Généricité

Université de Nice - Sophia Antipolis

Richard Grin

Version 1.7.1 – 6/3/11

Plan

- Pourquoi la généricité ?
- Présentation de la généricité
- Méthodes génériques
- Instanciation de type générique avec joker
- Paramètres de types contraints
- Implémentation (effacement de type)
- Interopérabilité avec du code non générique

Définition

- Une méthode peut être paramétrée avec des **valeurs**
- La généricité permet de paramétrer du code avec des **types de données**
- Exemple :
  - Classe **`ArrayList<T>`** dont le code est paramétré par un type **`T`**
  - Pour l'utiliser il faut passer un type en argument : **`new ArrayList<Employe>()`**

Pourquoi la généricité ?

Avant le JDK 5

- Une collection d'objets ne contient le plus souvent qu'un seul type d'objet : liste d'employés, liste de livres, liste de chaînes de caractères, etc.
- Mais avant le JDK 5.0 les éléments des collections étaient déclarés de type **`Object`**
- Il était impossible d'indiquer qu'une collection ne contenait qu'un seul type d'objet, comme on le fait avec les tableaux (par exemple **`String[]`**)

Exemple de collection non générique avant le JDK 5

- La classe **`ArrayList`** (implémente une liste en utilisant un tableau) contenait les méthodes :
  - **`boolean add(Object o)`**
  - **`Object get(int i)`**

Exemple d'utilisation de `ArrayList`

```
List employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
. . . // On ajoute d'autres employés
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      ((Employe) employes.get(i)).getNom());
}
```

Que se passe-t-il si on omet ce *cast* ?

Exemple de problème

```
List employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
. . . // On ajoute d'autres employés
// On ajoute un livre au milieu des employés
Livre livre = new Livre(...);
employes.add(livre);
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      ((Employe) employes.get(i)).getNom());
}
```

Ça compile ?
Ça s'exécute ?

Conséquences

- Grande souplesse : les collections pouvait contenir n'importe quel objet (mais pas les types primitifs)
- Mais,
  - il était impossible d'indiquer qu'une liste ne contenait que des instances d'un certain type, par exemple des **`Employe`**
  - certaines erreurs ne pouvaient être repérées qu'à l'exécution et pas à la compilation
  - il fallait sans arrêt *caster* les éléments des collections pour pouvoir utiliser les méthodes qui n'étaient pas dans **`Object`**

Le plus grave

- Si un objet contenu dans les collections n'est pas du type attendu, on a une erreur (**`ClassCastException`**) à l'exécution mais pas à la compilation

Collections à type particulier

- Si on voulait éviter ces problèmes (éviter les *casts* et vérifier les types à la compilation), il fallait écrire un code différent pour chaque collection d'un type particulier
- Par exemple, il fallait écrire une classe **`ListInteger`** et une classe **`ListEmploye`**
- Ce qui revenait à écrire plusieurs fois la même chose, en changeant seulement les types

Présentation de la généricité

Généralités

- La généricité permet de paramétrer une classe ou interface avec un ou plusieurs types de données
- On peut par exemple donner en paramètre le type des éléments d'un **`Arraylist`** : **`ArrayList<E>`**
- **`E`** est un paramètre de type formel (ou plus simplement paramètre de type) ou variable de type
- **`E`** sera remplacé par un argument de type pour typer des expressions ou créer des objets :

```
ArrayList<Integer> l =
        new ArrayList<Integer>();
```

Un peu de vocabulaire

- **`ArrayList<E>`** est un type générique (plus exactement une classe générique ; **`List<E>`** est une interface générique)
- **`ArrayList<Employe>`** est une instanciation du type générique **`ArrayList<E>`** ; c'est un type paramétré concret (plus exactement une classe paramétrée) dont l'argument de type est **`Employe`**
- Pour des raisons de compatibilité, Java a gardé les anciens types non génériques, **`ArrayList`** par exemple ; on les appelle des « types raw »

Utilité de la généricité

- Elle permet d'écrire des collections dont tous les éléments ont le même type,
  - sans avoir à répéter plusieurs fois le même code
  - en vérifiant le typage dès la compilation
  - en évitant les *casts*

Extraits de la classe `ArrayList`

```
public class ArrayList<E>
    extends AbstractList<E> {
  public ArrayList() // pas ArrayList<E>() !
  public boolean add(E element)
  public E get(int index)
  public E set(int index, E element)
  public Iterator<E> iterator()
  . . .
```

Utilisation d'un `ArrayList` paramétré

```
List<Employe> employes =
     new ArrayList<Employe>();
Employe e = new Employe("Dupond");
employes.add(e);
// On ajoute d'autres employés
. . .
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
       employes.get(i).getNom());
}
```

Plus besoin de *cast*

Erreur détectée à la compilation

```
List<Employe> employes =
  new ArrayList<Employe>();
Employe e = new Employe("Dupond");
employes.add(e);
// On ajoute d'autres employés
. . .
// Ajoute un livre au milieu des employés
Livre livre = new Livre(...);
employes.add(livre); // Erreur de compilation
```

Définitions

- Type générique : une classe ou une interface paramétrée par une section de paramètres de la forme <T1, T2,…, Tn>
- Les Ti sont les paramètres de type formels
- Ils représentent des types inconnus au moment de la compilation du type générique
- On place la liste des paramètres à la suite du nom de la classe ou de l'interface :
  `List<E>`
  `Map<K,V>`

Où peuvent apparaître les paramètres de type ?

- Dans le code du type générique, les paramètres de type formels peuvent être utilisés comme les autres types pour déclarer des variables, des paramètres, des tableaux ou des types retour de méthodes :

```
E element;
E[] elements;
```

Où peuvent apparaître les paramètres de type ? (2)

- On peut *caster* avec un paramètre de type ; ça revient à *caster* avec le type qui contraint le paramètre de type (ou avec **`Object`** si le paramètre de type n'est pas contraint ; voir types contraints plus loin dans ce cours)
- Un tel *cast* provoquera un avertissement du compilateur (car pas sûr)

Où ne peuvent pas apparaître les paramètres de type ?

