Version 1.1

January 2026

DATA STRUCTURES

MODULE 1 - GENERICS JAVA

Author:

MUHAMMAD ILHAM PERDANA. S.Tr.T., M.T.

AHMAD NUR MU’MININ

WAHYU ANDIKA

INFORMATICS LABORATORY

University of Muhammadiyah Malang

TABLE OF CONTENTS

INTRODUCTION

OBJECTIVES

1. Students are able to understand and explain the fundamental concepts of Java Generics and identify the role of generics in various real-world scenarios.

MODULE TARGETS

1. Students are able to understand : Implementing arrays of objects, Generic Classes, Generic Methods, Wildcards, Enumerated Types.

PREPARATION

1. Muslim students may recite the following prayer before starting the lesson
رَضِيْتُ بِاللهِ رَبًّا وَبِالإِسْلَامِ دِيْنًا وَبِمُحَمَّدٍ نَبِيًّا وَرَسُوْلًا. رَبِّ زِدْنِي عِلْمًا وَارْزُقْنِي فَهْمًا
2. Laptop or personal computer
3. Java Development Kit
4. Java Runtime Environment
5. IDE (Intellij IDEA, Eclipse, Netbeans, etc.)
6. Strong motivation and commitment to learning

KEYWORDS

Generics, Type Parameter, Generic Class, Generic Method, Wildcard, Type Safety, Parameterized Type.

REFERENCES

Oracle iLearning – Java Programming, Section 6-1: Generics

GeeksforGeeks: https://www.geeksforgeeks.org/generics-in-java/

YouTube (Indonesian): https://www.youtube.com/watch?v=bvWRDAI30Gs

YouTube (English): https://www.youtube.com/watch?v=K1iu1kXkVoA

Javatpoint: https://www.javatpoint.com/generics-in-java

Programiz: https://www.programiz.com/java-programming/generics

Note: These references may contain slight variations in explanation or examples.Focus on understanding the core concepts and apply them to the case study provided in this module.

CONTENTS

Welcome to the world of Data Structures. Before we construct more complex structures such as Stacks or Queues, we must first solve one fundamental challenge in programming.

How can we write code that handles different kinds of data without rewriting the same logic over and over again?

Imagine you are the architect of a railway station. You are asked to design a Storage Locker System.

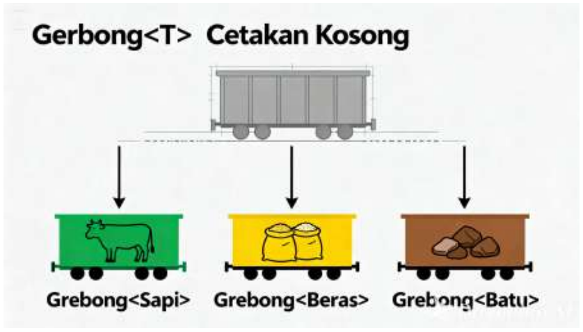
- If you design lockers that fit only backpacks, large suitcases won't fit
- If you design lockers specifically for suitcases, small wallets will be inconvenient to place

In traditional programming, we often fall into the trap of making rigid containers like NumberBox, StringBox, ObjectBox.This is wasteful and repetitive. Java Generics solves this problem. Generics act like a **Blank Label Sticker** on a container. When you design the container, you don't decide what goes inside. The label (data type) is written later, when the container is used.

