else Greater end;;
7  module OrderedString : sig type t = string val compare : 'a -> 'a ->  $\leftrightarrow$ 
    comparison end
```

## ▶ Ensemble de chaînes ordonnées

```
1  module StringSet = Set(OrderedString);;
2  module StringSet : sig
3    type element = OrderedString.t
4    type set = element list
5    val empty : 'a list
6    val add : OrderedString.t -> OrderedString.t list -> OrderedString val  $\leftrightarrow$ 
    member : OrderedString.t -> OrderedString.t list -> bool
7  end
8  # StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
9  - : bool = false
```

## Construction pour composants

Scala a trois concepts qui sont particulièrement intéressants pour les systèmes de composants.

- ▶ les membres de type abstrait permettent d'abstraire sur des types qui sont membres d'objets.
- ▶ les annotations de type "self-type" permettent d'abstraire sur le type de "self".
- ▶ la composition de mixin offre une manière flexible de composer les composants et les types de composants.

## 3ème unification : Modules are Objects

En Scala :

Composant           ≡ Classe

Interface           ≡ Trait - Classe abstraite

Composant exigé   ≡ membre abstrait (ou "Self")

Composition       ≡ Composition mixin

## Composition de mixins (1)

- ▶ classe abstraite (pas d'instanciation), peut interopérer avec Java

```
1 abstract class AbsIterator {  
2   type T // type opaque comme dans les modules OCaml  
3   def hasNext: Boolean  
4   def next: T  
5 }
```

- ▶ trait, peut contenir des implantations de méthodes, ne peut pas interopérer avec Java

```
1 trait RichIterator extends AbsIterator {  
2   def foreach(f: T => Unit) { while (hasNext) f(next) } // higher-order  
3 }
```

- ▶ un itérateur concret :

```
1 class StringIterator(s: String) extends AbsIterator { type T = Char  
2   private var i = 0  
3   def hasNext = i < s.length()  
4   def next = { val ch = s.charAt i; i += 1; ch }  
5 }
```

## Composition de mixins (2)

Il n'est pas possible de combiner `StringIterator` et `RichIterator` dans une simple classe en utilisant uniquement l'héritage simple, les 2 contiennent des implantations de méthodes.

- ▶ composition de mixin (mot-clé `with`) : réutilise la différence de la définition de classes (toutes les méthodes non-héritées).

```
1 object StringIteratorTest {
2   def main(args: Array[String]) {
3     class Iter extends StringIterator(args(0)) with RichIterator //mixin
4     val iter = new Iter
5     iter.foreach(println) } }
```

qui étend la classe ancêtre `StringIterator` avec les méthodes de `RichIterator` qui ne sont pas héritées de `AbsIterator` comme `foreach` mais pas `next` ou `hasNext`.

- ▶ exécution :

```
1 $ scala StringIteratorTest "bye"
2 b
3 y
4 e
```

- ▶ les mixins sont une jointure de la relation d'héritage?

# Abstraction de composants

Il existe deux formes principales d'abstraction dans les langages de programmation :

- ▶ la paramétrisation (en fonctionnel)
- ▶ les membres abstraits (en orientée objet)

Scala supporte les deux styles d'abstraction pour les types aussi bien que pour les valeurs

Les types et les valeurs peuvent être paramétrés, , et les deux peuvent être des membres abstraits.

Scala fonctionne avec la dualité fonctionnelle/OO en ce sens que le paramétrage peut être exprimé par des membres abstraits