1. Introduction to Generics

Java **Generics** allow developers to create **type-safe, reusable, and flexible** code by introducing **parameterized types**. Instead of using raw types (like `Object`), generics enable **compile-time type checking** and eliminate the need for explicit type casting.

2. Why Use Generics?

1. **Type Safety**: Ensures that only valid data types are used, preventing `ClassCastException` at runtime.
2. **Code Reusability**: A single generic class/method can work with multiple data types.
3. **Eliminates Type Casting**: Avoids unnecessary explicit casting, improving code readability.
4. **Compile-time Checking**: Errors are caught early during compilation rather than at runtime.

Type Parameters in Generics

Generics use **type parameters** to define placeholder types. These are replaced by actual types at runtime. Commonly used type parameters include:

- `T` → Type
- `E` → Element (used in collections)
- `K` → Key (used in maps)
- `V` → Value

Multiple type parameters can be used, such as `<K, V>` in a key-value pair structure.

Example:

```
class Box<T> {
    private T item;

    public void setItem(T item) {
        this.item = item;
    }
    public T getItem() {
        return item;
    }
}
```

Generic Classes

A **generic class** is a class with a type parameter that allows defining attributes and methods using a flexible data type. This improves reusability while ensuring type safety.

Key Points:

- A generic class can have **one or more** type parameters.
- Type parameters can be used as **method arguments, return types, or instance variables**.
- The actual data type is determined at object creation time.

Example:

```
class Flight<T> {
    private T flightNumber;

    public Flight(T flightNumber) {
        this.flightNumber = flightNumber;
    }

    public T getFlightNumber() {
        return flightNumber;
    }
}
```

Usage:

```
Flight<Integer> flight1 = new Flight<>(101);
Flight<String> flight2 = new Flight<>("AA202");
System.out.println(flight1.getFlightNumber()); // Output: 101
System.out.println(flight2.getFlightNumber()); // Output: AA202
```

Generic Methods

A **generic method** allows defining type parameters within a method instead of the entire class. It provides flexibility when the type is only relevant within the method.

Key Points:

- Type parameters are declared before the return type.
- They allow operations on various data types without requiring method overloading.
- Generic methods can exist in **both generic and non-generic classes**.

Example:

```
class Utility {
    public static <T> void printDetails(T data) {
        System.out.println("Details: " + data);
    }
}
```

Usage:

```
Utility.printDetails("Flight AA202");
Utility.printDetails(101);
```

Bounded Type Parameters

Bounded generics **restrict** the types that can be used as type parameters. The syntax `<T extends SomeClass>` ensures that `T` must be a subclass of `SomeClass`.

Types of Bounds:

- **Upper Bounded (`extends`)**: Restricts to a specific type or its subclasses.
- **Lower Bounded (`super`)**: Restricts to a specific type or its superclasses.

- **Multiple Bounds (`&`)**: Allows specifying multiple constraints, but only one class can be extended.

Key Points:

- Prevents usage of incompatible types.
- Enables access to methods of the bounded type.
- Ensures **compile-time safety** while maintaining flexibility.

Example:

```
class FlightSchedule<T extends Number> {
    private T flightCode;

    public FlightSchedule(T flightCode) {
        this.flightCode = flightCode;
    }

    public T getFlightCode() {
        return flightCode;
    }
}
```

✅ **Valid:** `FlightSchedule<Integer> flight1 = new FlightSchedule<>(202);`
❌ **Invalid:** `FlightSchedule<String> flight2 = new FlightSchedule<>("AA202"); // Compilation Error`

Wildcards in Generics

Wildcards (`?`) introduce **flexibility** in handling unknown types when dealing with **collections or hierarchies**.

Types of Wildcards:

1. **Unbounded Wildcard (`?`)**: Allows any type and is used when the exact type is unknown.
2. **Upper Bounded Wildcard (`? extends T`)**: Accepts `T` or any subclass of `T`, making it **read-only**.

