PF
Programmation Fonctionnelle

Legond-Aubry Fabrice

fabrice.legond-aubry@parisnanterre.fr

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Fonctions

Lambda Calculs

Streams

Compléments

Définitions (Hors Langage)

Exemple

Plan du Cours

Programmation Fonctionnelle – WTF ?

Rappels sur les Génériques (et autres)

Fonctions

Lambda Calculs

Streams

Compléments

Définitions (Hors Langage)

Exemple

Rappels sur les Génériques (et autres)

- Les génériques : Ou comment introduire de la PF dans le POO Java
  - ✓ Une nécessité pour renforcer le typage en java
  - ✓ Un prérequis pour la PF en Java
- « Génériques » (generics) : Définition
  - ✓ La base de la programmation générique
    - ➢ C'est l'écriture de code qui peut être utiliser par différents type d'objet
    - ➢ Ces objets pas forcement liée dans leur type !!! (p.ex. pas de relation de sous typage)
    - ➢ Utilisation de gabarits (template) de classe/code
  - ✓ Ajouté dans java 5 avec le typage des collections
  - ✓ Permet de renforcer le typage et les vérifications à la compilation
  - ✓ Permet de coder plus efficacement en limitant la sur-expressivité des « cast »
  - ✓ Sans : Erreurs apparaissent à l'exécution (DANGER !!!)

Rappels sur les Génériques (et autres)

- En C, on nomme cela du « template programming »
  - ✓ On génère des templates (patrons/modèle/gabarit) de codes
    - ➢ Que l'on peut placer ensuite dans différentes classes
    - ➢ Que l'on peut appliquer directement dans des méthodes
- Il existe des "generics" pour les classes, les interfaces, les constructeurs et les méthodes
  - ✓ Un peu comme les fonctions de la PF

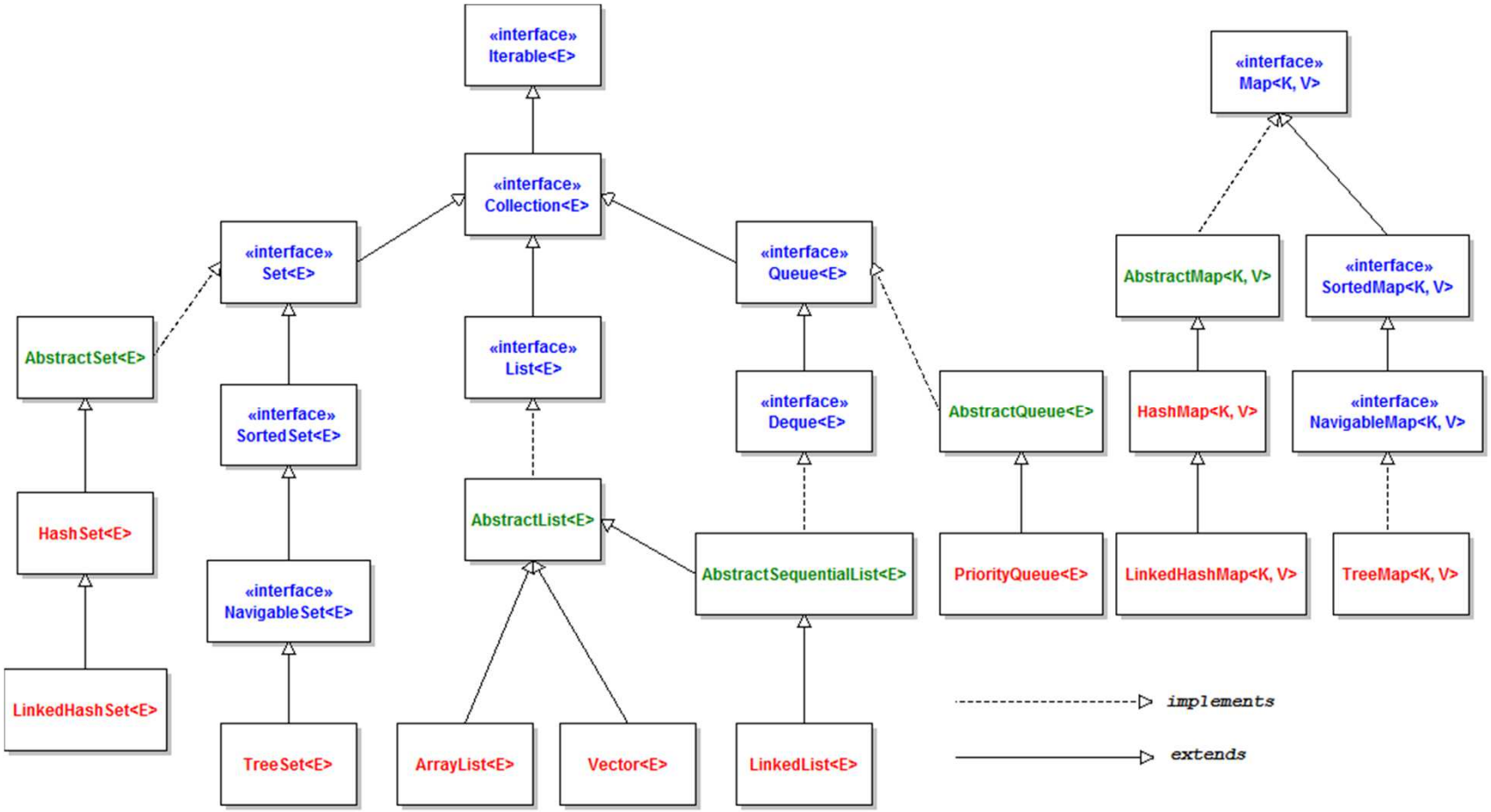
Rappels sur les Génériques (et autres)