- Ils ne peuvent être utilisés pour créer des objets ou des tableaux, ni comme super-type d'un autre type
  - pas de « **`new E()`** »
  - pas de « **`new E[10]`** »
  - pas de « **`class C extends E`** »
- On ne peut utiliser un paramètre de type à droite de **`instanceof`** : pas de « **`x instanceof E`** »

Types des arguments de type

- Les arguments de type peuvent être des classes, même abstraites, ou des interfaces ; par exemple **`new ArrayList<Comparable>`**
- Ils peuvent même être des paramètres de type formels ; par exemple :

```
public class C<E> {
   . . .
   f = new ArrayList<E>();
```

- Un argument de type ne peut pas être un type primitif

Arguments de type abstraits

- On peut utiliser un argument de type abstrait (classe abstraite, interface) pour instancier une classe générique ; si **`EmployeI`** est une interface, on peut écrire :

```
List<EmployeI> l =
   new ArrayList<EmployeI>();
```

- Évidemment, il faudra remplir cette liste avec des employés « concrets »

Création d'une instance de classe paramétrée

- Au moment de la création d'une classe paramétrée, on indique le type de la collection en donnant un argument de type pour chaque paramètre de type formel
- ```
  List<String> liste =
     new ArrayList<String>();
  ```
- ```
  Map<String,Integer> map =
     new HashMap<String,Integer>();
  ```

Simplification (diamant)

- ```
  List<String> liste =
     new ArrayList<>();
  ```
- ```
  Map<String,Integer> map =
     new HashMap<>();
  ```

Utilisation des types génériques

- On peut utiliser les types génériques comme sur-type (classe mère ou interface)
- Exemple :
  `public class C<P> extends M<P>`
- Sinon, on ne peut utiliser un paramètre de type que dans une classe générique paramétrée par ce paramètre de type

Rappel sur le typage

- Un type B est un sous-type du type A si une valeur de type B peut être affectée à une variable de type A
- On note `B <: A`
- On dit aussi que A est un sur-type de B

Exemples

- Une classe fille est un sous-type de sa classe mère
- Une classe est un sous-type d’une interface qu’elle implémente
- A est un sous-type de A

Classe interne d'une classe générique

- Si une classe interne non static d'une classe générique utilise un paramètre de type de la classe générique, il ne faut pas faire l'erreur de mettre le paramètre de type dans la définition de la classe interne
- En effet, ce paramètre de type serait considéré comme un paramètre de type différent de celui de la classe englobante
- Remarque : au contraire, une classe interne static n'a pas accès aux paramètres de type de la classe englobante

Exemple

- ```
  public class Cache<K,T> {
     private class EntreeCache<T> {
     ...
  ```
  est une erreur
- Il faut écrire
  ```
  public class Cache<K,T> {
     private class EntreeCache {
     ...
  ```
  et il est possible d'utiliser **`T`** dans le code de la classe **`EntreeCache`**
  ```

Méthodes génériques

Méthode générique

- Comme une classe ou une interface, une méthode (ou un constructeur) peut être paramétrée par un ou plusieurs types
- Une méthode générique peut être incluse dans un classe non générique, ou dans une classe générique (si elle utilise un paramètre autre que les paramètres de type formels de la classe)

Syntaxe

- Une liste de paramètres apparaît dans l'en-tête de la méthode pour indiquer que la méthode ou le constructeur dépend de types non connus au moment de l'écriture de la méthode : **`<T1,T2,…> … m(…)`**
- Exemple de l'interface **`Collection<E>`** :

```
public abstract <T> T[] toArray(T[] a)
```

Instanciation d'une méthode générique

- On peut appeler une méthode paramétrée en la préfixant par le (ou les) type qui doit remplacer le paramètre de type : **`<String>m()`**
- (Syntaxe : si **`m`** est une méthode **`static`** de la classe **`Util`**, on écrit : **`Util.<String>m()`** ; si elle n'est pas **`static`**, on écrit : **`x.<String>m()`**)
- Mais le plus souvent le compilateur peut faire une inférence de type (« deviner » le type) d'après le contexte d'appel de la méthode

Inférence de type

- On appelle alors la méthode paramétrée sans la préfixer par une argument de type :

```
ArrayList<Personne> liste;
. . .
Employe[] res =
   liste.toArray(new Employe[0]);
// Inutile de préfixer par le type :
// liste.<Employe>toArray(...);
```

Inférence de type (2)

- Parfois, c'est un peu plus complexe pour le compilateur :

```
public <T> T choose(T a, T b) { ... }
. . .
Number n = choose(new Integer(0),
                  new Double(0.0));
```

Le compilateur infère **`Number`** qui est la « plus proche » super-classe de **`Integer`** et **`Double`**

- Des compléments sur les méthodes génériques seront donnés dans la section sur les types contraints
- On verra que le paramètre de type **`T`** peut être contraint par un type ; par exemple **`<T extends Comparable…>`**

Instanciation de type générique avec joker (*wildcard instantiation* en anglais)

Sous-typage et généricité

- Puisque que **`ArrayList<E>`** implémente **`List<E>`**, **`ArrayList<E>`** est un sous-type de **`List<E>`**
- Ceci est vrai pour tous les types paramétrés construits à partir de ces 2 types
- Par exemple, **`ArrayList<Personne>`** est un sous-type de **`List<Personne>`** ou **`ArrayList<Integer>`** est un sous-type de **`List<Integer>`**

Sous-typage pour les types paramétrés

- **Mais attention**, si **`B`** hérite de **`A`**, les classes **`ArrayList<B>`** et **`ArrayList<A>`** n'ont aucun lien de sous-typage entre elles
- Exemple : **`ArrayList<Integer>`** n'est pas un sous-type de **`ArrayList<Number>`** !! (**`Number`** est la classe mère de **`Integer`**)

Raison

- Si **`ArrayList<B>`** était un sous-type de **`ArrayList<A>`**, le code suivant compilerait :

```
ArrayList<A> la = new ArrayList<B>;
la.add(new A());
```

- Quel serait le problème ?
- Ce code autoriserait l'ajout dans un **`ArrayList<B>`** d'un élément qui n'est pas un **`B`** !

Conséquences

- Une méthode **`sort(ArrayList<Number> aln)`** qui trie un **`ArrayList<Number>`** ne peut être appliquée à un **`ArrayList<Integer>`**
- De même, l'instruction suivante est interdite :

```
ArrayList<Number> l =
   new ArrayList<Integer>();
```

Exemple de problème

- ```
  public static void printElements(
      ArrayList<Number> liste) {
    for(Number n : liste) {
      System.out.println(n.intValue());
    }
  }
  ```
- ```
  ArrayList<Double> liste =
    new ArrayList<Double>();
  ...
  printElements(liste); // Ne compile pas !
  ```

Problème pour la réutilisation

- Ceci nuit évidemment beaucoup trop à la réutilisation
- Les instanciations de type générique avec « joker » (*wildcards)* ont été ajoutées pour pallier ce problème

Instanciations avec joker

- Elles permettent de relâcher les contraintes sur les types paramétrés pour rendre des méthodes plus réutilisables
- Faire une instanciation avec joker d'un type générique c'est donner un type avec joker (« ? » est le joker) comme argument de type

Exemple d'instanciation avec joker

- **`List<? extends Number>`**
  est une liste dont l'argument de type est un sous-type de **`Number`**
- On ne connaît pas l'argument de type exact mais on sait qu'il est un sous-type de **`Number`**

Une bonne méthode pour afficher les nombres d'une liste