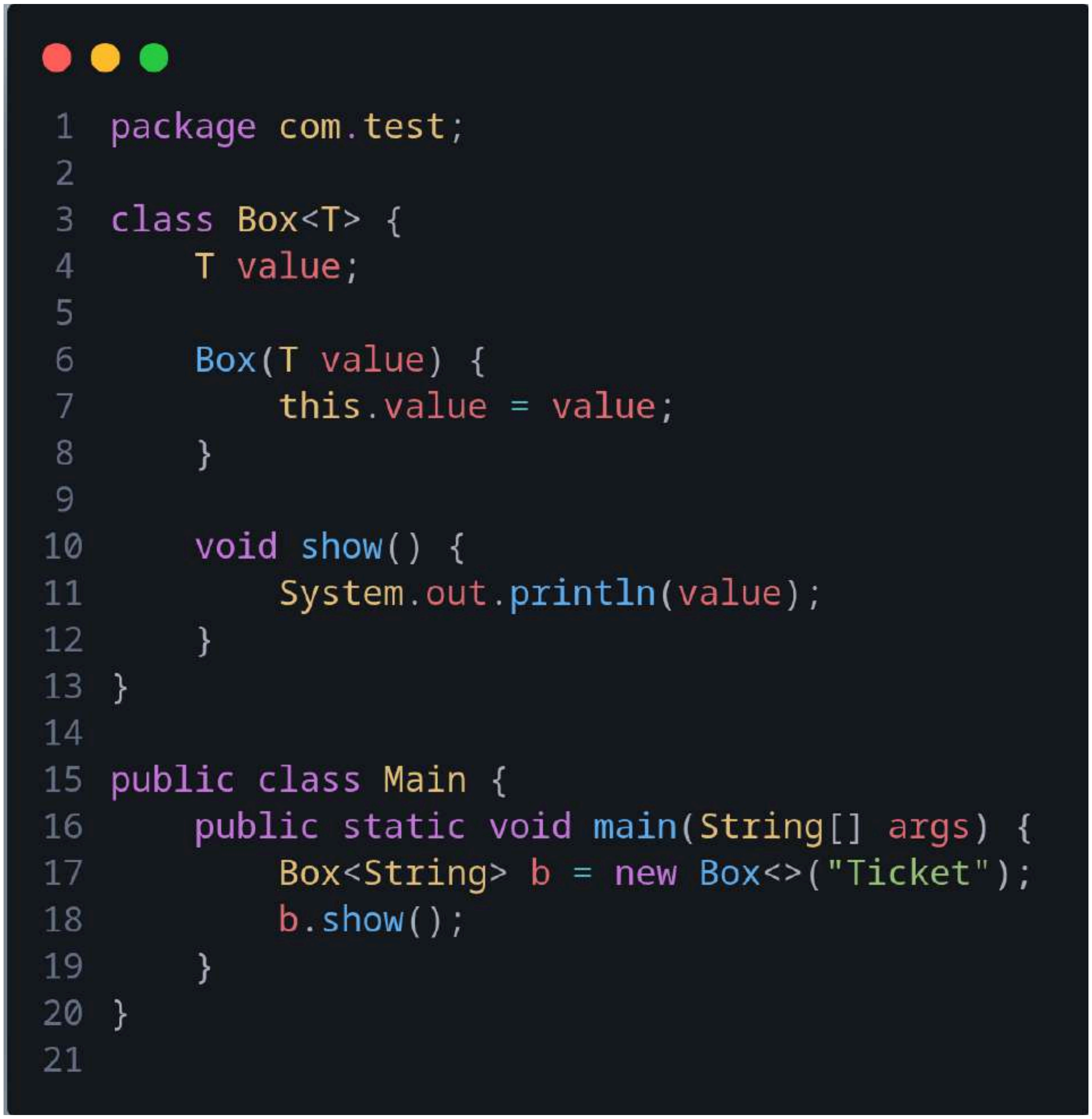
1. Generics

1.1 What is Generics

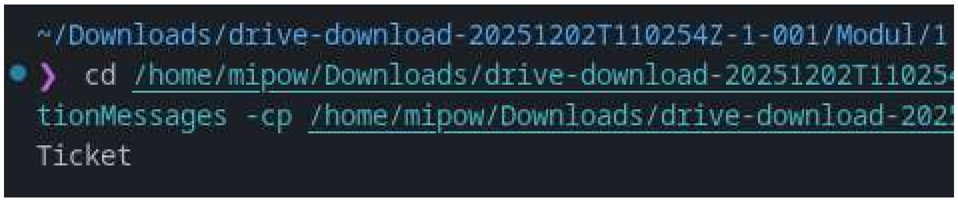
Generics are a Java feature that allows us to use Data Types as Parameters when defining a Class, Interface, or Method. In simple terms, Generics lets us create a Class that is flexible. The data type inside the class is not fixed during coding, but determined later when the class is instantiated.



Code example


```java
package com.test;

class Box<T> {
    T value;

    Box(T value) {
        this.value = value;
    }

    void show() {
        System.out.println(value);
    }
}

public class Main {
    public static void main(String[] args) {
        Box<String> b = new Box<>("Ticket");
        b.show();
    }
}

```

Output


```
~/Downloads/drive-download-20251202T110254Z-1-001/Modul/1
❯ cd /home/mipow/Downloads/drive-download-20251202T11025
tionMessages -cp /home/mipow/Downloads/drive-download-202
Ticket
```

1.2 Why Study Generics

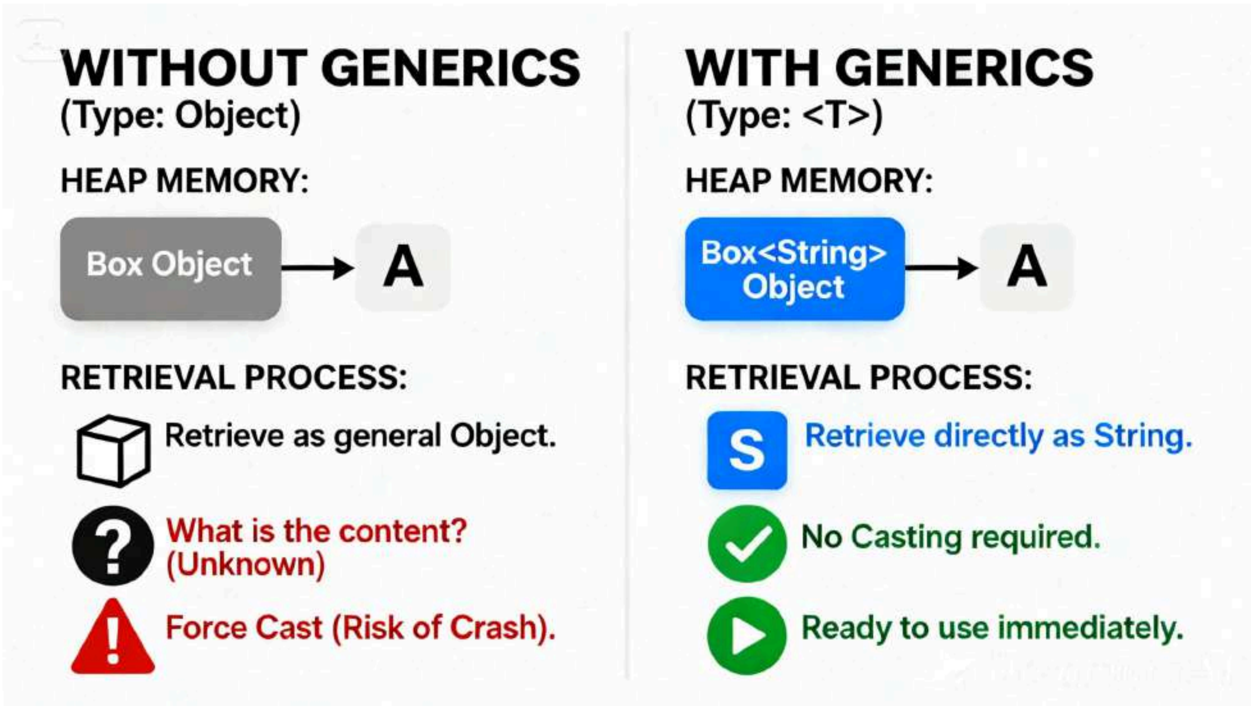
Learning Generics is crucial in modern software development for two main reasons: safety and efficiency.

