EISTI
Ecole Internationale des Sciences du Traitement de l'Information

PROGRAMMATION GÉNÉRIQUE

Cours 12

1 - Le problème

- Soit une collection particulière d'objets :

```
Vector v = new Vector()
```

implémentée par un développeur pour y stocker des objets de classe A.

Le développeur ajoute régulièrement des objets A à cette collection :

```
A a = new A();
v.add(a)
```

2 - Le problème (2)

- Le vecteur enregistre tous ces objets dans un `Object[]` interne qu'il gère à sa manière.
- Tant que le développeur met des A explicitement, il sait (à peu près) ce qu'il fait.
- Mais il peut mettre des A indirectement :

```
Object o = aBlob.getSomeThingFromBlob(); // un B
v.add(o);
```

- Comment le développeur peut-il savoir à l'avance que cette utilisation est erronée ?

3 - Le principe générique

- La programmation générique répond à ce problème, en permettant d'écrire des modèles de classe dont on peut forcer le « type » d 'utilisation.
- Si le Vector était un « Vecteur de A », le compilateur aurait pu voir le problème

Rappel : « *The compiler is your friend* »

4 - Le principe générique (2)

- Ecrire une classe une fois, puis paramétrer quel type d'objets elle manipule.
- Très utile pour les collections, afin d 'obtenir des « ensemble de », « listes de », etc.
- La programmation générique est généralisable à toute classe.

5 - Type générique

- Un type générique est une « classe » paramétrée.

```
class BoiteGenerique<T>{
    private T t;
    public BoiteGenerique(T t){
        this.t = t;
    }
    public T getContent(){
       return t;
    }
}
```

- Chaque fois qu'on voit T, on peut imaginer de le remplacer par n'importe quelle classe Java.

6 - Invocation d'un type générique

- On invoque un type en précisant pour quelle classe on l'utilise :

```
BoiteGenerique<Integer> bgi = new
        BoiteGenerique<Integer>(new
                                Integer(4));
```

Une fois obtenue l'instance, la méthode `getContent()` renvoie un `Integer` :

```
Integer i = bgi.getContent();
```

7 - Conventions

- La lettre T s'appelle un « paramètre de type »
- On admet certaines conventions lorsqu'on choisit une notation pour les paramètres de type :
  - E - Elément (utilisé dans les collections Java)
  - K - Clef
  - N - Nombre
  - T - Type
  - V - Valeur
  - S,U,V etc. - 2nd, 3ème, 4ème types.

8 - Méthode générique

- Le principe de la programmation générique peut être localisée à une méthode :

```
public <U> void putInBox(U u,
  BoiteGenerique<U>){
  ...
}
```

Cette méthode, placée dans une classe tout à fait normale, permet de mettre un objet de type variable dans une boîte appropriée.

9 - Méthode générique (2)

- Utiliser une méthode générique :
  - Pour utiliser la méthode précédente

```
public <U> void putInBox(U u,
  BoiteGenerique<U>){
  ...
}
```

il faudrait écrire :

```
<Crayon>putInBox(crayonRouge, trousse);
```

10 - Méthode générique (3)

- Simplification des appels
  - en fait, on admet que l'on puisse simplement écrire :

```
putInBox(crayonRouge, trousse);
```

cette simplification s'appelle

« inférence de type »

11 - Multigénéricité

- On peut utiliser plusieurs paramètres de type dans la même construction générique :

```
Mapping<String,String> m;
```

- associe nécessairement des chaînes de caractères à des chaînes de caractères et serait une utilisation d'une instance d'une classe :

```
class Mapping<U, V>{
  ... U ... V ...
}
```

12 - Multigénéricité (2)

- La généricité peut s'utiliser en cascade :
  - Si Boite est une classe générique

    ```
    class Boite<T>{...
    ```

  - et List est une classe générique

    ```
    class List<T>{…
    ```

  - alors on peut utiliser des listes de boîtes

    ```
    List<Boite<Crayon>> trousses = ...
    ```

13 - Multigénéricité (3)

– Dans l'expression :

```
List<Boite<Crayon>> trousses = ...
```

Le compilateur peut vérifier une double contrainte :

- que la Boîte utilisée contient des crayons,
- que la List utilisée contient bien des « Boîtes de crayons »

14 - Types contraints

- Problème :
  - Peut-on imposer la nature des classes qui sont utilisables au moment d'exploiter un type paramétrique ?

```
public <T> void metGen(T t){ … }
```

admet un paramètre de type, mais n'impose rien sur la nature de T. On voudrait interdire l'utilisation de certaines classes au moment de l'utilisation.