- ```
  public void printElements(
      List<? extends Number> liste) {
    for(Number n : liste) {
      System.out.println(n.intValue());
    }
  }
  ```
- ```
  List<Double> liste =
      new ArrayList<Double>();
  ...
  printElements(liste); // Compile
  ```

Les « types » avec joker

- **<?>** désigne un type *inconnu*
- **<? extends A>** désigne un type *inconnu* qui est **A** ou un sous-type de **A**
- **<? super A>** désigne un type *inconnu* qui est **A** ou un sur-type de **A**
- **A** peut être une classe, une interface, ou même un paramètre de type formel
- On dit que **A** contraint le joker (impose une contrainte sur le type inconnu)

Remarque

- Un seul type à la suite de **`extends`** ou de **`super`**
  (au contraire des types contraints que l'on verra plus loin)

Où peuvent apparaître les « types » avec joker ?

- Attention, c'est un abus de langage, ce ne sont pas des vrais types
- Ils ne peuvent être utilisés que pour instancier un type paramétré :
  `List<? extends Number>`
- Ils ne peuvent pas être utilisés, par exemple, pour déclarer une variable :
  ~~**`<? extends Number> n;`**~~

Où peuvent apparaître les instanciations de type avec joker ?

- Dans une classe quelconque (non générique ou générique)
- Peuvent être utilisées pour typer des variables, des paramètres, des tableaux ou les valeurs retour des méthodes :

```
List<? extends Number> l;
```

Ce qui est interdit avec les instanciations de type avec joker

- Ne peuvent pas être utilisées pour créer des objets :

  ~~`new ArrayList<? extends Integer>()`~~ // interdit
- Ne peuvent pas être utilisés comme super type :

  ~~`class C implements Comparable<? super C>`~~

Ce qui est interdit avec les instanciations de type avec joker

- Ne peuvent pas être utilisées pour indiquer le type des éléments d'un tableau au moment de sa création :
  ~~`new List<? extends Number>[10]`~~ // interdit
- Cependant le type d'un tableau peut être une instanciation avec joker non contraint :
  `new List<?>[10]` // autorisé

Exemples avec les tableaux

```
List <? extends Number>[] t1;          // autorisé
t1 = new List<? extends Number>[10]; // interdit
// On crée un tableau dont chaque élément est
// une liste d'un certain type.
// Les types de chaque ligne ne sont pas obligés
// d'être les mêmes
List <?>[] t2 = new List<?>[10];
t2[0] = new ArrayList<Integer>();
t2[1] = new ArrayList<String>();
```

Quels types pour contraindre un joker ?

- Tous les types sont permis, sauf les types primitifs, y compris les paramètres formels de type et les instanciations avec joker
- Exemples :

```
List<? extends String[]>
List<? extends Comparable<String>>
List<? extends Comparable<? super Long>>
List<? extends Map.Entry<?,?>>
List<? extends T>
```

(dans une classe générique paramétrée par **T**)

extends, même si le type qui contraint est une interface

Les « ? » sont indépendants. Il ne s'agit pas du même type.

Exemples d'utilisation

- Classe **`ArrayList<E>`** :

```
public boolean addAll(
    Collection<? extends E> c)
```

  Intuition : si une collection contient des **E**, on peut lui ajouter des éléments d'une sous-classe de **E**

- Classe **`Collections`** :

```
public static boolean disjoint(
    Collection<?> c1,
    Collection<?> c2)
```

  Les ? de **`c1`** et **`c2`** sont indépendants

  Intuition : on peut toujours voir si 2 collections ont des éléments en commun, quels que soient les types qu'elles contiennent

Exemple de code de la classe Collections

```
static <T> void
fill(List<? super T> liste, T elem) {
  int size = liste.size();
  for (int i = 0; i < size; i++)
    liste.set(i, elem);
}
```

- Remplace tous les éléments d'une liste par un certain élément
- Intuition : pour remplir avec un objet de type **T**, le type des éléments de la liste doit être un sur-type de **T**

Moyen mnémotechnique

- On a vu que les types avec joker permettent d'écrire des méthodes plus réutilisables
- « PECS » peut aider à se souvenir des cas où le remplacement du type **`T<E>`** d'un paramètre par un type avec joker **`T<? extends E>`** ou **`T<? super E>`** permet d'étendre les utilisations d'une méthode : Producteur-Extends, Consommateur-Super

cf livre « Effective Java » de Joshua Bloch

Moyen mnémotechnique

- Un producteur de type **T<E>** est utilisé par la méthode pour fournir un objet de type **E** (on lui envoie un message qui retourne un tel objet) ; un consommateur utilise un objet de type **E** (on lui envoie un message qui prend en paramètre un objet de type **E**)
- Si le paramètre de type **T<E>** est à la fois consommateur et producteur, il n'est pas possible de généraliser et il faut laisser **T<E>**

Exemple

- **`liste`** dans **`printElements(ArrayList<Number> liste)`** ne fait que produire les éléments de type **`Number`** qui seront affichés ; on peut généraliser avec **`ArrayList<? extends Number>`**
- **`liste`** dans **`fill(List<T> liste, T elem)`** ne fait que consommer les éléments de type **`T`** qui lui seront affectés ; on peut généraliser avec **`ArrayList<? super T>`**

Type retour

- En revanche, il ne faut pas déclarer un type avec joker comme type retour d'une méthode
- En effet, si on écrit « **`List<? extends D> m()`** » dans la classe **`C`**, on ne pourra écrire

```
C c = new C(…);
List<D> l = c.m();
```

car **`List<D>`** n'est pas un sur-type de **`List<? extends D>`**

- On sera obligé de récupérer la valeur de retour dans une variable d'un type avec joker, ce qui imposera des restrictions sur son utilisation (voir transparents suivants)

Problème ?

- On a vu que si **`ArrayList<B>`** héritait de **`ArrayList<A>`**, on pourrait ajouter un **`A`** dans une liste de **`B`**
- Ne va-t-on pas avoir le même problème avec les instanciations avec joker, puisque **`ArrayList<B>`** est un sous-type de **`ArrayList<? extends A>`** ?

Pas de problème...

- Réponse : non, car le compilateur impose des restrictions sur l'utilisation des instanciations de types avec joker
- C'est ce qu'on va étudier dans les transparents suivants
- On va voir en particulier que le code suivant ne sera pas autorisé :

```
ArrayList<? extends A> la =
    new ArrayList<B>();
la.add(new A());
```

Des règles strictes pour la sécurité

- Le compilateur impose des règles strictes sur l'utilisation des objets dont le type déclaré est une instanciation avec joker
- Ces règles sont utiles pour éviter, par exemple, qu'un **`Double`** ne soit ajouté dans une liste de **`Integer`**
- Les transparents suivants donnent des indices pour comprendre ces règles et savoir écrire du code qui les respectent
- Ils demandent une attention soutenue

Réflexions sur le typage (1)

- Mettons-nous à la place du compilateur qui rencontre l'instruction

```
T1 v1 = v2;
```

- Supposons que le compilateur n'a qu'une connaissance partielle du type **`T1`** (c'est le cas si **`T1`** correspond à un « type » avec joker)
- Examinons quelques cas où le compilateur peut accepter cette instruction et d'autres cas où il doit la rejeter

Réflexions sur le typage (2)