First, Generics provide Type Safety. Without Generics, type errors are often detected only at runtime, which can cause applications to crash for end users. With Generics, such mistakes are caught by the compiler while we are writing code (Compile-Time Error), which is much safer.

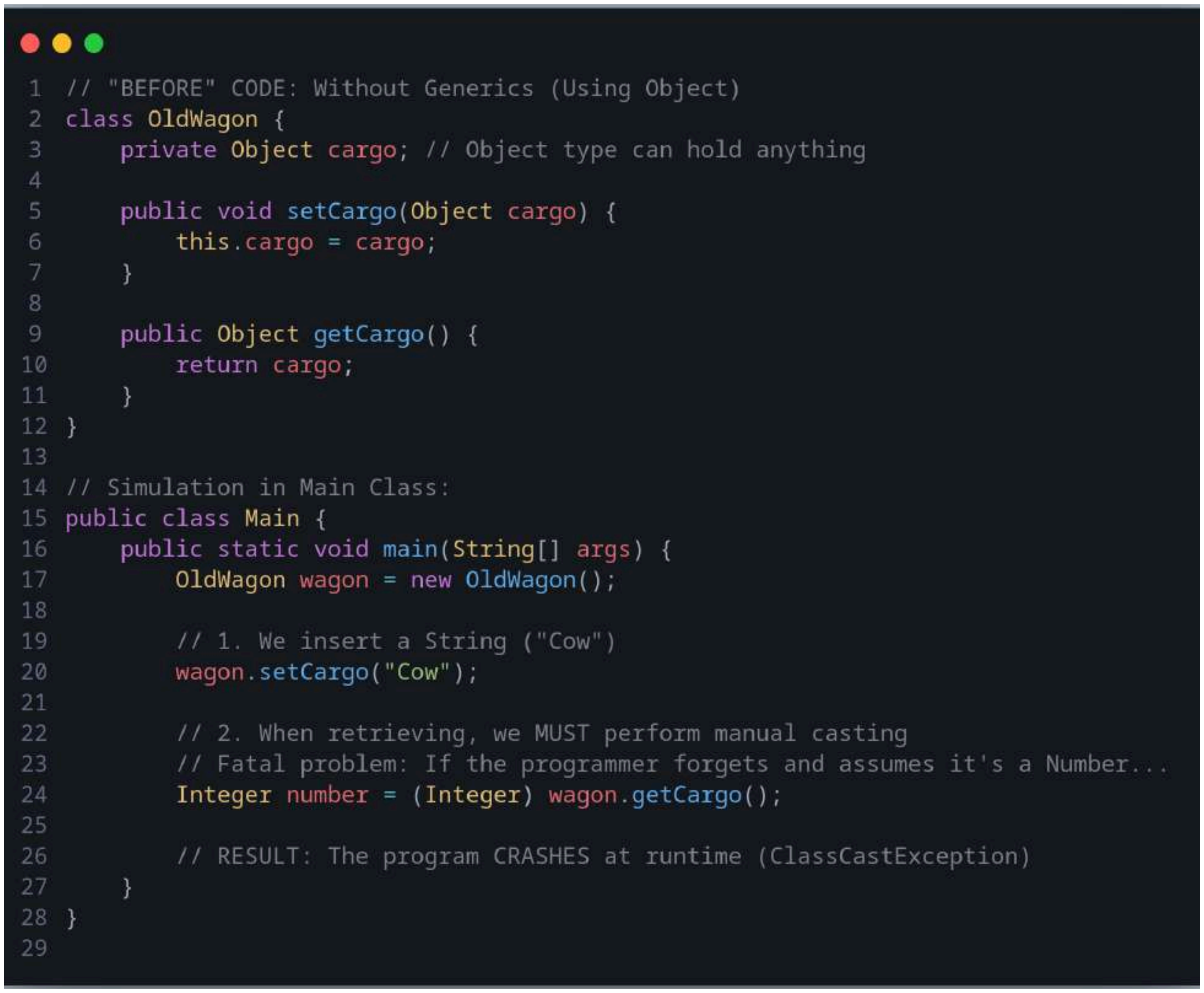
Second, Generics enable Code Reusability. As a programmer, you don't have to waste time writing duplicate classes such as `WagonString`, `WagonInteger`, or `WagonDouble` that contain the same logic. You can simply write one class `Wagon <T>` that works for any data type. This is the foundation of modern data structures (Stack, Queue, Linked List) that we will study later.

1.3 Difference in Code With and Without Generics

In the past (before Java 5), programmers relied on the Object type as a universal container. This approach is like a mysterious box into which you can put Strings, Integers, or custom objects indiscriminately. Although flexible, it creates a major risk of type unsafety. When you take an item from that box, the compiler does not know what the object actually is, forcing you to perform a manual "cast" or conversion. If you misremember the type, for example trying to treat a stored String as an Integer, the program will crash when executed (`ClassCastException`).