LES GENERIQUES PAR LA PETITE PORTE DES COLLECTIONS

- Lorsque l'on crée des listes (ArrayList, …)
  - ✓ On applique les mêmes fonctions quelque soit la nature de la « liste »
  - ✓ On applique les mêmes fonctions quelque soit la nature des données stockées
  - ✓ Les données peuvent être homogènes ou non
  - ✓ Nécessité de typage dynamique
  - ✓ Nécessité de contrôle de type lors de **l'insertion ou l'extraction**
- https://www.codejava.net/java-core/collections/overview-of-java-collections-framework-api-uml-diagram
- Problèmes lorsque les données sont homogènes
  - ✓ complexité du code pour un problème simple
  - ✓ Génération possible de beaucoup d'effets de bord
- On utilise les types génériques
- On utilise aussi le Un/Boxing : p.ex. int VS Integer
  - ✓ encapsulation des types basiques dans des objets pour les intégrés aux génériques
  - ✓ Gestion de l'immutabilité

Rappels sur les Génériques (et autres)

LES GENERIQUES PAR LA PETITE PORTE DES COLLECTIONS



Rappels sur les Génériques (et autres)

LES GENERIQUES PAR LA PETITE PORTE DES COLLECTIONS

Avant les generics

- Exemple :

```
ArrayList notes = new ArrayList();
// Est-ce vraiment un Integer ?
// Nécessité du cast ! A l'execution !
Integer note = (Integer) data.get(0);
List ecs = new ArrayList();
ecs.add(new Matiere("PROF"));
// Nécessité du cast ! A l'execution !
Matiere m = (Matiere) ecs.get(0);
```

- Il n'y a pas de vérification statique du typage possible pour certaines opérations (Collection)
- Tester le type de retour du get est lourd !!!
  - ✓ Try/catch ClassCastException, instanceof, …

Après les generics

- Exemple :

```
ArrayList<Integer> notes = new ArrayList<>();
// Plus de cast. Vérifications à la compilation
Integer notes = data.get(0);
// Liste typée et homogène
List<Matiere> ecs = new ArrayList<>();
ecs.add(new Matiere("PROF"));
Matière m = ecs.get(0);
```

- Solution avec « génériques »:
  - ✓ Le compilateur peut faire des vérifications de type (« type safety »)
  - ✓ Plus sécurisé , plus simple à lire

Rappels sur les Génériques (et autres)