```
T1 v1 = v2;
```

- Si le compilateur connaît une borne basse **`BBT1`** de **`T1`** (un sous-type de **`T1`**)
  (c'est le cas si **`T1`** est le pseudo-type « **`? super BBT1`** »)
- Il ne doit accepter cette affectation que si (condition suffisante mais pas nécessaire) **`v2`** est d'un sous-type de **`BBT1`**
- S'il ne connaît pas un borne basse de **`T1`**, l'affectation doit être rejetée (sauf si **`v2`** est **`null`**)

Réflexions sur le typage (3)

- `T1 v1 = v2;`
- Si le compilateur connaît une borne haute **`BHT2`** (un sur-type) pour le type **`T2`** de **`v2`** (c'est le cas si **`T2`** est le pseudo-type « **`? extends BHT2`** »)
- Cette affectation sera acceptée si **`T1`** est d'un sur-type de **`BHT2`**
- Sinon, le seul cas où le compilateur acceptera l'instruction est le cas où **`T1`** est le type **`Object`**

Echanges d'une classe avec l'extérieur

- Une classe fournit des objets à l'extérieur par ses méthodes qui renvoient des objets : **`TypeRetour m2()`**
- Elle reçoit des objets de l'extérieur par ses méthodes qui prennent des objets en paramètre : **`Ret m1(TypeDuParametre x)`**
- Lorsque la classe est une instanciation avec joker d'une classe générique, il y a des restrictions sur l'utilisation des méthodes qui échangent des objets du type paramétré avec l'extérieur

Classe générique

- **`C<E>`** : classe qui manipule des objets de type **`E`**
- Les restrictions portent sur les méthodes qui échangent des objets de type **`E`** avec l'extérieur :
  - **`void m1(E e)`** (extérieur → C) (le type retour n'a pas d'importance ; on a mis **`void`** pour simplifier)
  - **`E m2()`** (C → extérieur)

Objets passés à C<? extends T>

- Utilisation de la méthode m1 de signature « void m1(E) » : **`c.m1(x);`**
- « Signature » de m1 pour cette instantiation : « void m1(? extends T) »
- Comment doit être le type **X** de **x** pour que l'affectation « x → ? extends T » soit acceptée par le compilateur ?
- Aucun **X** pour lequel cette affectation est assurée de ne pas poser de problème
- Donc l'appel d'une telle méthode est interdit

Application

- Cette règle s'applique à la méthode **`add`** de **`List`** :

```
ArrayList<? extends A> la =
     new ArrayList<B>();
la.add(new A());
```

Autre exemple

```
static <T> void
fill(List<? extends T> liste, T elem) {
  int size = liste.size();
  for (int i = 0; i < size; i++)
    liste.set(i, elem);
}
```

Ca compile ?

- Cette méthode ne compile pas. Pourquoi ?

Autre exemple

```
static <T> void
fill(List<? extends T> liste, T elem) {
  int size = liste.size();
  for (int i = 0; i < size; i++)
    liste.set(i, elem);
}
```

- set de List<E> a pour signature set(int, E)
- Il est interdit d'appeler une telle méthode pour **`List<? extends T>`**

Objets fournis par C<? extends T>

- Utilisation de la méthode m2 de signature « E m2() » : **`X x = c.m2();`**
- « Signature » de m2 pour cette instanciation : « ? extends T m2() »
- Comment doit être le type **X** pour que l'affectation « **`? extends T`** → **X** » soit acceptée par le compilateur ?
- **X** doit être de type **T** ou d'un sur-type de **T**

Exemple

```
ArrayList<? extends Personne> l =
      new ArrayList<Employe>();
...
Personne e = l.get(0);
```

Ca compile ?

Et si on remplace **`Personne`** par **`Employe`** ?

Objets passés à C<? super T>

- Utilisation de la méthode m1 de signature « void m1(E) » : **`c.m1(x);`**
- « Signature » de m1 pour cette instanciation : « void m1(? super T) »
- Comment doit être le type **X** du paramètre **x** pour que l'affectation « **X → ? super T** » soit acceptée ?
- **X** doit être le type **T**, ou un sous-type de **T**

Exemple

```
void f(List<? super Cercle> liste) {
   liste.add(machin);
}
```

De quel type peut être machin ?

- **`machin`** doit être déclaré du type **`Cercle`** (ou d'un sous-type)

Objets fournis par C<? super T>

- Utilisation de la méthode m2 de signature « E m2() » : **`X x = c.m2();`**
- « Signature » de m2 pour cette instanciation : « ? super T m2() »
- Comment doit être le type **`X`** pour que l'affectation « **`? super T`** → **`X`** » soit acceptée ?
- Le seul cas possible : **`X`** doit être le type **`Object`**

Exemple

```
ArrayList<? super Employe> l =
        new ArrayList<Personne>();
...
Employe e = l.get(0);
```

Ca compile ?

Et si on remplace **`Employe`** par **`Personne`** ?

Message d'erreur du compilateur

- Pour pouvoir repérer, comprendre et corriger plus rapidement les erreurs liées aux utilisations interdites des méthodes des instanciations avec joker, examinons le message d'erreur envoyé par le compilateur pour le code du transparent 73 :

```
set(int,capture of ? extends T) in
java.util.List<capture of ? extends T>
cannot be applied to (int,T)
```

Étude du message d'erreur

- ```
  set(int,capture of ? extends T) in
  java.util.List<capture of ? extends T>
  cannot be applied to (int,T)
  ```
- « **`capture of ?`** » : le compilateur désigne ainsi le type inconnu **`<? extends T>`** ; appelons-le « ***`T?`*** »
- Comme le type **`E`** est instancié par ***`T?`***, la signature de **`set`** est : ***`T?`*** **`set(int, T?)`**
- ***`T?`*** étant un sous-type de **`T`** (sans borne basse connue), il n'y a aucune raison que cette méthode accepte comme 2ème paramètre **`elem`** qui est de type **`T`**
  ```

Test avec extends

```
void f(List<? extends Figure> liste) {
  // Autorisé ?
  Figure ret = liste.get(0);
  // Autorisé ?
  Cercle ret = liste.get(0);
}
```

Complément sur le test

```
void f(List<? extends Figure> liste) {
  // On peut caster :
  Cercle ret = (Cercle)liste.get(0);
```

- Pour passer outre l'interdiction, on peut *caster*
- Ça compilera, mais une erreur éventuelle ne sera connue qu'à l'exécution ; on ne profite pas de la sécurité offerte par les types génériques

Test avec super

```
void f(List<? super Cercle> liste) {
   // Autorisé ?
  Object ret = liste.get(0);
   // Autorisé ?
  Cercle ret = liste.get(0);
}
```

