Integer sequence; cours 1

### Lab: Advanced Language Constructs

## Génériques

#### 1.1 Solutions existantes

* 1) Dupliquer la classe générique selon l’objet à manipuler ⇒ cata car gestion duplication de code pas stable (maintenabilité)
* 2) généraliser la classe générique (remplacer integer par object)

Erreur commune non détectée :

public void add(Object element);

**public Object get(int indem);**

Sequence**<String>** stringSequence = new Sequence***<String>****(*);

***Argument type***

string Sequence.add(10);

**String premier = (String)stringSequence.get(0);**

**Correction de l’erreur.** Toujours pas terrible parce qu’il faut caster....

Solution : les templates.

* 3) génériques :

Sequence<E> : E = paramètre type

ex : **correction**. ( le bleu clair = pas obligatoire depuis Java 7 pcq le compilateur déduit tout seul).

#### 1.2 Solution générique

Problèmes rencontrés :

* Faire **new E()** ne marchera jamais. Same pour **new E[]**
* due à l’assurance de la compatibilité ascendante avec les vieilles librairies (qui sont écrit sans les génériques).
* E n’est utilisé que lors de la compilation, et disparaît à la fin de celle-ci. E n’existe donc plus lors de l'exécution.
* En C++, le code généré est dupliqué. Typiquement, pour un Sequence<Integer> et un Sequence<String>, il y aurait deux classes générées

⇒ solution : utiliser object

* Object[] elements = new Object[100]

**@SuppressWarnings(“unchecked”)**

public E get(int index)

{

return **cast**element[index];

}

Sequence<Integer> integerSequence = new Sequence<>();

(pas obligé de répéter l’intérieur du diamond)

* Integer sous classe d’Object ⇒ Sequence<Integer> sous classe de Sequence<Object>

ObjectSequence = IntegerSequence; (pointe/référence)

ObjectSequence.add(“foo”); ⇒ *authorisé mais après ça ne marche pas.*

* **FAUX.** Raison : sécurité des types.
* nomClass<E,K,L>

⇒ classes paramétrées par 3 types.

* On peut paramétrer des méthodes dans des classes non paramétrées

class Foo {

public <T> T get(int i)

[...]

int k = foo.get(0); *//le compilateur comprendra qu’il faut mettre un objet du type Integer*

}

#### 1.3 max

* interface Comparable**<E>** {

int compareTo(Object **<E> (à la place de Object)** other);

}

obj1.compareTo(obj2);

(avant java 1.5)

**depuis java 1.5**

**⇒ erreur**

Il faut pouvoir y mettre des objets comparables. Il faut donc faire un extend. Préciser que E doit être une sous classe de comparable.

* **Sequence <E extends Comparable<E>>**

ctrl + shift + i ⇒ ajoute les imports manquants & retire les imports superflux

#### 1.4 removeAll

dans une séquence d’objets, il peut y avoir des entiers. On peut vouloir retirer les entiers de cette séquence.

public boolean removeAll(Sequence <**?**> other)

⇒ et non pas Sequence <E> other (pas assez général)

⇒ et non pas Sequence <Object> other (raisons de sécurité vues plus haut)

integerSequence.removeAll(Object Sequence) (ou (Integer Sequence))

## Annotations

**annotation =** meta donné dans le code

**ex :** @Override

@Derecated : la méthode ne doit plus être utilisée → erreur

@SuppressWarnings

* Qualifient des méthodes, des classes ou des attributs
* On peut même désormais préciser l’utilisation de certains types
* utilisées par le compilateur
* Permettent de prévenir les problèmes de typo (ex : toString / ToString)

Il y a également des annotations qui peuvent être utilisées lors de l’exécution.

Vont dans le bytecode; transportées, dispo dans l’exécutable

@TestSpec

Définition d’une annotation dans le cas général : sous forme de pseudo interface.

Derrière une annotation, il peut y avoir plusieurs valeurs.

public @interface Toto {

int valeur1() default - 1;

String valeur2();

}

class Truc{

@Toto (valeur1 = 2, valeur2 = “machin”)

public void methode1()

{

}

}

class getAnnotations ⇒ récupère les valeurs & annotations utilisées

Il existe également des annotations pour décorer d’autres annotations. Définies dans java.lang.annotations. Celles-ci permettent de spécifier la rétention d’une annotation.

ex : une annotation doit survivre à la compilation

**@Retention(value = RUNTIME)**

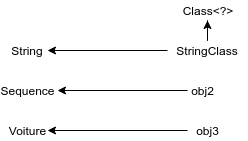
spécifier si l’annotation porte sur une méthode, une classe ou les 2

**@Target(value=...)**

## Reflexion

reflexion = capacité d’un programme à se voir lui-même, à se connaître

Il y a un méta attribut statique redéfini pour chaque class



1. Class stringClass = String.class;
2. String s = “abc”;

stringClass s.getClass();

**Utilisation :**

stringClass.getMethods...();

⇒ on prend connaissance de nos composants. On aboutit à quelque chose de très flexible, qui permet de prendre connaissance des composantes d’un objet && d’appeler dynamiquement des méthodes dessus.