LES GENERIQUES PAR LA PETITE PORTE DES COLLECTIONS

- Exemple Boxing:

```
public static int sum (List<Integer> ints) {
        int s = 0;
        for (int i=0; i<ints.size(); i++ ) { s += ints.get(i); }
        return s;
}
// 60% plus lent, Integer immutable, création d'objets
public static Integer sum (List<Integer> ints) {
        Integer s = 0;
        for (Integer num : ints ) { s += num; }
        return s;
}
public static int sum (List<Integer> ints) {
        int s = 0;
        for (int num : ints ) { s += num; }
        return s;
        //en changeant le type de retour il est possible de faire « return new Integer(s) »
}
```

Rappels sur les Génériques (et autres)

CLASSES / INTERFACES GENERIQUES

- “Une classe [interface] est générique si elle déclare une ou plusieurs variables génériques (non typée)” (JLS 8.1.2)

  JLS : https://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html

- Utilisation de variables muettes (ie au sens mathématique)
  - ✓ Nommées Variable de type (Type Variables)
  - ✓ Nommées aussi Variable Paramétrée
  - ✓ Nommées aussi Variable Générique

{ClassModifier} **class** Identifier **[TypeParameters]** [Superclass] [Superinterfaces]

{InterfaceModifier} **interface** Identifier **[TypeParameters]** [ExtendsInterfaces]

TypeParameter : < TypeParameterList >

Rappels sur les Génériques (et autres)

CLASSES / INTERFACES GENERIQUES

- Les variables de Type qui seront utilisées dans la classe sont placées après le nom de la classe et placées entre <>
  - ✓ Il est courant d'utiliser des lettres majuscules
  - ✓ On utilise en général une seule lettre mais il s'agit d'un identifiant java classique.
- Il est ensuite possible de déclarer des variables Java avec un type générique comme type
  - ✓ Les variables de Type peuvent être utilisées tout au long de la définition de la classe

Rappels sur les Génériques (et autres)

CLASSES / INTERFACES GENERIQUES

- Attention il est interdit de déclarer une variable de type générique :
  - ✓ Comme variable membre « static » dans une classe ou une classe incluse
  - ✓ dans un initialiseur statique ou un initialiseur statique d'une classe incluse
- Une variable de type est « instanciée » en substituant le type générique ("muet") par un autre type existant (non générique)
  - ✓ Une classe générique ne peut être substitué par Throwable ou une de ses sous classes
  - ✓ Les classes avec des types génériques agissent comme une "factory" à la compilation

Rappels sur les Génériques (et autres)

CLASSES / INTERFACES GENERIQUES

- Lorsqu'on ne substitue pas dans le code, on utilise <>
  - ✓ <> est appelé le marqueur **DIAMANT**
- Choisir de ne pas substituer les types paramétrés :
  - ✓ Ils resteront « génériques » ou « **RAW** »
  - ✓ Lève un warning à l'édition (selon IDE) et à la compilation (sauf si masqué)
  - ✓ Typage à l'exécution → DANGEREUX !!! (effets de bords).
  - ✓ BREF A EVITER. (ANTI-PATTERN)

Rappels sur les Génériques (et autres)

CLASSES / INTERFACES GENERIQUES

- Instanciation (type arguments) :

```
interface Nom<T1,T2,…,Tn> { … }
class Nom<T1,T2,…,Tn> {

        …
        … {
        T1 a1;
        T2 methode(T3 x) { return f(a1,x); }
        }
}
```

Rappels sur les Génériques (et autres)

CLASSES / INTERFACES GENERIQUES

- Exemple :

```
public interface Arbre<T>
        { int size(); }
public class Feuille<T> implements Arbre<T> {
  protected final T val;
  public Feuille(T val)
                { this.val = val; }
  @Override public int size()
                { return 1; }
}
public class Noeud<T> implements Arbre<T> {
        protected Arbre<T> g,d;
        public Noeud(Arbre<T> g, Arbre<T> d)
                { this.g = g; this.d = d; }
        @Override public int size()
                { return g.size() + d.size(); }
  }
```

Rappels sur les Génériques (et autres)