Instanciation C<?>

- En toute logique on a les contraintes des 2 autres types d'instanciation (**`E`** est le paramètre de type de la classe **`C`**) :
  - l'appel d'une méthode qui prend un paramètre de type **`E`** (le paramètre de type) est interdit (sauf si on lui passe la valeur **`null`**)
  - la valeur retournée par une méthode qui renvoie un **`E`** ne peut être affectée qu'à une variable de type **`Object`**

Utilité de ces règles

- Le plus souvent les règles que l'on vient de voir ne doivent pas être contournées car elles sont nécessaires pour obtenir du code correct
- Prenons par exemple la méthode **`add(E elt)`** de la classe **`ArrayList<E>`**
- Grâce à ces règles on ne peut pas ajouter d'élément à une **`ArrayList<? extends Number>`**, ce qui est normal car le type fixé, mais inconnu, des éléments de la liste peut être n'importe quel sous-type de **`Number`**

Contourner une restriction

- Quelquefois, cependant, ces règles ajoutent une contrainte qui ne sert à rien
- C’est en particulier le cas pour les méthodes qui ne modifient pas la collection

Exemple

- Si la méthode de **`ArrayList<E>`** qui indique si un élément appartient à la liste avait cette signature :

```
boolean contains(E elt)
```

- On ne pourrait appeler la méthode pour une **`ArrayList<? extends Number>`**, alors que cet appel ne peut causer de problème
- C'est pour cette raison que la méthode de l'API n'a pas la signature ci-dessus mais celle-ci :

```
boolean contains(Object elt)
```

Remarque

- Ne pas considérer que **<?>** est identique à **Object**
- **<?>** est un type que l'on ne connaît pas, qui peut être différent de **Object**
- Exemple de différence :

```
ArrayList<Object> l1 =
  new ArrayList<Object>();
ArrayList<?> l2 =
  new ArrayList<Object>();
Object o = new Object();
l1.add(o); // Compile
l2.add(o); // Ne compile pas
```

Méthode paramétrée ou joker ?

- Exemples de l'interface **`Collection<E>`** :

```
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
```

- Les versions avec méthode paramétrée seraient :

```
<T> boolean containsAll(Collection<T> c);
<T extends E> boolean addAll(Collection<T> c);
```

(voir types contraints à la section suivante)

- Quand le type paramétré **`T`** apparaît une seule fois comme dans ces 2 exemples, la version avec joker est préférable (plus simple à comprendre)

Méthode paramétrée ou joker ?

- Une méthode paramétrée permet d'indiquer que 2 paramètres ont des types dépendants :

```
static <T> void
fromArrayToCollection(T[] a,
                      Collection<T> c)
```

- C'est impossible à traduire avec une instanciation avec joker
- Question subsidiaire : pouvez-vous donner une meilleure signature ?

Sous-typage

- Soit **`B`** une classe fille de **`A`**
- **`ArrayList<B>`** est-il un sous-type de **`ArrayList<? extends A>`** ?
- Oui, car **`B`** est bien un sous-type de **`A`**
- **`ArrayList<? extends A>`** est-il un sous-type de **`ArrayList<B>`** ?
- Non, le « ? » peut représenter un sous-type de **`A`** qui n'est ni **`B`**, ni un sous-type de **`B`**

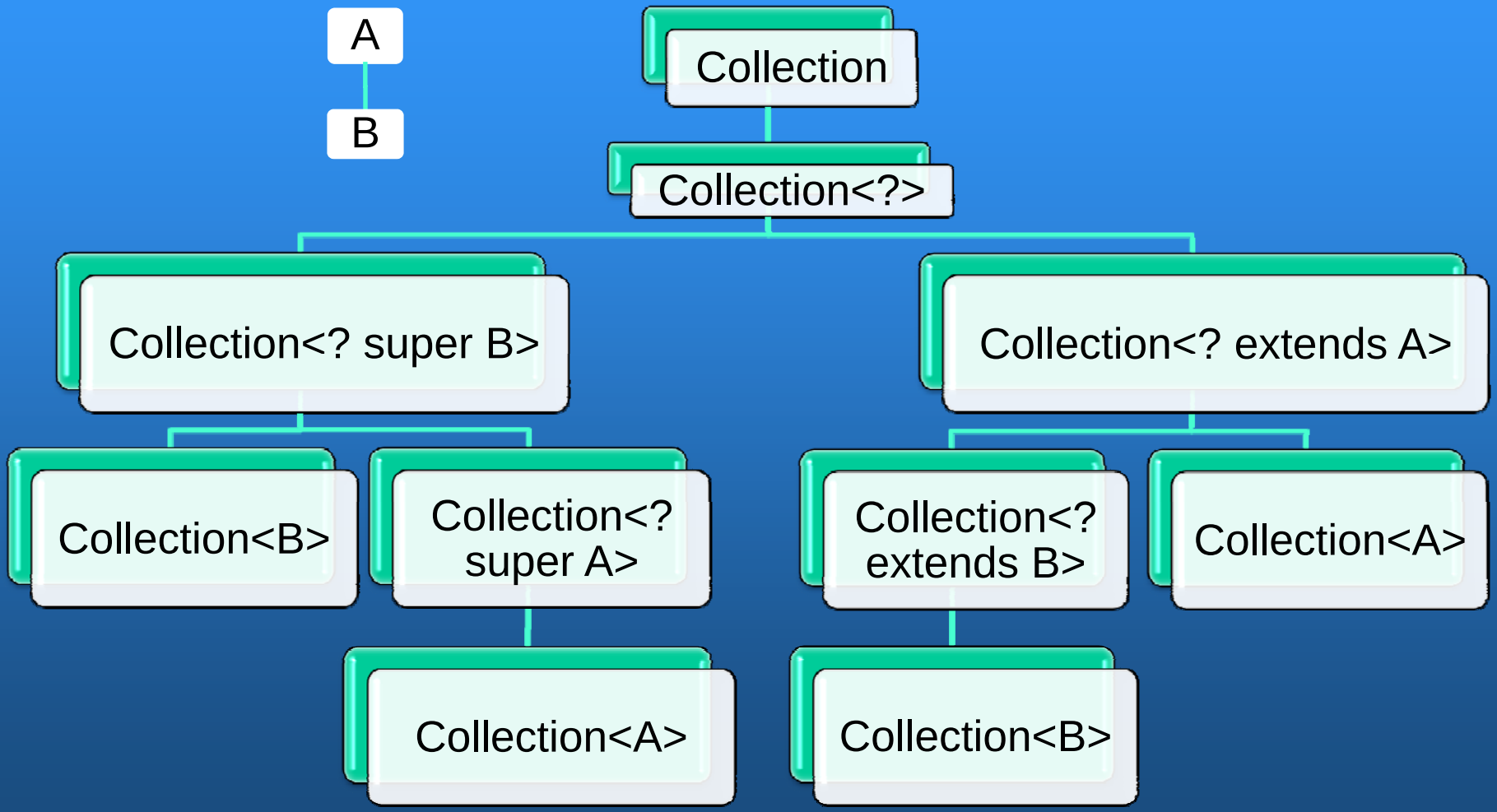
Relations de sous-typage (1/2)

