CodInClub

Powered By BridgeLabz

Best Practices for Java Generics

Using **Generics** effectively ensures **type safety, reusability, and maintainability** in Java applications. Below are the key best practices to follow:

---

1. Use Generics to Ensure Type Safety

- Prevents `ClassCastException` at runtime.
- Ensures type checking at **compile-time** rather than runtime.

---

2. Prefer Generic Methods Over Overloading

- Reduces redundancy by allowing **a single method** to handle multiple data types.
- Improves **code reusability** without requiring multiple overloaded methods.

---

3. Use Upper Bounded Wildcards (? extends T) for Read-Only Access

- Allows **reading** elements from a collection without modification.
- Useful when working with **inherited types** to ensure flexibility.

---

4. Use Lower Bounded Wildcards (? super T) for Write Operations

- Allows **modifying** a collection while maintaining compatibility with **superclasses**.
- Prevents unintended operations that could introduce type mismatch errors.

---

5. Avoid Using Raw Types (List Instead of List<T>)

- Raw types bypass **type safety**, leading to **unchecked warnings** at compile-time.

- Always use **parameterized types** (`List<String>`, `List<Integer>`) instead of raw `List`.

---

6. Use Bounded Type Parameters for Restriction (<T extends SomeClass>)

- Restricts **type parameters** to a specific class or interface.
- Ensures **only valid types** can be used with a generic class or method.

---

7. Favor Generic Interfaces for Common Behaviors

- Improves **code reuse** by defining a common behavior for **multiple implementations**.
- Helps in designing **flexible APIs** that work with different data types.

---

8. Minimize Wildcard Usage in Public APIs

- Use wildcards (`? extends T`, `? super T`) **only when necessary** to improve API flexibility.
- Avoid wildcards in **method return types**, as it complicates type inference.

---

9. Combine Generics with Functional Interfaces and Streams

- Works well with **Java Streams API** for **processing collections dynamically**.
- Improves **readability** and **efficiency** in functional-style programming.

---

10. Use Generic Constructors Where Necessary

- Allows creating **type-safe** instances in a **flexible** way.
- Improves **encapsulation** while maintaining **generic behavior**.

---

11. Avoid Type Erasure Pitfalls

- Remember that **type parameters do not exist at runtime** due to **Type Erasure**.
- Cannot use **`instanceof`** with generic type parameters (`T`), as type information is erased.

---

12. Favor Composition Over Inheritance in Generic Hierarchies

- Reduces **complexity** by avoiding deep inheritance chains.
- Enhances **maintainability** and **flexibility** by composing objects rather than inheriting them.

---

13. Keep Generics Simple and Understandable

- Avoid **overly complex** generic hierarchies.
- Use meaningful type parameter names (`T`, `E`, `K`, `V`) to improve **code readability**.

1. Smart Warehouse Management System

Concepts: Generic Classes, Bounded Type Parameters, Wildcards

Problem Statement:

You are developing a **Smart Warehouse System** that manages different types of items like **Electronics, Groceries, and Furniture**. The system should be able to store and retrieve items dynamically while maintaining type safety.

Hints:

- Create an **abstract class** **`WarehouseItem`** that all items extend (`Electronics`, `Groceries`, `Furniture`).
- Implement a **generic class** `Storage<T extends WarehouseItem>` to store items safely.
- Implement a **wildcard method** to display all items in storage regardless of their type (`List<? extends WarehouseItem>`).

```
import java.util.*;

abstract class WarehouseItem {
    String name;
    WarehouseItem(String name) { this.name = name; }
    public String toString() { return name; }
}

class Electronics extends WarehouseItem {
    Electronics(String name) { super(name); }
}

class Groceries extends WarehouseItem {
    Groceries(String name) { super(name); }
}

class Furniture extends WarehouseItem {
    Furniture(String name) { super(name); }
}

class Storage<T extends WarehouseItem> {
    private List<T> items = new ArrayList<>();
    void addItem(T item) { items.add(item); }
```

```
    List<T> getItems() { return items; }
}

public class SmartWarehouse {
    public static void displayAllItems(List<? extends WarehouseItem> items)
{
        for (int i = 0; i < items.size(); i++) {
            System.out.println(items.get(i));
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Storage<Electronics> electronicsStorage = new Storage<>();
        Storage<Groceries> groceriesStorage = new Storage<>();
        Storage<Furniture> furnitureStorage = new Storage<>();

        System.out.println("Enter number of electronics items:");
        int e = sc.nextInt(); sc.nextLine();
        for (int i = 0; i < e; i++) electronicsStorage.addItem(new
Electronics(sc.nextLine()));

        System.out.println("Enter number of groceries items:");
        int g = sc.nextInt(); sc.nextLine();
        for (int i = 0; i < g; i++) groceriesStorage.addItem(new
Groceries(sc.nextLine()));

        System.out.println("Enter number of furniture items:");
        int f = sc.nextInt(); sc.nextLine();
        for (int i = 0; i < f; i++) furnitureStorage.addItem(new
Furniture(sc.nextLine()));

        System.out.println("\nElectronics:");
        displayAllItems(electronicsStorage.getItems());
        System.out.println("Groceries:");
        displayAllItems(groceriesStorage.getItems());
        System.out.println("Furniture:");
        displayAllItems(furnitureStorage.getItems());
    }
}
```