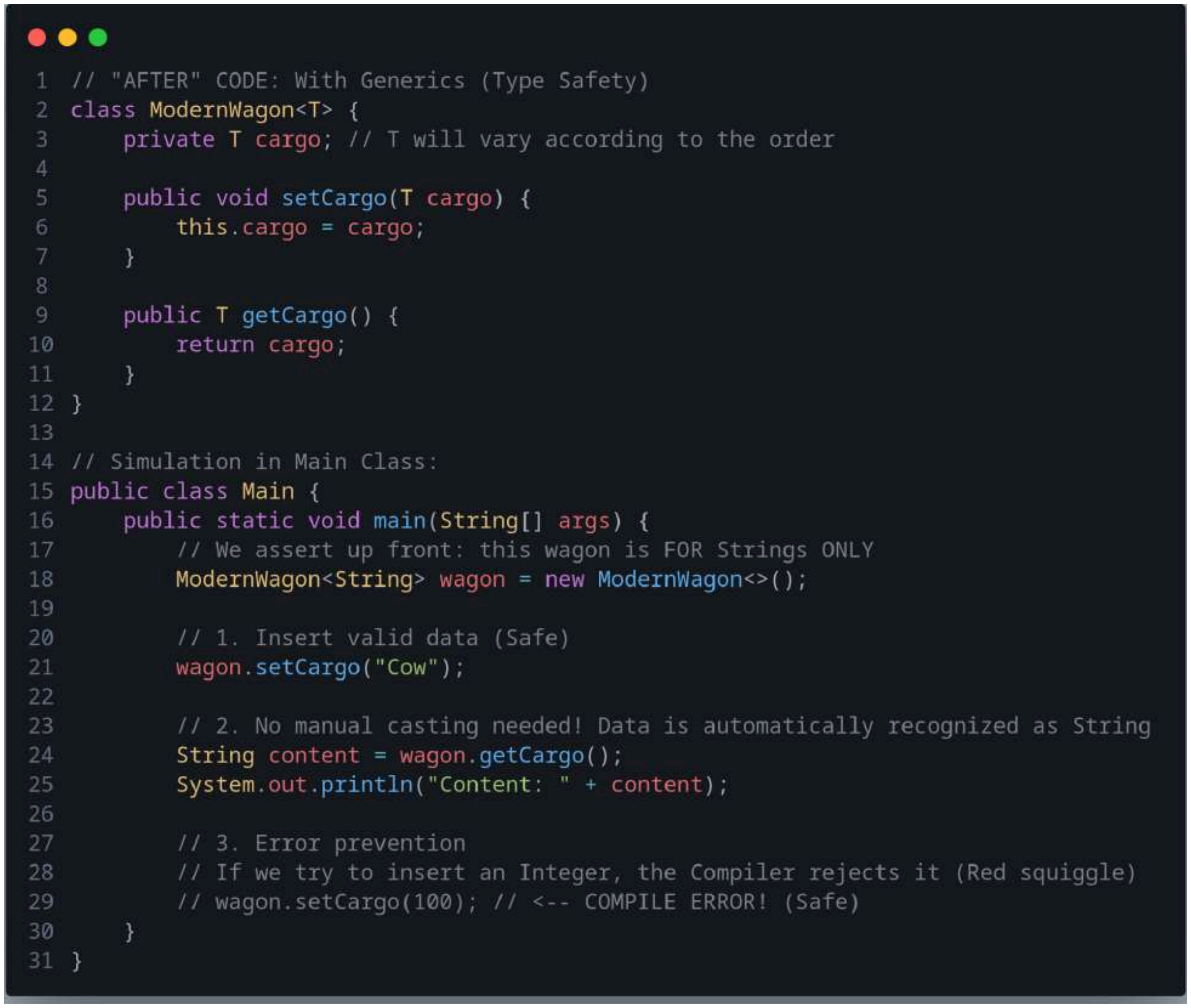


Below is an example of code before Generics (The Old Way). Note that we must guess the data type and perform manual casting, which is highly prone to runtime errors that can crash the program.



```java
// "BEFORE" CODE: Without Generics (Using Object)
class OldWagon {
    private Object cargo; // Object type can hold anything

    public void setCargo(Object cargo) {
        this.cargo = cargo;
    }

    public Object getCargo() {
        return cargo;
    }
}

// Simulation in Main Class:
public class Main {
    public static void main(String[] args) {
        OldWagon wagon = new OldWagon();

        // 1. We insert a String ("Cow")
        wagon.setCargo("Cow");

        // 2. When retrieving, we MUST perform manual casting
        // Fatal problem: If the programmer forgets and assumes it's a Number...
        Integer number = (Integer) wagon.getCargo();

        // RESULT: The program CRASHES at runtime (ClassCastException)
    }
}