- Voici les relations de sous-typage principales
- Soit **`B`** est un sous-type de **`A`**
- **`List<A>`** est un sous-type de **`Collection<A>`**
- **`ArrayList<A>`** est un sous-type de **`List<A>`**
- Mais **`List<B>`** **n'est pas** un sous-type de **`List<A>`**
- Tous les types génériques construits à partir de **`List`** sont des sous-types de **`List`** (« raw »)

Relations de sous-typage (2/2)

- **`List<A>`** est un sous-type de **`List<?>`**
- **`List<? extends A>`** est un sous-type de **`List<?>`**
- **`List<? super A>`** est un sous-type de **`List<?>`**
- **`List<A>`** est un sous-type de **`List<? extends A>`**
- **`List<B>`** est un sous-type de **`List<? extends A>`**
- **`List<B>`** n'est pas un sous-type de **`List<? super A>`**
- **`List<? extends A>`** n'est pas un sous-type de **`List<A>`**

Résumé des relations de sous-typage



Types paramétrés contraints (*bounded* en anglais)

Pourquoi des types contraints ?

- Lorsqu'on utilise un paramètre de type formel **`T`**, on ne peut rien supposer sur le type **`T`**
- Si une variable ou un paramètre **`v`** est déclaré de type **`T`**, aucune méthode ne peut être appelée sur **`v`** (à part celles qui sont dans la classe **`Object`**)

Pourquoi des types contraints ?

- Cependant pour écrire une méthode de tri « **`sort(ArrayList<E>)`** », il est indispensable de pouvoir comparer les éléments de la liste
- Il faut un moyen de dire que le type **`E`** contient une méthode pour comparer les instances du type
- Le moyen habituel en Java est de dire que le type **`E`** hérite d'une classe ou implémente une interface qui contient la méthode de comparaison ; c'est possible grâce aux types contraints

Syntaxe

- Soit **`T`** un paramètre de type d'une classe, d'une interface ou d'une méthode générique
- On peut indiquer que **`T`** doit hériter d'une classe mère **`M`** (au sens large ; **`T`** pourra être **`Mere`**) :
  `<T extends Mere>`
- ou qu'il doit implémenter (ou qu'il doit hériter d') une ou plusieurs interfaces :
  `<T1 extends I1 & I2, T2, T3>`

2 autres paramètres de type

Syntaxe (2)

- Si **`T`** hérite d'une classe mère et implémente des interfaces, la classe mère doit apparaître en premier :
  `<T extends Mere & I1 & I2>`
- Remarque : comme le but est de permettre d'appeler les méthodes d'une classe mère ou d'une interface, on peut contraindre un type par **`extends`** mais pas par **`super`** comme pour les instanciations avec joker

Quels types pour contraindre un paramètre de type ?

- Tous les types sont permis, sauf les types primitifs et les tableaux
- Y compris les instanciations avec joker, les énumérations, les paramètres de type formels :

```
<T extends Number>
<T extends List<String>>
<T extends ArrayList<? extends Number>>
<T extends E>
<A extends Personne, B extends A>
```

Exemple

```
public static <T extends Comparable<? super T>>
  T min(List<T> liste) {
  if (liste == null || liste.isEmpty())
    return null;
  T min = liste.get(0);
  for (int i = 1; i < liste.size(); i++) {
    if (liste.get(i).compareTo(min) < 0)
      min = liste.get(i);
  }
  return min;
}
```

Question subsidiaire : meilleure signature ?

Où peut apparaître un type contraint ?

- Partout où un paramètre de type peut apparaître, dans l'en-tête d'un type ou d'une méthode générique
- Exemples :
  - `class Pair<A extends Comparable<A>, B extends Comparable<B> >`
  - `<T extends Personne> void m()`

Argument de type d'un type contraint

- La contrainte impose des conditions sur l'argument de type qui prendra la place du paramètre de type
- Le compilateur provoquera une erreur si cette contrainte n'est pas satisfaite
- Ainsi, la méthode **`min`** vue dans l'exemple précédent pourra être instanciée avec le type **`String`**, **`<String>min`** car **`String`** implémente **`Comparable<String>`**, mais pas avec le type **`Object`**

Capture de joker

- La capture de joker permet de résoudre un problème

Description du problème

- On veut écrire une méthode qui transforme un ensemble en ensemble non modifiable (exemple extrait de l'article sur les instanciation avec jokers cité à la fin de ce support, section 3.2)
- La signature naturelle :
  `<T> Set<T> unmodifiableSet(Set<T> set)`
- Problème : on ne devrait pas pouvoir lui passer un ensemble déclaré de type **`Set<?>`** car **`Set<?>`** n'est pas un sous-type de **`Set<T>`** (on ne peut affecter un **`Set<?>`** à un **`Set<T>`**)

Une mauvaise solution

- La signature
  `Set<?> unmodifiable(Set<?> set)`
  permet d'accepter tous les ensembles, y compris les **`Set<?>`**, mais on perd l'information importante que l'ensemble renvoyé a des éléments de même type que l'ensemble passé en paramètre
- En effet, 2 « **`Set<?>`** » n'ont aucune raison d'avoir des éléments de même type

La bonne solution : capture de joker

- Dans les cas où le compilateur n'a affaire qu'à un seul type pour un seul joker, il peut faire comme si on lui avait passé un type fixé, appelons-le « **T?** »
- C'est le cas s'il n'y a qu'un seul joker utilisé dans les paramètres et qu'il concerne le premier niveau, par exemple **`Set<?>`** ou **`List<? extends A>`**
- Ça n'est pas le cas pour **`List<List<?>>`** (le **`?`** pourrait correspondre à plusieurs types différents)
- **T?** est appelé une capture du joker

Conclusion

- On peut donner à la méthode la signature naturelle **`<T> Set<T> unmodifiableSet(Set<T> set)`**
- Le code suivant sera tout de même accepté par le compilateur, grâce à la capture de joker (bien que **`Set<?>`** ne soit pas un sous-type de **`Set<T>`**) :

```
Set<?> set = …..
Set<?> set2 = unmodifiableSet(set);
```

- Le compilateur capture toutes les informations sur le joker ; on pourra donc écrire (**`T`** est remplacé par **`? extends A)`** :

```
 Set<? extends A> set = …..
Set<? extends A> set2 = unmodifiableSet(set);
```

Autre utilisation de la capture

- On veut écrire une méthode qui mélange les éléments d'une liste
- La signature naturelle la plus simple : **`static void shuffle(List<?> list)`**
- Mais on ne pourra implémenter la méthode avec cette signature car, dans le code on aura besoin d'affectation du type « **`T elt = …`** » où **`T`** est le type des éléments de la liste ; si on utilise une signature avec joker, on ne peut écrire « **`? elt = …`** » (un joker ne peut être utilisé dans ce contexte)