CLASSES / INTERFACES GENERIQUES

- Instanciation (type arguments - JLS 4.5) :

```
Nom<TC1,TC2,...,TCn> var = new Nom<TCp1,TCp2,...,TCpn>(...);
// les TCi et TCpi sont des types concrets
// on peut utiliser new Nom<>(...) si on peut inférer les types
```

- Exemple d'utilisation

```
Arbre<Integer> f1 = new Feuille<>(1);
Arbre<Integer> f2 = new Feuille<>(2);
Arbre<Integer> n  = new Noeud<>(f1, f2);
System.out.println(n.size());
// Ci-dessous : Pas de vérification de typage possible à la compilation
// DANGEREUX !!
Arbre nraw = new Noeud(f1,f2);
```

Rappels sur les Génériques (et autres)

METHODES GENERIQUES

- Une méthode simple peut être définie comme une méthode générique
  - ✓ Une méthode générique est une méthode qui utilise un type générique
  - ✓ Le type paramétré peut être utilisé en type de retour ou en paramètre
  - ✓ Une méthode générique peut être utilisé à la fois dans une classe générique ou normal
  - ✓ Une méthode statique générique est autorisée
- Déclaration :

  {MethodModifier} **TypeParameters** {Annotation} Result MethodDeclarator [Throws]
- Vous pouvez omettre à certains moments le type d'un type générique et laisser faire le compilateur
  - ✓ L'inférence : C'est parfois risqué si vous ne maîtrisez pas le typage et ses arcanes
  - ✓ Peut conduire à des ambiguïté
  - ✓ Parfois la sur-expressivité à du bon !!!

Rappels sur les Génériques (et autres)

METHODES GENERIQUES

- Exemple : Renvoie quoi ? Integer ? Double ? → Number !!!

```
public static <T> T getMiddle(T... a) { /* ... */ }
...
double middle = getMiddle(3.14, 1729, 0);
```

- une méthode qui introduit ses paramètres de type

```
public static <T> String repr(Collection<T> es) {
    StringBuilder rtr = new StringBuilder();
    for (T element : es)
            { rtr.append(element.toString()); }
    return String.format("[%s]", rtr.toString());
}
```

- Utilisation avec ou sans inférence de type

```
Arbre<Matiere> n = new Noeud<>(f1, f2);
String  reprComplique = App.<Matiere>repr(n.elements());
String  reprSimple = repr(n.elements());
// avec List<T> elements() { ... } dans Arbre<T>/Noeud<T>/Feuille<T>
```

Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE

- Par défaut, on peut substituer un type générique par :
  - ✓ tout type dérivant de Object
  - ✓ Sauf les classes dérivant de Throwable
- Il est possible de restreindre les substitutions des types génériques
- Cela permet de limiter ce qui peut instancier un paramètre de type
  - ✓ typiquement en fonction de méthodes accessibles
- On peut définir une limite (bound) sur un type paramétré // JLS 4.4
  - ✓ Défini par le mot clef « extends »
  - ✓ Le type après le mot clef « extends » peut aussi être paramétré
  - ✓ La substitution est limitée à toutes les sous classes du type données après le mot clef

Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE

- Exemple :

```
// T peut être toute sous classe de Number (Number est la classe limite)
public class Calculator<T extends Number> { ... }
// T doit implémenter Comparable<T>
//exemple de méthode statique
public static <T extends Comparable<T>> int nbGT(T[] es, T val) {
  int nb = 0;
  for (T e : es) { if (e.compareTo(val) > 0) { nb++; } }
  return nb;
}
```

Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE

- Exemple:

```
public interface Descriptif
    { String desc(); }
public class Matiere
    { ... } // pas de méthode desc
public class MatiereAvecDescriptif implements Descriptif
    { ... } // méthode desc à définir
public static <T extends Descriptif> String reprDesc(Collection<T> es) {
   StringBuilder rtr = new StringBuilder();
   for (T element : es)
                { rtr.append(element.desc()); }
   return String.format("[%s]", rtr.toString());
}
```

Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE

- Exemple suite:

```
MatiereAvecDescriptif mad  = new MatiereAvecDescriptif
    (« module de PF »);
Matiere   m   = new Matiere (« module P12");
// autre possibilité: List<Descriptif>     listD = List.of(mad);
List<MatiereAvecDescriptif> listMAD = List.of(mad)
List<Matiere>   listM  = List.of(m);
String ok  = reprDesc(listMAD); // OK
String ko  = reprDesc(listM);  // KO
```

Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE

- il est possible d'indiquer plusieurs bornes
  - ✓ ordre : une ou zéro classe puis zéro, une ou plusieurs interfaces
  - ✓ Séparés par une éperluette (**&**)
  - ✓ les bornes peuvent aussi être génériques
  - ✓ T doit être une sous classe de tous les types après « extends »
- Exemple :

```
class     C  { ... }
interface I1 { ... }
interface I2 { ... }
class     D  <T extends C  & I1 & I2> { ... } // OK
class     E  <T extends I1 & C  & I2> { ... } // KO
```

Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE - JOKER

- Un type générique peut avoir un Joker (Wildcards) : **?**
  - ✓ Il représente un type inconnu
  - ✓ Ce n'est pas tous les types mais « un parmi »
  - ✓ "?" équivalent à "? extends Object"
- Un joker peut avoir une limite simple
- Différence variable générique / joker
  - ✓ Une variable T décrit un type défini équivalent mais inconnu
  - ✓ Un joker décrit un type inconnu (et potentiellement non équivalent)
- Exemple :

```
// Ici une très simple collection de type totalement inconnu
Collection<?> coll = new ArrayList<>();
```

Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE - JOKER

- Un Joker est utile dans des situation où (JLS 4.5.1) :
  - ✓ seulement une connaissance partielle du paramètre générique est requis
  - ✓ Où on se refuse à fixer un des éléments du générique

- Collection<?> est different de Collection<Object>
  - ✓ Object est un type connu

- Exemple :

```
// Joker '?', Collection d'éléments inconnus
public void printAny(Collection<?> c) {
    for (Object o : c) { // This is what we can assume
                System.out.println(o);  // Call Object.toString()
    }
}
```

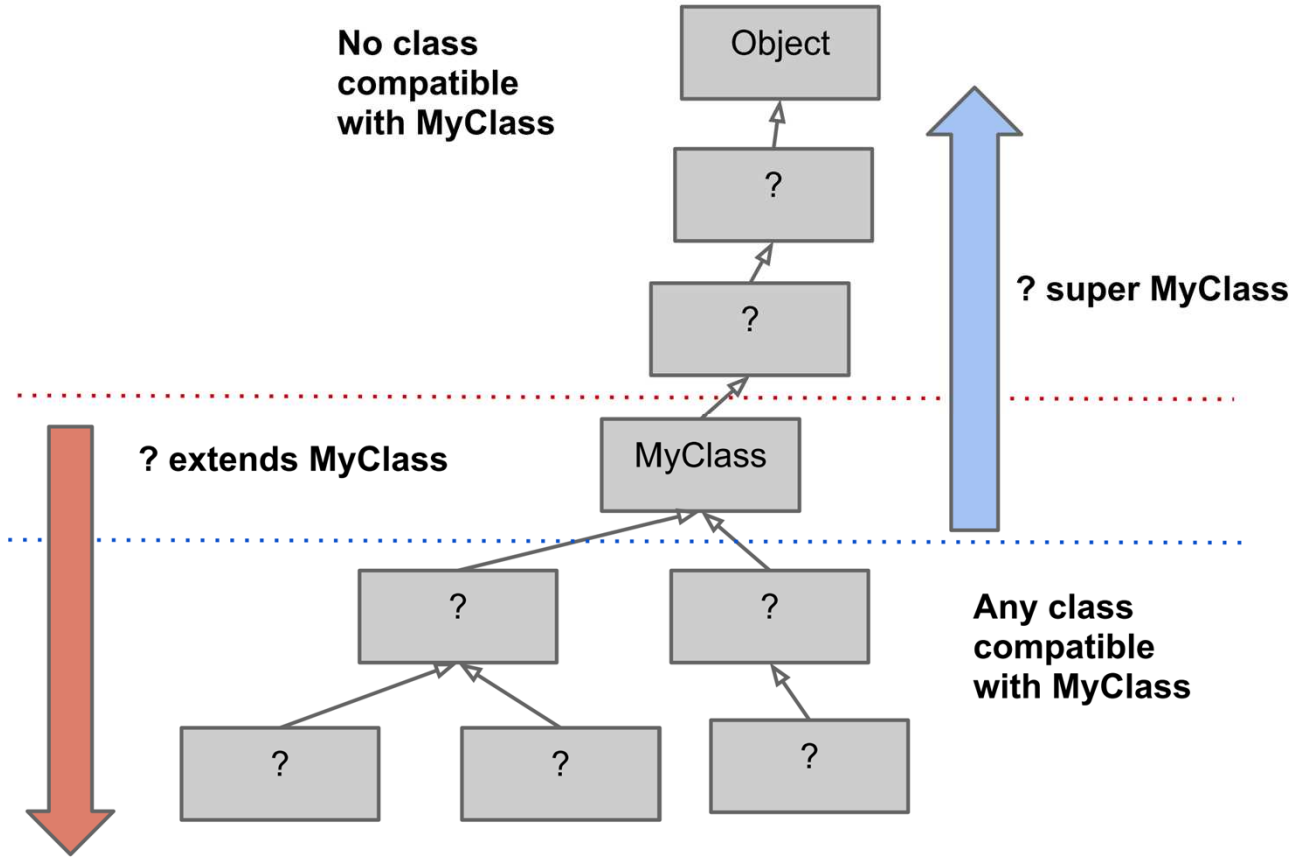
Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE - JOKER