2. Dynamic Online Marketplace

Concepts: Type Parameters, Generic Methods, Bounded Type Parameters

Problem Statement:

Build a **generic product catalog** for an online marketplace that supports various product types like **Books, Clothing, and Gadgets**. Each product type has a **specific price range and category**.

Hints:

- Define a **generic class `Product<T>`** where `T` is restricted to a category (`BookCategory`, `ClothingCategory`, etc.).
- Implement a **generic method** to apply discounts dynamically (`<T extends Product> void applyDiscount(T product, double percentage)`).
- Ensure type safety while allowing **multiple product categories** to exist in the same catalog.

```java
import java.util.*;

interface Category {}

class BookCategory implements Category {}
class ClothingCategory implements Category {}
class GadgetCategory implements Category {}

class Product<T extends Category> {
    String name;
    double price;
    Product(String name, double price) {
        this.name = name;
        this.price = price;
    }
```

```java
    public String toString() { return name + ": $" + price; }
}

public class OnlineMarketplace {
    public static <T extends Product<?>> void applyDiscount(T product,
double percentage) {
        product.price -= product.price * (percentage / 100);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Product<BookCategory> book = new Product<>("Java Book", 500);
        Product<ClothingCategory> shirt = new Product<>("Shirt", 1000);
        Product<GadgetCategory> phone = new Product<>("Smartphone", 15000);

        System.out.print("Enter discount for book: ");
        applyDiscount(book, sc.nextDouble());
        System.out.print("Enter discount for shirt: ");
        applyDiscount(shirt, sc.nextDouble());
        System.out.print("Enter discount for gadget: ");
        applyDiscount(phone, sc.nextDouble());

        System.out.println("\nDiscounted Products:");
        System.out.println(book);
        System.out.println(shirt);
        System.out.println(phone);
    }
}
```

---

3. Multi-Level University Course Management System

Concepts: Generic Classes, Wildcards, Bounded Type Parameters

Problem Statement:

Develop a **university course management system** where different departments offer courses with **different evaluation types** (e.g., **Exam-Based, Assignment-Based, Research-Based**).

Hints:

- Create an **abstract class** **`CourseType`** (e.g., `ExamCourse`, `AssignmentCourse`, `ResearchCourse`).
- Implement a **generic class** **`Course<T extends CourseType>`** to manage different courses.
- Use **wildcards** (**`List<? extends CourseType>`**) to handle **any type of course** dynamically.

```
import java.util.*;

abstract class CourseType {
    String name;
    CourseType(String name) { this.name = name; }
    public String toString() { return name; }
}

class ExamCourse extends CourseType {
    ExamCourse(String name) { super(name); }
}

class AssignmentCourse extends CourseType {
    AssignmentCourse(String name) { super(name); }
}

class ResearchCourse extends CourseType {
    ResearchCourse(String name) { super(name); }
}

class Course<T extends CourseType> {
    private List<T> courseList = new ArrayList<>();
    void addCourse(T course) { courseList.add(course); }
    List<T> getCourses() { return courseList; }
}

public class UniversitySystem {
    public static void displayCourses(List<? extends CourseType> list) {
        for (int i = 0; i < list.size(); i++) {
```

```
            System.out.println(list.get(i));
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Course<ExamCourse> examCourses = new Course<>();
        Course<AssignmentCourse> assignCourses = new Course<>();

        System.out.println("Enter number of exam courses:");
        int ex = sc.nextInt(); sc.nextLine();
        for (int i = 0; i < ex; i++) examCourses.addCourse(new
ExamCourse(sc.nextLine()));

        System.out.println("Enter number of assignment courses:");
        int as = sc.nextInt(); sc.nextLine();
        for (int i = 0; i < as; i++) assignCourses.addCourse(new
AssignmentCourse(sc.nextLine()));

        System.out.println("Exam Courses:");
        displayCourses(examCourses.getCourses());
        System.out.println("Assignment Courses:");
        displayCourses(assignCourses.getCourses());
    }
}
```