Une mauvaise solution

- Une méthode générique de signature
  `static <T> void shuffle(List<T> list)`
  conviendrait mais on a vu qu'utiliser un joker est la meilleure solution dans le cas où le type n'apparaît qu'une seule fois et c'est le cas ici
- C'est dommage de choisir une signature uniquement à cause d'un problème d'implémentation

La bonne solution : capture de joker

- On donne la signature naturelle la meilleure à la méthode mais on fait appel en interne à une méthode générique (qui acceptera en paramètre un **`List<?>`** grâce à la capture de joker) :

```
static void shuffle(List<?> list) {
  privShuffle(list);
}
private static
<T> void shuffle(List<T> list) {
  ...
  T elt = ...;
}
```

Implémentation de la généricité et restrictions associées

Instanciation des classes génériques

- Au contraire des *templates* de C++, les instanciations des types génériques ne donnent à l'exécution qu'un seul type dans la JVM (le type « *raw* »)
- Par exemple, **`ArrayList<Integer>`** et **`ArrayList<Employe>`** sont représentés dans la JVM par une seule classe **`ArrayList`**
- C# se comporte comme Java si l'argument de type est un type objet et comme C++ si c'est un type primitif (possible en C#)

Une contrainte importante

- Pour Java, une des contraintes fixée au départ dans le cahier des charges de la généricité :

  tout le code écrit avant la généricité doit pouvoir être utilisé avec la nouvelle version de Java qui offre la généricité

- Cette contrainte est difficile à respecter

Effacement de type

- La solution choisie pour respecter la contrainte est appelée « effacement de type » (*type erasure*)
- Par la suite on utilisera « *erasure* » pour désigner l'effacement de type

C’est le compilateur qui fait tout

- En Java, c’est le compilateur qui fait tout le travail pour permettre l’utilisation de types génériques dans le code
- En fait le compilateur transforme le code générique en enlevant presque toute trace de généricité
- A peu de chose près le code compilé pourrait fonctionner sur les JVM antérieures à l’introduction de la généricité

Actions effectuées par le compilateur

- Vérification du typage compte tenu de toutes les informations qu'il détient (paramètres de type compris)
- Remplacement des types paramétrés par les types « *raw* » correspondant
- Remplacement des paramètres de type par des types « ordinaires »
- Ajouts de *casts* si nécessaire
- Ajouts de méthodes « ponts » si nécessaire

Exemple d'effacement de type

Voici comment on pourrait représenter le résultat de l'effacement de type pour le code du transparent 15 :

```
List employes = new ArrayList();
Employe e = new Employe("Dupond");
employes.add(e);
. . . // On ajoute d'autres employés
for (int i = 0; i < employes.size(); i++) {
  System.out.println(
      ((Employe)employes.get(i)).getNom());
}
```

Transformation des types génériques

- **`List<String>`** et **`List<Integer>`** correspondent à la seule Interface **`List`** dans la JVM
- De même les classes **`ArrayList<String>`** et **`ArrayList<Integer>`** correspondent à la seule classe **`ArrayList`** dans la JVM
- **`HashMap<K,V>`** est transformé en **`HashMap`**
- **`ArrayList<Integer>[]`** est transformé en **`ArrayList[]`**

Remplacement des paramètres de type

- Chaque paramètre de type est remplacé par un type « ordinaire »
- **`<T>`** est remplacé par **`Object`**
- **`<T extends Comparable>`** est remplacé par **`Comparable`**
- Plus généralement, les types contraints sont remplacés par le premier type donné dans la liste ; par exemple, « **`<T extends T1 & T2 & T3>`** » est remplacé par le type **`T1`**

Casts ajoutés par le compilateur

- Puisque les paramètres de type ne sont pas conservés, le compilateur est obligé d'ajouter des *casts* explicites comme on le faisait avec les versions de Java sans généricité :

```
List<String> liste = …;
. . .
String s = liste.get(i);
```

va être transformé en

```
List liste = …;
. . .
String s = (String)(liste.get(i));
```

Transformation des méthodes

- Les signatures et les types retour des méthodes sont aussi transformés en remplaçant les types génériques par leur *erasure*
- Par exemple, « **`E get(int i)`** » est transformée en « **`Object get(int i)`** », « **`boolean add(E elt)`** » devient « **`boolean add(Object elt)`** »

Implication pour les surcharges de méthodes

- Du fait de l'effacement de type dans les signatures des méthodes il est interdit d'avoir dans une classes 2 méthodes dont les signatures ne dépendent que par l'argument de type d'un des paramètres de type générique
- Par exemple, il est interdit d'avoir ces 2 méthodes dans une classe :
  - `void m(List<Message> messages)`
  - `void m(List<Email> emails)`

Pont pour les méthodes

- Pour les méthodes une difficulté supplémentaire vient de l'*erasure* des méthodes génériques qui sont redéfinies
- Le compilateur doit ajouter une méthode « pont » dans les sous-classes pour que le polymorphisme puisse fonctionner correctement

Nécessité des ponts

- L'interface **`Comparable<T>`** contient la méthode **`public int compareTo(T o)`**
- La classe **`Integer`** implémente l'interface **`Comparable<Integer>`** ; elle contient la méthode **`public int compareTo(Integer i)`**
- L'erasure des 2 méthodes **`compareTo`** donne : **`int compareTo(Object)`** pour **`Comparable`** **`int compareTo(Integer)`** pour **`Integer`**
- La deuxième ne redéfinit donc pas la première !

Ajout d'un pont

- Le compilateur va donc rajouter une méthode pont dans **`Integer`**, qui va avoir la signature de la méthode **`compareTo`** de **`Comparable`** et qui va appeler la méthode **`compareTo`** de **`Integer`** :

```
public int compareTo(Object o) {
  return compareTo((Integer) o);
}
```

Restrictions

- L’effacement de type va impliquer des compromis et des impossibilités qui nuisent à la facilité d’utilisation et à la compréhension de la généricité en Java
- Les transparents suivants étudient quelques restrictions induites par l’effacement de type

Quelques restrictions

- Une classe ne peut implémenter plusieurs interfaces de même type *raw*, par exemple **`Comparable<String>`** et **`Comparable<Integer>`**
- 2 méthodes d'une classe ne peuvent donner la même signature après *erasure* ; par exemple, **`equals(T)`** et **`equals(Object)`**

Quelques restrictions (2)

- On ne peut utiliser **`instanceof`** sur un type paramétré (~~**`instanceof List<String>`**~~) car à l'exécution la JVM ne connaît pas le type paramétré d'une instance (seulement le type *raw* et le type générique)
- Un seul cas particulier : **`instanceof List<?>`** est autorisé
- On ne peut créer un tableau dont le type est un paramètre de type (~~**`new T[]`**~~) ; utiliser plutôt une collection

Tableaux de types génériques

- On ne peut créer un tableau dont le type des éléments est un type paramétré

```
liste = new ArrayList<String>[50];
```