15 - Types contraints (2)

- Solution :
  - Contraindre en imposant la classe de départ :

```
public <T extends Number> void
        metGen(T t){ … }
```

n'admet à l'utilisation que Number ou ses sous-types.

```
metGen("10");
```

échoue à la compilation (utilisation d'une String pour T).

16 - Types contraints (3)

- Contraintes multiples :
  - Pour ajouter une interface obligatoire à la contrainte :

```
<T extends SuperType & Interface1>
```

17 - Génériques et héritage

- 1 - Pour des classes courantes :

```
Object o = new Object();
Integer i = new Integer(1);
o = i ; // Valide
```

- 2 - Pour des génériques :

```
Boite<Number> bn1 = new
   Boite<Number>(new Integer(1));
Boite<Number> bn2 = new
   Boite<Number>(new Double(3.14));
```

18 - Génériques et héritage (2)

- 3 - Le type générique comme un type à part entière ?
  - Examinons la méthode :

```
public void inspect(Boite<Number> bn);
```

  - Peut-on l'utiliser avec un type **`Boite<Integer>`**, puisque **`Integer`** est sous-classe de **`Number`** ?

19 - Génériques et héritage (3)

- NON :

« Un type G<U> ne vaut pas pour un type G<T>, même si U vaut pour un T. »

20 - Jokers de type

- Nous revenons au problème des contraintes.
  - Peut-on dire pour une déclaration de classe générique : utilise uniquement cette sous-famille d'objets ?

```
class Boite<? extends Number>{}
```

Seuls Number et ses sous-classes peuvent être utilisées avec la boîte :

```
Boite<Integer> bi = new Boite<Integer>(1); // légal
Boite<String> bs = new Boite<String>("1"); // illégal
```

21 - Jokers de type

- Nous revenons au problème des contraintes.
  - Peut-on dire pour une déclaration de classe générique : utilise uniquement des classes très générales (au dessus de la classe C ou C)

```
class Boite<? super C>{}
```

Alors :

```
class C extends B{…}
class D extends C{…}
Boite<B> bb = new Boite<B>(); // légal
Boite<C> bc = new Boite<C>(); // légal
Boite<D> bd = new Boite<D>(); // illégal
```

22 - Jokers de type

- Ressemblances :

```
class Boite<T extends Number>{…}
class Boite<? extends Number>{…}
```

  - Dans les deux cas, l'usage du générique est restreint à la seule descendance de Number
- Différences :
  - Boite<Integer> n'est pas sous-classe de Boite<Number>
  - Boite<Integer> est sous-type de Boite<? extends Number>, mais avec certaines restrictions d'usage.

23 - Effacement du type

- Est nommé « effacement du type » la technique qui permet au compilateur de produire une classe compilée résolue, à partir d'un générique.
- La classe produite est compatible avec des machines virtuelles ne connaissant pas les génériques.

24 - Effacement du type (2)

- Toute référence explicite au type ne peut être compilé :

```
public class MyClass<E> {
    public static void myMethod(Object item) {
        if (item instanceof E) {  //Compiler error
            ...
        }
        E item2 = new E();   //Compiler error
        E[] iArray = new E[10]; //Compiler error
        E obj = (E)new Object(); //Unchecked cast warning
    }
}
```

- E n'est pas une classe. Le terme E disparaît dans le code compilé.

EISTI
Ecole Internationale des Sciences du Traitement de l'Information

INTRODUCTION À JAVA 8

Cours 12

ÉVOLUTION DE JAVA

- Java 8 introduit plusieurs nouvelles fonctionnalités
- Méthodes par défaut
- Expressions Lambda
- JavaFX inclut dans cette version (applications Web en Java)
- Nouvelles classes (Date, Time, ...)
- Gestion des flux (Streams)
- Concurrence / Sécurité

MÉTHODES D'EXTENSION

- Nouveau mot-clé : default
- Fournit une implémentation par défaut

```
interface Formula {
    double calculate(int a);
    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}

Formula formula = new Formula() { ... };
formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

Expressions Lambda

- Une expression Lambda est une fonction anonyme permettant de passer des méthodes en paramètres

```
(paramètres) -> { code }
(int a, int b) -> { return a > b }
```

- Une expression Lambda peut avoir 0+ paramètres
- Fonctionne pour les interfaces avec une seule méthode à implémenter
- Ces interfaces peuvent porter l'annotation @FunctionalInterface

EXPRESSIONS LAMBDA

- Trier une liste en Java 7
- List<String> names = Arrays.asList("Nom1", "Nom2", "Nom3");