---

4. Personalized Meal Plan Generator

Concepts: Generic Methods, Type Parameters, Bounded Type Parameters

Problem Statement:

Design a **Personalized Meal Plan Generator** where users can choose **different meal categories** like **Vegetarian, Vegan, Keto, or High-Protein**. The system should ensure **only valid meal plans are generated**.

Hints:

- Define an **interface** **`MealPlan`** with subtypes (`VegetarianMeal`, `VeganMeal`, etc.).
- Implement a **generic class** **`Meal<T extends MealPlan>`** to handle different meal plans.
- Use a **generic method** to validate and generate a personalized meal plan dynamically.

import java.util.*;

```
interface MealPlan {
    String getDetails();
}
class VegetarianMeal implements MealPlan {
    public String getDetails() { return "Vegetarian Plan: Includes fruits,
grains, legumes."; }
}
class VeganMeal implements MealPlan {
    public String getDetails() { return "Vegan Plan: No animal products.";
}
}
class KetoMeal implements MealPlan {
    public String getDetails() { return "Keto Plan: High fat, low carbs.";
}
}
class Meal<T extends MealPlan> {
    T mealType;
    Meal(T mealType) { this.mealType = mealType; }
    void generatePlan() { System.out.println(mealType.getDetails()); }
}
public class MealPlanner {
    public static <T extends MealPlan> void generatePersonalizedPlan(T
meal) {
        Meal<T> m = new Meal<>(meal);
        m.generatePlan();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Choose Meal Type: 1. Vegetarian 2. Vegan 3.
Keto");
        int choice = sc.nextInt();
```

```
        switch (choice) {
            case 1: generatePersonalizedPlan(new VegetarianMeal()); break;
            case 2: generatePersonalizedPlan(new VeganMeal()); break;
            case 3: generatePersonalizedPlan(new KetoMeal()); break;
            default: System.out.println("Invalid choice.");
        }
    }
}
```

---

5. AI-Driven Resume Screening System

Concepts: Generic Classes, Generic Methods, Bounded Type Parameters, Wildcards

Problem Statement:

Develop an **AI-Driven Resume Screening System** that can process resumes for **different job roles** like **Software Engineer, Data Scientist, and Product Manager** while ensuring type safety.

Hints:

- Create an **abstract class** **`JobRole`** (`SoftwareEngineer`, `DataScientist`, `ProductManager`).
- Implement a **generic class** **`Resume<T extends JobRole>`** to process resumes dynamically.
- Use a **wildcard method** (**`List<? extends JobRole>`**) to handle multiple job roles in the screening pipeline.

```
import java.util.*;

abstract class JobRole {
    String candidate;
    JobRole(String candidate) { this.candidate = candidate; }
    public String toString() { return candidate; }
}

class SoftwareEngineer extends JobRole {
    SoftwareEngineer(String candidate) { super(candidate); }
}

class DataScientist extends JobRole {
    DataScientist(String candidate) { super(candidate); }
}

class ProductManager extends JobRole {
    ProductManager(String candidate) { super(candidate); }
}

class Resume<T extends JobRole> {
    T role;
    Resume(T role) { this.role = role; }
    void process() {
        System.out.println("Screening resume for: " + role);
    }
}

public class ResumeScreening {
    public static void screenAll(List<? extends JobRole> list) {
        for (int i = 0; i < list.size(); i++) {
            System.out.println("Screening: " + list.get(i));
        }
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        List<SoftwareEngineer> seList = new ArrayList<>();
        List<DataScientist> dsList = new ArrayList<>();

        System.out.println("Enter number of software engineer
```

```
applicants:");
        int se = sc.nextInt(); sc.nextLine();
        for (int i = 0; i < se; i++) seList.add(new
SoftwareEngineer(sc.nextLine()));

        System.out.println("Enter number of data scientist applicants:");
        int ds = sc.nextInt(); sc.nextLine();
        for (int i = 0; i < ds; i++) dsList.add(new
DataScientist(sc.nextLine()));

        System.out.println("Screening Software Engineers:");
        screenAll(seList);
        System.out.println("Screening Data Scientists:");
        screenAll(dsList);
    }
}
```