- Utiliser une liste à la place :

```
liste =
   new ArrayList<ArrayList<String>>();
```

Autres restrictions

- On ajoute ici d'autres restrictions que l'on a déjà vues, liées aux types génériques
- Un paramètre de type ne peut être utilisé pour créer un objet (~~`new T();`~~) ou comme classe mère d'une classe
- Une instanciation de type avec joker ne peut être utilisé pour créer un objet (~~`new ArrayList<? extends Integer>();`~~)
- ou pour créer un tableau (~~`new List<? extends Number>[10]`~~)

Contexte static

- Il est interdit d'utiliser une variable de type dans un contexte **`static`**
- Dans une classe générique **`G<T>`**, il est interdit d'utiliser **`T`** dans la déclaration d'une variable de classe **`static`** ou dans une méthode de classe **`static`**
- La raison : toutes les classes instanciations de la classe générique pour des argument de type différents partageraient ces membres **`static`** à cause de l'*erasure* (toutes ces classes correspondent à une seule classe **`G`** dans la JVM)

Types génériques et exceptions

- Une classe générique ne peut hériter de **`Throwable`** (donc ne peut être lancée comme exception)
- Une variable de type **`T`** ne peut typer une variable d'une clause **`catch`**
- Mais une variable de type peut apparaître à la suite d'une clause **`throws`** dans l'en-tête d'une méthode

Implantation d'une classe générique

- Si on veut écrire sa propre collection générique **`Coll<E>`**, on a toutes les chances de tomber sur le problème posé par le fait qu'il est interdit de créer un tableau du type générique
- En effet, on souhaitera certainement ranger les éléments de la collection dans un tableau de type **`E[]`**
- Comment faire ?

Solution préconisée

- On pourrait utiliser une des collections de **`java.util`** mais on perdrait en performance
- La solution préconisée est de ranger les éléments de la collection dans un tableau de type **`Object[]`** et de *caster* ses éléments en **`T`** quand c'est nécessaire ; par exemple :

```
public T get(int i) {
  return (T) elements[i];
}
```

Interopérabilité avec du code non générique

Utilisation d'anciennes méthodes avec des types paramétrés

- Soit une méthode **`m(List)`**
- On peut appeler cette méthode avec un paramètre de type **`List<String>`**
- On aura cependant un avertissement du compilateur car celui-ci ne peut assurer que la méthode **`m`** n'ajoutera pas des éléments d'un autre type que **`String`** à la liste

Utilisation de nouvelles méthodes avec des types « raw »

- De même, soit une méthode **`m(List<String>)`**
- On peut l'appeler en lui passant une **`List`**
- On aura cependant un avertissement du compilateur car celui-ci ne peut assurer que la liste passée en paramètre ne contient que des **`String`**

Option de javac

- Si on veut voir les détails des avertissements liés aux types « *raw* », il faut lancer javac avec l'option **`-Xlint:unchecked`**
- On peut ajouter une annotation **`@SuppressWarnings("unchecked")`** à une méthode ou à une classe pour éviter d'avoir ce type d'avertissement pour le code de la méthode ou de la classe
- À n'utiliser que lorsque l'on est certain que le code ne provoquera pas de problèmes de typage !

Quiz

- Quelles lignes provoquent une erreur à la compilation ?
- a) Panier p1 = new Panier();
- b) Panier pf = new Panier<Fruit>();
- c) Panier<Fruit> pf2 = new Panier<Fruit>();
- d) Panier<Pomme> pp1 = new Panier<Fruit>();
- e) Panier<Fruit> pf3 = new Panier<Pomme>();
- f) Panier<?> p2 = new Panier<Pomme>();
- g) Panier<Pomme> pp2 = new Panier<?>();

Réponse : d), e), g)

Tout le quiz : http://www.grayman.de/quiz/java-generics-en.quiz

Quiz (suite)

- Panier p = new Panier();
- a) Panier<Pomme> pp = p;
- b) Panier<Orange> po = p;
- c) pp.setElement(new Pomme());
- d) Orange orange = (Orange)po.getElement();

Réponse : Tout compile mais a), b) compilent avec un avertissement (« unchecked conversion ») et d) provoque une erreur à l'exécution (ClassCastException)

Réflexivité

Restrictions

- A cause de l'implémentation de la généricité par effacement de type, Java ne permet pas une réflexivité complète sur les types génériques (une partie des informations est perdue au moment de l'exécution)

Une nouvelle JVM

- Malgré tout, des modifications ont été apportées à la JVM du JDK 5 pour garder des traces du paramétrage des types génériques
- Le code compilé des types génériques ne peut donc être exécuté que sur une JVM compatible avec le JDK 5

Information sur les types génériques dans la JVM

- Ces informations permettent par exemple de retrouver l'en-tête d'une classe ou d'une méthode générique, par exemple **`ArrayList<E>`**
- Cependant cette information ne peut dire avec quel type la classe a été instanciée puisque tous les types **`ArrayList<String>`**, **`ArrayList<Integer>`**, **…** ne correspondent qu'à une seule classe dans la JVM

Nouvelles classes et interfaces

- Pour permettre un minimum de réflexivité sur les classes génériques et paramétrées, des classes et interfaces ont été ajoutées au JDK ; par exemple, l'interface **`java.lang.reflect.Type`** représente un super-type pour tous les types utilisés par Java, même les types primitifs et les types paramétrés
- Des méthodes ont été ajoutées à la classe **`Class`**, par exemple **`getTypeParameters()`** ou **`getGenericSuperClass()`**

Types paramétrés

- Ils sont représentés par l'interface **`java.lang.reflect.ParameterizedType`** qui contient une méthode qui permet de récupérer les arguments de type concrets du type paramétré : **`Type[] getActualTypeArguments()`**
- La récupération de la classe d'un argument peut permettre de créer une instance du type d'un des paramètres de type (on sait que « **`new T()`** » est interdit si **`T`** est un paramètre de type)

Exemple schématique (1)

```
class A<T> {
  private Class<T> classe;
  A() { // Appelé au début du constructeur de B
    // this est alors de type B = A<Machin>
    this.classe = (Class<T>)
     (((ParameterizedType)getClass()
        .getGenericSuperclass())
        .getActualTypeArguments()[0]);
  }
  T m() throws Exception {
    return classe.newInstance();
  }
}
```

Exemple schématique (2)

```
// Machin est une classe quelconque qui
// possède un constructeur sans paramètre
class B extends A<Machin> {}
...
B b = new B();
// b.m() renvoie bien une instance de Machin
Machin m = b.m();
```

Des liens

- Tutoriel de Sun, écrit par le responsable de l'introduction de la généricité dans Java : http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
- Article sur les instanciations avec jokers : http://bracha.org/wildcards.pdf
- Pleins de liens sur la généricité (dont une FAQ très complète) : http://www.langer.camelot.de/Resources/Links/JavaGenerics.htm
- Pour se tester : http://www.grayman.de/quiz/java-generics-en.quiz