```

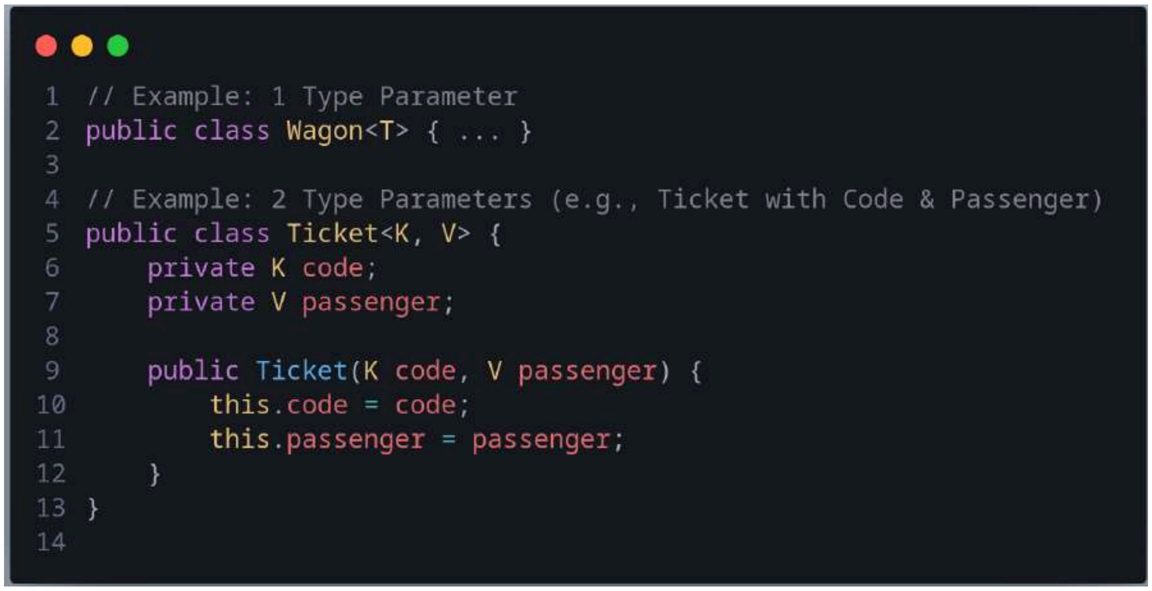
Compare that to the modern approach using Generics, which enforces strict type checking. By specifying the type up front with angle brackets `<...>`, the compiler acts as a strict gatekeeper. It ensures only data of the correct type may enter. Below is an example of code after using Generics . Notice we no longer need to perform casting and type errors are prevented by the compiler before the program runs.



```java
// "AFTER" CODE: With Generics (Type Safety)
class ModernWagon<T> {
    private T cargo; // T will vary according to the order

    public void setCargo(T cargo) {
        this.cargo = cargo;
    }

    public T getCargo() {
        return cargo;
    }
}

// Simulation in Main Class:
public class Main {
    public static void main(String[] args) {
        // We assert up front: this wagon is FOR Strings ONLY
        ModernWagon<String> wagon = new ModernWagon<>();

        // 1. Insert valid data (Safe)
        wagon.setCargo("Cow");

        // 2. No manual casting needed! Data is automatically recognized as String
        String content = wagon.getCargo();
        System.out.println("Content: " + content);

        // 3. Error prevention
        // If we try to insert an Integer, the Compiler rejects it (Red squiggle)
        // wagon.setCargo(100); // <-- COMPILE ERROR! (Safe)
    }
}
```

1.4 Declaration class with Generics Method

Declaring a generic class involves a simple yet powerful syntax change: add angle brackets containing the type parameter (commonly `<T>`) immediately after the class name. The letter T serves as a placeholder or type variable for the data type that will be defined later when the object is created. It acts as a contract that wherever T appears in the class—whether as an attribute, method parameter, or return type—it will be replaced by an actual type such as String, Integer, or a custom class like Passenger.



```java
// Example: 1 Type Parameter
public class Wagon<T> { ... }

// Example: 2 Type Parameters (e.g., Ticket with Code & Passenger)
public class Ticket<K, V> {
    private K code;
    private V passenger;

    public Ticket(K code, V passenger) {
        this.code = code;
        this.passenger = passenger;
    }
}

```

This flexibility is not limited to a single parameter. For more complex data structures, such as maps that require key-value pairs, you can declare multiple type variables separated by commas, e.g., `<K, V>`. In this context, K usually represents the Key type and V the Value type. When the class is used, the programmer supplies specific types for both, enabling highly-structured and type-safe associations.

1.5 Parameter Naming Conventions for Generics

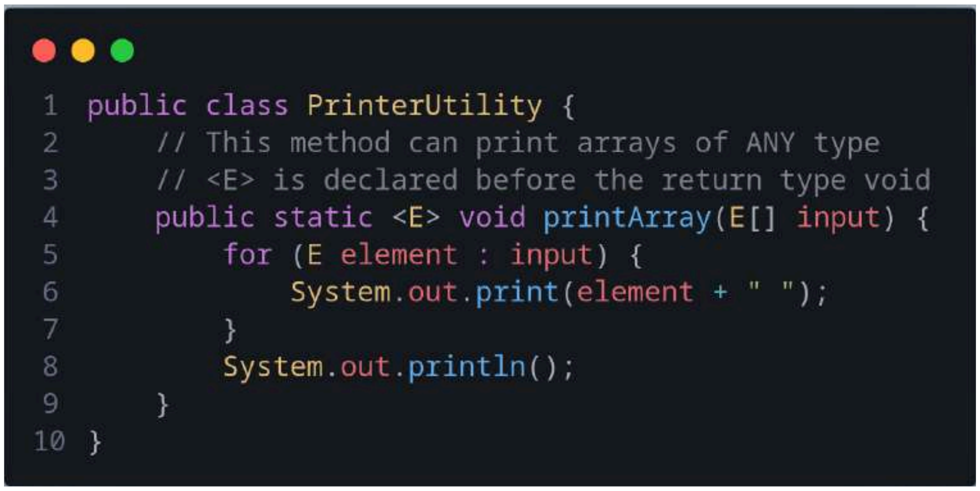
Although Java technically allows any name to be used as a type parameter, the developer community follows a strict single-letter naming convention. This practice is important for readability, as it allows developers to instantly distinguish type parameters (placeholders) from regular class names.

