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>`).

---

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.

---

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.

---

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.

---

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.