Université Sultan Moulay Slimane
Ecole Supérieure de Technologie Béni Mellal
Département ITG

DUT- IDIA 1 - S2

M122 - Ingénierie Logicielle avec Java et UML

Année universitaire: 2024-2025

Support de cours

Ingénierie logicielle avec Java et UML

Partie I: Programmation Orientée Objet en Java

Chapitre V: La Généricité en Java

Pr. Siham BAKKOURI
Email: s.bakkouri@usms.ma

Plan

- **INTRODUCTION**
- **PRINCIPE DE BASE DE LA GÉNÉRICITÉ**
- **LES CLASSES ET LES MÉTHODES GÉNÉRIQUES**
- **GÉNÉRICITÉ POLYMORPHIQUE**
- **LES WILDCARDS (?)**
- **EXERCICES**

Introduction

- La **généricité** en Java permet de créer des classes, interfaces et méthodes qui peuvent fonctionner avec différents types de données tout en garantissant la sécurité de type au moment de la compilation.
- Elle offre la possibilité de créer des composants réutilisables capables de traiter des types différents sans sacrifier la sûreté de typage.

La **généricité** a été introduite dans Java avec le **JDK** version 1.5 (Java 5).

Principe de base

Les principaux points à retenir sur la généricité en Java sont :

- **Déclaration de types génériques** : Les types génériques sont définis à l'aide des chevrons < >. Par exemple, List<T>, Map<K, V>, où T, K et V sont des paramètres de type.
- **Réutilisabilité et flexibilité** : La généricité permet de créer des classes, interfaces et méthodes génériques, ce qui favorise la réutilisation du code pour différents types de données.
- **Sûreté de typage (type-safety)** : Les génériques offrent une sécurité de type accrue en détectant les erreurs de typage au moment de la compilation plutôt qu'à l'exécution, ce qui évite les ClassCastException.

Principe de base

Les principaux points à retenir sur la généricité en Java sont :

- **Évite les castings explicites** : En utilisant des génériques, les castings redondants et potentiellement risqués peuvent être évités lors de l'utilisation de collections ou de structures de données.
- **Méthodes génériques** : Les méthodes peuvent être génériques, ce qui signifie qu'elles peuvent accepter différents types de paramètres sans perdre la sûreté de typage.
- **Wildcards (?)** : Les **wildcards** permettent de spécifier des paramètres de types inconnus dans les génériques, offrant ainsi une plus grande flexibilité lors de la déclaration de types.

Les classes et les méthodes génériques

→ Classe générique

- Une **classe générique** utilise des paramètres de type (<T>, <K, V>).

Exemple (1/2):

```
// Définition de la classe Boite de manière générique
 public class Boite<T> {
    private T contenu;

    public void mettre(T objet) {
        this.contenu = objet;
    }

    public T obtenir() {
        return contenu;
    }
}
```

Les classes et les méthodes génériques

Exemple (2/2):

```
public class Main {
    public static void main(String[] args) {
        // Création d'une boîte pour stocker des entiers
        Boite<Integer> boiteEntiers = new Boite<>();
        boiteEntiers.mettre(10);

        // Obtention de la valeur stockée dans la boîte à entiers
        int valeurEntiere = boiteEntiers.obtenir();
        System.out.println("Valeur dans la boîte à entiers : " + valeurEntiere);

        // Création d'une boîte pour stocker des chaînes de caractères
        Boite<String> boiteChaines = new Boite<>();
        boiteChaines.mettre("Bonjour, monde!");

        // Obtention de la valeur stockée dans la boîte à chaînes
        String valeurChaine = boiteChaines.obtenir();
        System.out.println("Valeur dans la boîte à chaînes : " + valeurChaine);
    }
}
```

→ Méthode générique

- Une **méthode générique** permet d'utiliser un type générique sans rendre la classe entière générique.

Exemple :

```
class Utilitaire {
    // Méthode générique qui affiche le type d'un élément
    public static <T> void afficherType(T element) {
        System.out.println("Type : " + element.getClass().getName());
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Utilitaire.afficherType(42);
        Utilitaire.afficherType("Type 1");
    }
}
```

Généricité polymorphique

- La "**généricité multiple**" ou "**généricité polymorphique**" fait référence à la capacité d'une classe ou d'une méthode à utiliser plusieurs types génériques simultanément.
- En Java, cela est réalisé par le biais des génériques, où une classe ou une méthode peut accepter plusieurs types de paramètres génériques.