- `E - Element` (used by Java Collections Framework)
- `K - Key`
- `N - Number`
- `T - Type`
- `V - Value`

1.6 Generics Method

Generics are not limited to class-level declarations; they can also be applied to individual methods to create versatile and reusable logic, regardless of the class where they reside. A Generic Method declares its own type parameter, placed before the return type. This allows the method to accept arguments of different types in different calls.

This is analogous to a universal tool that can adapt to different bolt sizes. For example, a single utility method designed to print the contents of an array can handle an integer array in one call and a string array in another, without requiring method overloading or code duplication.

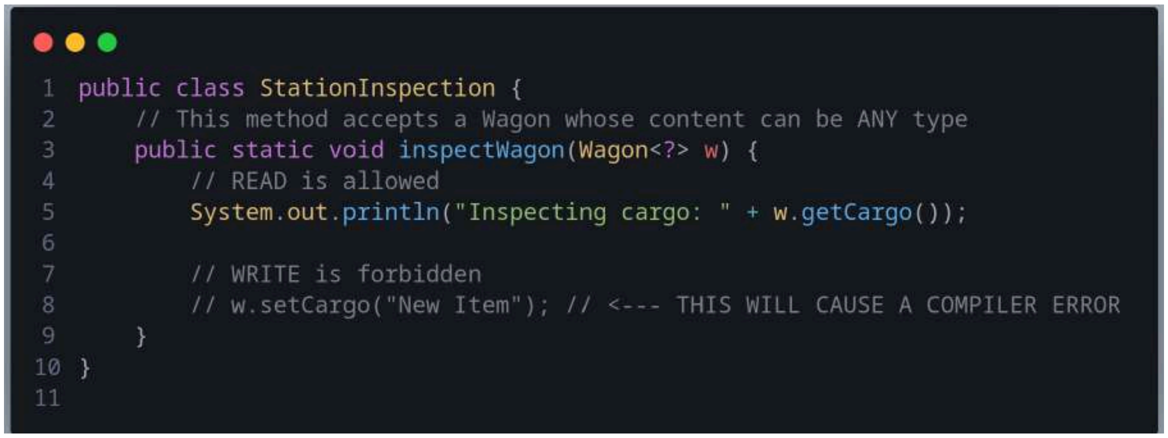


```java
public class PrinterUtility {
    // This method can print arrays of ANY type
    // <E> is declared before the return type void
    public static <E> void printArray(E[] input) {
        for (E element : input) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

2. Wildcards

In certain scenarios we need code that interacts with generic objects without knowing their concrete type arguments, or when we want to handle a family of generic types. This is where the wildcard (represented by the question mark <?>) becomes essential. A wildcard stands for an unknown type, allowing a method to accept, for example, both `Wagon<String>` and `Wagon<Integer>`.

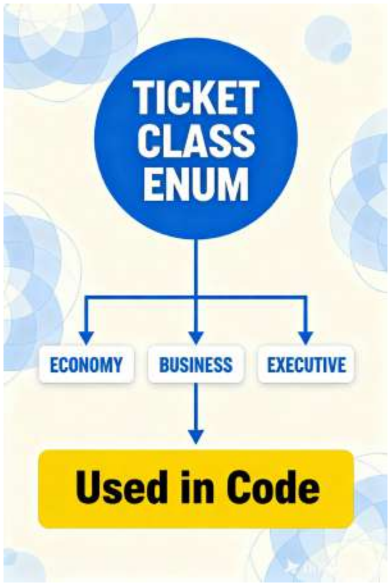
However, this flexibility comes with a strict safety constraint known as the "Get and Put" principle. Because the compiler cannot guarantee the actual underlying type of the wildcarded object, it treats the object as read-only. You can safely retrieve and inspect data (since every Java object is at least an Object), but you are forbidden to modify or add new data into it. This restriction prevents type corruption; for example, it prevents you from accidentally adding an Integer into a List that actually contains Strings.



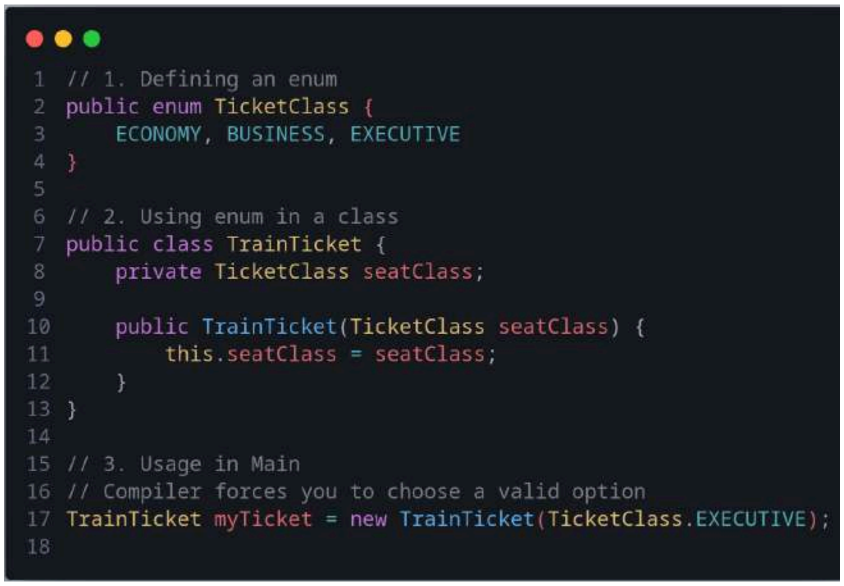
```java
public class StationInspection {
    // This method accepts a Wagon whose content can be ANY type
    public static void inspectWagon(Wagon<?> w) {
        // READ is allowed
        System.out.println("Inspecting cargo: " + w.getCargo());