```
 Collections.sort(names, new Comparator<String>() {
@Override
public int compare(String a, String b) {
        return b.compareTo(a);
        }
} );
```

EXPRESSIONS LAMBDA

- Trier une liste en Java 8
- List<String> names =
  Arrays.asList("Nom1", "Nom2", "Nom3");

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});

//ou
Collections.sort(names, (String a, String b) -> b.compareTo(a));

//ou
Collections.sort(names, (a, b) -> b.compareTo(a));
```

PASSAGE DE RÉFÉRENCES

- Le mot-clé :: permet de passer des références de méthodes

- Something something = **new** Something();
  Converter<String, String> **converter** = something**::startsWith**;
  String converted = **converter.convert**("**Java**");
  System.***out***.println(converted); *// "J"*

- PersonFactory<Personne> personFactory **= Personne::new;**
- Personne person **=** personFactory**.**create**(**"Prénom"**,** "Nom"**);**

NOUVELLES INTERFACES

- Java 8 inclut plusieurs nouvelles classes et interfaces
- Predicate – Expression booléennes (un paramètre)
- Function – Functions à appeler (un paramètre)
- Supplier – Semblable à une fonction (sans paramètre)
- Consumer – Opérations à appliquer sur un paramètre
- Comparator – Comparer des objets
- Optional – Indique quoi faire pour éviter les NullPointerExceptions

PRÉDICATS / FONCTIONS

- ```
  Predicate<String> predicate = (s) -> s.length() > 0;
  predicate.test("foo");              // true
  predicate.negate().test("foo");     // false
  ```

- ```
  Function<String, Integer> toInteger = Integer::valueOf;
  Function<String, String> backToString =
                          toInteger.andThen(String::valueOf);
  backToString.apply("123");    // "123"
  ```

SUPPLIERS / CONSUMERS

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();   // new Person

Consumer<Person> greeter = (p) ->
System.out.println("Hello, " + p.nom);
greeter.accept(new Person("Luke Skywalker"));
```

COMPARATORS

```
Comparator<Person> comparator =
        (p1, p2) -> p1.nom.compareTo(p2.nom);

Person p1 = new Person("John Doe");
Person p2 = new Person("Alice Wonderland");

comparator.compare(p1, p2);              // > 0
comparator.reversed().compare(p1, p2);   // < 0
```

OPTIONAL

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();                   // true
optional.get();                         // "bam"
optional.orElse("fallback");            // "bam"

optional.ifPresent( (s) -> System.out.println(s.charAt(0)) );
// => "b"
```

STREAMS

- Les Streams (Flux) en Java 8 représentent une séquence d'éléments
- On peut appliquer des opérations sur ces éléments

- À partir d'une collection typique, il faut appeler la méthode **stream** pour en récupérer le flux
- List<String> stringCollection = new ArrayList<>();
- stringCollection.stream()

STREAMS

- stringCollection
  .stream()
  .sorted()
  .filter((s) -> s.startsWith("a"))
  .forEach(System.*out*::println);
- *// "aaa1", "aaa2"*
- La liste est triée
- Les éléments ne débutant pas par "a" sont exclus
- Les éléments sont parcourus puis affichés
- La liste originale n'est pas modifiée
- System.*out*.println(stringCollection);
  // ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

STREAMS

- Convertir des éléments d'une liste avec **map**
- stringCollection

```
        .stream()
        .map(String::toUpperCase)
        .sorted((a, b) -> b.compareTo(a))
        .forEach(System.out::println);
// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2",
"AAA1"
```

- Les flux peuvent être séquentiels .stream() ou parallèles .parallelStream() afin d'accélérer certains traitements comme des tris

GESTION DU TEMPS

- Nouvelles classes pour gérer le temps, les dates
- Clock clock = Clock.*systemDefaultZone*();
  long millis = clock.millis();

  Instant instant = clock.instant();
  Date legacyDate = Date.*from*(instant); *// legacy java.util.Date*

GESTION DU TEMPS

```
ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");

LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false
```

GESTION DU TEMPS

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014,
Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek);    // FRIDAY
```

GESTION DU TEMPS

```
DateTimeFormatter formatter =
        DateTimeFormatter
                .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.
                        parse("Nov 03, 2014 - 07:13", formatter);
String string = formatter.format(parsed);

System.out.println(string);    // Nov 03, 2014 - 07:13
```