- Un Joker peut avoir deux types de limites simples (en exclusion)
  - ✓ Contrairement au type générique sans joker qui n'a que « extends »
  - ✓ Mot clef "extends" qui à la même signification que précédemment
  - ✓ Mot clef "super"
- "super"
  - ✓ La classe doit être une superclasse de la classe après le mot clef
  - ✓ Assez contre-intuitif
- Exemple :

```
// List peut contenir des éléments de toute sousclasse de Number. Limite Haute.
List<? extends Number> nlist;
// List peut contenir des éléments de toute superclasse de Integer. Limite Basse.
List<? super Integer> ilist;
```

Rappels sur les Génériques (et autres)



Rappels sur les Génériques (et autres)

METHODES – TYPAGE BORNE – JOKER

- Principe du Get/Put (insertion/extraction) :
  - ✓ Utiliser les jokers “extend” quand on sort une donnée d’une structure
  - ✓ Utiliser les jokers “super” quand on insert une donnée dans une structure
  - ✓ Ne pas utiliser de joker quand on insert et extrait

Rappels sur les Génériques (et autres)

EFFACEMENT DU TYPAGE (TYPE ERASURE)

- La JVM (le moteur d'exécution du code java) ne supporte pas les génériques
  - ✓ Les types génériques sont donc effacés par le compilateur Java
  - ✓ Nom de l'opération : « **Type Erasure** »
  - ✓ On ne crée pas de nouvelle classes à partir des types génériques
  - ✓ Les types génériques sont remplacés par leur limite ou par Object (qui est la limite par défaut)
- La substitution à la compilation remplace les variables muettes (types génériques)
  - ✓ Les « cast » sont insérés automatiquement
  - ✓ Des méthodes appelés « bridge methods » sont insérées pour préserver le polymorphisme
- La substitution contrôle la cohérence des type (type safety)
  - ✓ Pour chacun de type générique défini
  - ✓ Pour chacune des utilisations des types génériques

Rappels sur les Génériques (et autres)

TYPE ERASURE

- Les méthodes génériques sont aussi converties
  - ✓ On ne génère pas des familles de méthodes
  - ✓ On utilise les types limites
- Exemple de code :

```
class Tuple <T, U> {        // T et U n'ont pas de limite définie
        private T premier;
        private U second;
        // ...
         public T getPremier () { return premier; }
         public U getSecond () { return second; }
}
…
Tuple <Integer, String> unTuple = new Tuple<>();
Integer first = unePaire.getPremier();
String second = unePaire.getSecond();
```