        // WRITE is forbidden
        // w.setCargo("New Item"); // <--- THIS WILL CAUSE A COMPILER ERROR
    }
}

```

3. Enumerated Types

Enumerated types, or Enums, provide a robust mechanism to define a named set of fixed constants, representing a collection of related values that never change. Instead of relying on free-form strings or integers that are prone to typos and invalid input (for example, typing "**ekonomi**" in lowercase while the system expects "**EKONOMI**"), Enums enforce strict type safety by limiting values to the pre-defined list.



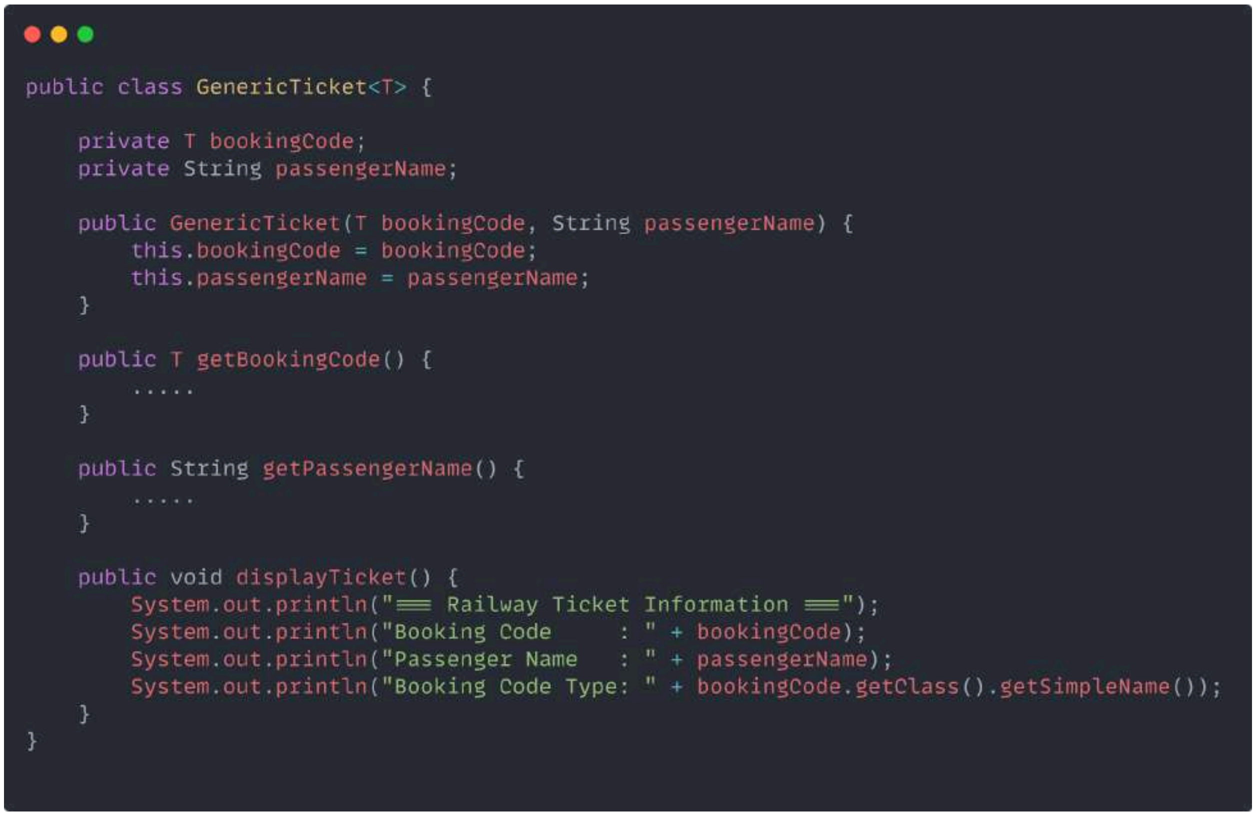
In the context of our railway system, attributes such as ticket class or cargo category are perfect candidates for Enums. By defining them as **ECONOMY**, **BUSINESS**, or **EXECUTIVE**, we ensure the system rejects undefined values at compile time. This makes code more readable, easier to maintain, and more robust, because the compiler verifies that only allowed constants are used throughout the application.



```java
// 1. Defining an enum
public enum TicketClass {
    ECONOMY, BUSINESS, EXECUTIVE
}

// 2. Using enum in a class
public class TrainTicket {
    private TicketClass seatClass;

    public TrainTicket(TicketClass seatClass) {
        this.seatClass = seatClass;
    }
}

// 3. Usage in Main
// Compiler forces you to choose a valid option
TrainTicket myTicket = new TrainTicket(TicketClass.EXECUTIVE);

```

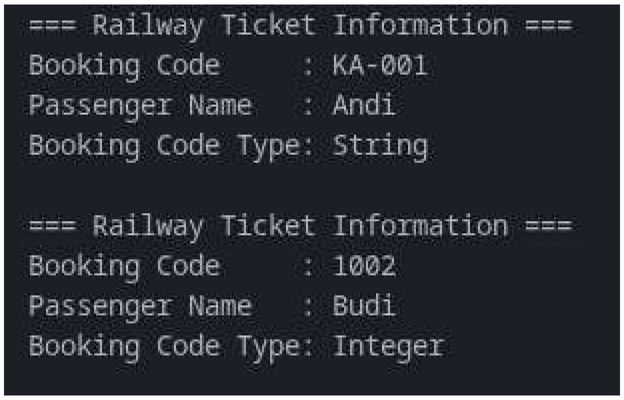
CODELAB

Codelab offers students a fully functional Java program to illustrate the fundamental application of Java Generics. Complete the following code so that it runs.