3. **Lower Bounded Wildcard (`? super T`)**: Accepts `T` or any superclass of `T`, making it **write-compatible**.

Key Points:

- Wildcards help in designing **generic APIs**.
- They provide flexibility while maintaining **type safety**.
- **Wildcard capture** occurs when attempting modifications on collections with unknown types.

Types of Wildcards

1. **Unbounded Wildcard (`?`)**
   - Used when we don't know the type.

```
public static void displayFlightDetails(List<?> flights) {
    for (Object flight : flights) {
        System.out.println(flight);
    }
}
```

2. **Upper Bounded Wildcard (`? extends T`)**
   - Accepts any class that is a subclass of `T`.

```
public static void printFlightNumbers(List<? extends Number>
flightNumbers) {

    for (Number num : flightNumbers) {
        System.out.println(num);
    }
}
```

3. ✅ **Valid:** `List<Integer> flights = Arrays.asList(101, 202, 303);`
4. **Lower Bounded Wildcard (`? super T`)**
   - Accepts any class that is a superclass of `T`.

```
public static void addFlightCodes(List<? super Integer>
flightCodes) {
    flightCodes.add(404);
}
```

5. ✅ **Valid:** `List<Number> flights = new ArrayList<>(); addFlightCodes(flights);`

Flight Scheduling System using Generics

Let's design a **Flight Scheduling System** using Generics.

1. Generic Class for Flights

```
class Flight<T> {
    private T flightNumber;
    private String departure;
    private String destination;

    public Flight(T flightNumber, String departure, String
destination) {
        this.flightNumber = flightNumber;
        this.departure = departure;
        this.destination = destination;
    }

    public T getFlightNumber() {
        return flightNumber;
    }

    public void displayFlightInfo() {
        System.out.println("Flight: " + flightNumber + " | From: " +
departure + " | To: " + destination);
    }
}
```

2. Generic Class for Booking System

```
class Booking<T> {
    private T bookingId;
    private Flight<?> flight;
    private String passengerName;

    public Booking(T bookingId, Flight<?> flight, String
passengerName) {
        this.bookingId = bookingId;
        this.flight = flight;
        this.passengerName = passengerName;
    }

    public void displayBookingInfo() {
        System.out.println("Booking ID: " + bookingId);
        System.out.println("Passenger: " + passengerName);
        flight.displayFlightInfo();
    }
}
```

---

3. Flight Management System with Bounded Types

```
class FlightManager<T extends Number> {
    private List<Flight<T>> flights = new ArrayList<>();

    public void addFlight(Flight<T> flight) {
        flights.add(flight);
    }

    public void displayAllFlights() {
        for (Flight<T> flight : flights) {
            flight.displayFlightInfo();
        }
    }
}
```

4. Utility Methods with Wildcards

```
class FlightUtility {
     public static void displayFlightDetails(List<? extends Flight<?>>
flights) {
        for (Flight<?> flight : flights) {
            flight.displayFlightInfo();
        }
    }
}
```

Usage Example

```
public class FlightSystem {
    public static void main(String[] args) {
            Flight<Integer> flight1 = new Flight<>(101, "New York",
"London");
         Flight<String> flight2 = new Flight<>("AA202", "Los Angeles",
"Tokyo");

         Booking<Integer> booking1 = new Booking<>(5001, flight1, "John
Doe");
            Booking<String> booking2 = new Booking<>("B102", flight2,
"Jane Smith");

        booking1.displayBookingInfo();
        System.out.println("-----------------");
        booking2.displayBookingInfo();

        System.out.println("\n--- Flight Management ---");
        FlightManager<Integer> manager = new FlightManager<>();
        manager.addFlight(flight1);
        manager.displayAllFlights();
    }
}
```

Output

```
Booking ID: 5001
Passenger: John Doe
Flight: 101 | From: New York | To: London
----------------
Booking ID: B102
Passenger: Jane Smith
Flight: AA202 | From: Los Angeles | To: Tokyo

--- Flight Management ---
Flight: 101 | From: New York | To: London
```