Exemple (1/3):

Supposons que nous voulions créer une classe Couple qui peut stocker une paire de deux éléments de types différents de manière générique.

```
public class Couple<T, U> {
    private T premier;
    private U deuxieme;
   // .....
}
```

Généricité polymorphique

Exemple (1/3)

```
public class Couple<T, U> {
    private T premier;
    private U deuxieme;

    public Couple(T premier, U deuxieme) {
        this.premier = premier;
        this.deuxieme = deuxieme;
    }

    public T getPremier() {
        return premier;
    }

    public U getDeuxieme() {
        return deuxieme;
    }

    public void afficherCouple() {
        System.out.println("(" + premier + ", " + deuxieme + ")");
    }
}
```

Généricité polymorphique

Exemple (3/3)

```java
public class Main {
    public static void main(String[] args) {
        // Création d'une instance de Couple avec types génériques différents
        Couple<String, Integer> coupleStringInt = new Couple<>("Bonjour", 2023);
        coupleStringInt.afficherCouple();

        // Accès aux éléments individuels du couple
        String premierElement = coupleStringInt.getPremier();
        int deuxiemeElement = coupleStringInt.getDeuxieme();

        System.out.println("Premier élément : " + premierElement);
        System.out.println("Deuxième élément : " + deuxiemeElement);
    }
}
```

- La classe Couple utilise deux types génériques **T** et **U** pour représenter les deux éléments du couple.
- Lors de l'instanciation de la classe **`Couple`**, nous spécifions les types concrets que nous voulons utiliser (**String** et **Integer** dans cet exemple).

Wildcards (?)

Le **wildcard** (**?**) est utilisé dans les génériques Java pour représenter un type inconnu.

- **Limitations du wildcard** :
  - Restreint les opérations sur les collections où il est appliqué.
  - Convertit les collections en **lecture seule** dès son application.
  - Limite l'ajout d'éléments à la collection en raison de l'incertitude quant au type exact des éléments ajoutables.
- **Conséquences de son utilisation :**
  - L'incapacité d'ajouter de nouveaux éléments à la collection survient dès l'application du wildcard.
  - Garantit la sécurité de typage en empêchant les insertions non typées, mais restreint également la mutabilité de la collection.

Wildcards (?)

Exemple (1/2):

Supposons que nous ayons une classe **`Imprimante`** qui peut imprimer des éléments d'une liste.

→ le **wildcard** permettrait à notre méthode d'accepter une liste d'éléments de type inconnu, offrant une flexibilité lors de l'appel de cette méthode avec différents types de listes.

```
import java.util.List;

public class Imprimante {
    public static void imprimerListe(List<?> liste) {
        for (Object element : liste) {
            System.out.println(element);
        }
    }
}
```

Wildcards (?)

Exemple (2/2):

- La méthode **`imprimerListe`** accepte une liste d'éléments de type inconnu grâce au wildcard (**`List<?>`**). Elle parcourt ensuite la liste et imprime chaque élément.
- L'utilisation du wildcard permet à la méthode **`imprimerListe`** d'accepter des listes contenant n'importe quel type d'objet (Integer, String, etc.). Cela offre une grande flexibilité lors de l'appel de cette méthode avec différents types de listes sans sacrifier la sûreté de typage.

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        // Exemple avec une liste d'entiers
        List<Integer> listeEntiers = List.of(1, 2, 3, 4, 5);
        System.out.println("Impression de la liste d'entiers :");
        Imprimante.imprimerListe(listeEntiers);

        // Exemple avec une liste de chaînes de caractères
        List<String> listeChaines = List.of("Java", "est", "fantastique");
        System.out.println("\nImpression de la liste de chaînes :");
        Imprimante.imprimerListe(listeChaines);
    }
}
```

Exercices

Exercice 1

1. Crée une classe générique ***Message<T>*** qui stocke un message de n'importe quel type.
2. Implémente un constructeur, une méthode ***afficher***(), et un getter.
3. Teste avec un String et un Integer.

→ Correction

Exercice 2

1. Crée une classe générique ***Score<T, U>*** qui associe un ***nom*** (***T***) à un ***score*** (***U***).
2. Implémente un constructeur, une méthode ***afficher***(), et des getters.
3. Teste avec String et Double.

→ Correction

ANNEXE

Exercice 1 (1/2)

```
class Message<T> {
    private T contenu;

    public Message(T contenu) {
        this.contenu = contenu;
    }

    public T getContenu() {
        return contenu;
    }

    public void afficher() {
        System.out.println("Message : " + contenu);
    }
}
```

Exercice 1 (2/2)

```
public class Main {
    public static void main(String[] args) {
        // Test avec un String
        Message<String> msgTexte = new Message<>("Bonjour !");
        msgTexte.afficher(); // Affiche : Message : Bonjour !

        // Test avec un Integer
        Message<Float> msgNombre = new Message<>(2024);
        msgNombre.afficher(); // Affiche : Message : 2024
    }
}
```

Exercice 2 (1/2)

```
class Score<T, U> {
    private T nom;
    private U valeur;

    public Score(T nom, U valeur) {
        this.nom = nom;
        this.valeur = valeur;
    }

    public T getNom() {
        return nom;
    }

    public U getValeur() {
        return valeur;
    }

    public void afficher() {
        System.out.println(nom + " a un score de " + valeur);
    }
}
```

Exercice 2 (2/2)

```java
public class Main {
    public static void main(String[] args) {
        // Test avec un String (nom) et un Double (score)
        Score<String, Double> joueur = new Score<>("Ali", 95.5);
        joueur.afficher(); // Affiche : Ali a un score de 95.5
    }
}
```