```java
public class GenericTicket<T> {

    private T bookingCode;
    private String passengerName;

    public GenericTicket(T bookingCode, String passengerName) {
        this.bookingCode = bookingCode;
        this.passengerName = passengerName;
    }

    public T getBookingCode() {
        .....
    }

    public String getPassengerName() {
        .....
    }

    public void displayTicket() {
        System.out.println("=== Railway Ticket Information ===");
        System.out.println("Booking Code      : " + bookingCode);
        System.out.println("Passenger Name    : " + passengerName);
        System.out.println("Booking Code Type: " + bookingCode.getClass().getSimpleName());
    }
}
```

Rewrite the Data Structure Implementation program in Java Generics above. Here, the expected Output:



```
=== Railway Ticket Information ===
Booking Code      : KA-001
Passenger Name    : Andi
Booking Code Type: String

=== Railway Ticket Information ===
Booking Code      : 1002
Passenger Name    : Budi
Booking Code Type: Integer
```

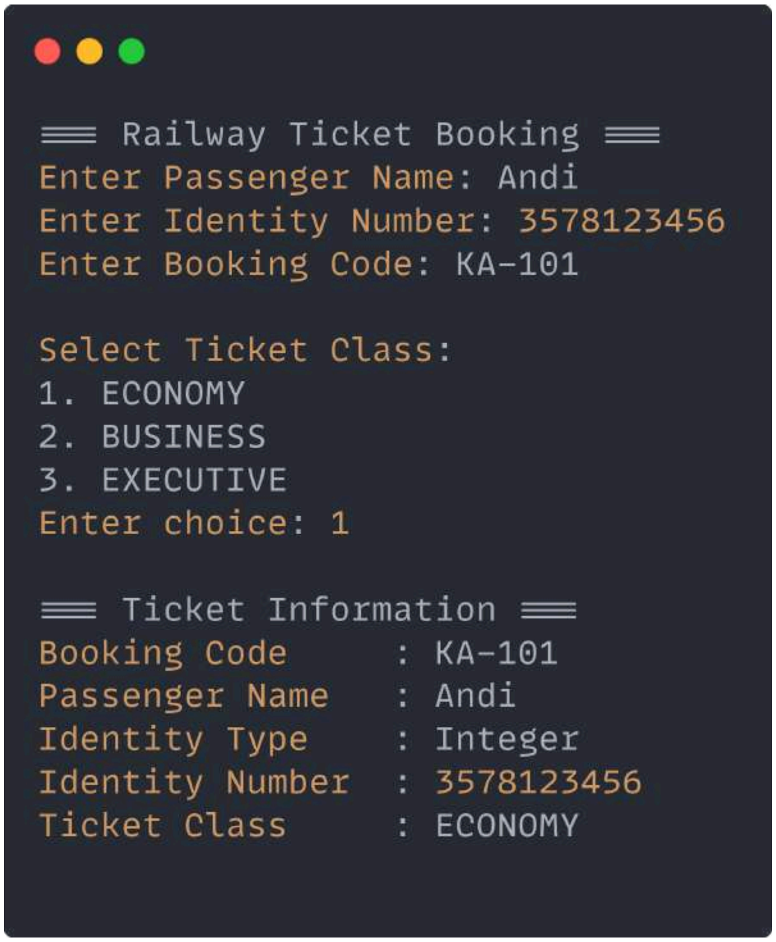
ASSIGNMENT

Task 1

In this DEMO, students are required to build the foundational data model for a Railway Ticket Booking System using Java Generics. Students must implement the following requirements in their program:

1. Students must create a generic class that represents a train passenger and represents a train ticket
2. Students must define an enum named `TicketClass` with the following constants: (Economy, Business, Executive).
3. Students must implement at least one method that uses wildcards (<?>).

Output :



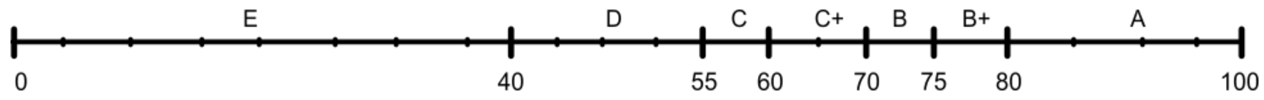
```
=== Railway Ticket Booking ===
Enter Passenger Name: Andi
Enter Identity Number: 3578123456
Enter Booking Code: KA-101

Select Ticket Class:
1. ECONOMY
2. BUSINESS
3. EXECUTIVE
Enter choice: 1

=== Ticket Information ===
Booking Code      : KA-101
Passenger Name    : Andi
Identity Type     : Integer
Identity Number   : 3578123456
Ticket Class      : ECONOMY
```

Assessment

Grading Rubric

Assessment Aspect	Points (Total 100%)
Codelab	**Total 5%**
Successfully run	5%
Demo	**Total 95%**
Implement Generic	20%
Implement Wildcards	15%
Implement Enum for ticket class	20%
Running Smoothly	10%
Answer Questions	30%

Grading Scale



A	**=**	**(81 - 100) → Sepuh**
B+	**=**	**(75 - 80) → Very Good**
B	**=**	**(70 - 74) → Good**
C+	**=**	**(60 - 69) → Fairly Good**
C	**=**	**(55 - 59) → Fair**
D	**=**	**(41 - 54) → Poor**
E	**=**	**(0 - 40) → Bro really...**