Rappels sur les Génériques (et autres)

TYPE ERASURE

- **Exemple de la classe après effacement:**

```
// classe générique Tuple<T, U>
class Tuple {
    private Object premier;
    private Object second;
    // ...
}
```

- **Exemple de méthode après effacement:**

```
public static <T extends Comparable> T min(T[] a)
→
// Après effacement
public static Comparable min(Comparable[] a)
```

Rappels sur les Génériques (et autres)

TYPE ERASURE

- **Exemple d'utilisation de la classe après effacement:**

```
Tuple<Personne, Salaire> employe = …
Personne pers = employe.getPremier()
→
Tuple employe = …
Personne pers = (Personne) employe.getPremier()
// Renvoie un Object qu'on "cast".
// Mais la vérification a déjà été faite
// Evitera tout cas de typage incompatible durant l'exécution du code
```

Rappels sur les Génériques (et autres)

TYPE ERASURE

- Convertir les méthodes génériques lors de l'effacement peut imposer la création de méthodes de pont
- Exemple du tutorial Java :

```
public class Node<T> {
   public T data;
   public Node(T data) { this.data = data; }
   public void setData(T data) {
      System.out.println("Node.setData");
      this.data = data;
   }
}
public class MyNode extends Node<Integer> {
   public MyNode(Integer data) { super(data); }
   @Override
   public void setData(Integer data) {
      System.out.println("MyNode.setData");
      super.setData(data);
   }
}
```

Rappels sur les Génériques (et autres)

TYPE ERASURE

- Après effacement, le type de la méthode setData à le mauvais type et ne peut pas “override” la méthode parente :
  - ✓ public void setData (Object data) dans Node VS public void setData(Integer data) dans MyNode
- Exemple

```
public class Node {
   public Object data;  // projection
   public Node(Object data) { this.data = data; }
   public void setData(Object data) {
      System.out.println("Node.setData");
      this.data = data;
   }
}
public class MyNode extends Node {
   public MyNode(Integer data) { super(data); }
   // POLYMORPHISME HS !!! INCOHERENCE AVEC LA METHODE setData de node
   public void setData(Integer data) {
      System.out.println("MyNode.setData");
      super.setData(data);
   }
}
```

Rappels sur les Génériques (et autres)

TYPE ERASURE

- Exemple insertion de la méthode de pont dans la classe

```
public class MyNode extends Node {
    ...
   // méthode bridge setData AJOUTE
   public void setData(Object data) {
      setData((Integer) data);
   }
    ...
}
```

Rappels sur les Génériques (et autres)

LIMITATIONS / RESTRICTIONS / HERITAGES

- On ne peut substituer un paramètre générique avec un type primitif (simple)
  - ✓ Type simple : int, short, …
  - ✓ D'où la nécessité de boxing/unboxing
- On ne peut pas créer des tableaux (type primitif Array) de type génériques
- Jouer avec le varargs et les types génériques est dangereux
  - ✓ De toute façon varargs est le premier pas vers l'enfer …
- On instancie pas des types génériques (pas de new)
- Rappel : Pas de type générique en environnement static
  - ✓ ie pas de variable static générique
- Rappel : une variable générique ne être substitué par un Throwable
  - ✓ On ne peut « Throw » ou « Catch » une instance de classe générique
- Attention aux clashes de méthode après un effacement (bridge method, …)
- Les tests dynamiques de typage ne fonctionne que sur les classes classiques
  - ✓ Classes classiques = Classes non génériques

Rappels sur les Génériques (et autres)

LIMITATIONS / RESTRICTIONS / HERITAGES

- Une classe générique peut « extend » ou « implement » un autre type générique
  - ✓ Par exemple ArrayList<T> implements List<T>
  - ✓ Conséquence ArrayList<Manager> est un sous-type List<Manager>
- Attention aux autres relations de typage avec les génériques
  - ✓ Voir les compléments (covariance / contravariance / invariance)
  - ✓ Plus tard: un type générique en java est invariant
- Il n'y a aucune relation de typage possible entre 2 types génériques
  - ✓ Nécessité pour la vérification du typage (type safety)