Java Unit - 5

I/O Fundamentals : Describing the basics of input and output in Java, Read and write data from various sources, Using streams to read and write files, Writing and read objects using serialization

Generics : Creating a custom generic class, Using the type inference diamond to create an object, Using bounded types and Wild Cards.

I/O Fundamentals

Overview of Java I/O

Java provides a comprehensive API for handling input and output (I/O) operations efficiently. These operations allow programs to read from and write to various data sources, such as:

- **Files** (reading/writing to disk)
- **Network connections** (sending/receiving data over the internet)
- **Memory buffers** (storing temporary data in memory)
- **External devices** (such as keyboards, printers, and sensors)

1. Input in Java

Java supports input from multiple sources, such as the keyboard, files, and network connections. The `java.io` package provides several classes to handle input operations.

Reading from the Keyboard

The most common way to read input from the keyboard is through the `Scanner` class, which is part of the `java.util` package.

Reading input from the user through the console is common in programs where user interaction is required. Java provides the `Scanner` class to handle input from the console.

Content to Cover:

- **Scanner Class**:
  The `Scanner` class is part of the `java.util` package and simplifies reading input from the console, files, or other input sources. It can read various data types like strings, integers, floats, etc.

- **Common Methods**:
    - `nextLine()`: Reads an entire line of input as a `String`.
    - `nextInt()`: Reads an integer from the input.
    - `nextDouble()`: Reads a double (floating-point number) from the input.

```java
import java.util.Scanner;                          //Ummed Singh

public class KeyboardInput {

   public static void main(String[] args) {

       Scanner scanner = new Scanner(System.in);

       System.out.print("Enter your name: ");

       String name = scanner.nextLine(); // Reading input from
the keyboard

       System.out.println("Hello, "+ name);

   }

}
```

2. Output in Java

Java provides several ways to send output to different destinations, such as the console, files, or network connections.

Writing to the Console

Writing output to the console is fundamental for displaying results or debugging information in a Java program. To write output to the console, Java provides the `System.out` object. The `print()` and `println()` methods are used for outputting text.

- **System.out.print() / System.out.println()**:
    - `System.out.print()`: Outputs text on the console but **does not move to the next line**.
    - `System.out.println()`: Outputs text on the console and **moves to the next line**.

```
public class ConsoleOutput {                        //Ummed Singh

   public static void main(String[] args) {

       System.out.println("Hello, World!"); //Writing output to
the console

   }

}
```

System.out.printf(): You can also format output using `printf()`, similar to how it is done in languages like C.

```
System.out.printf("Name: %s, Age: %d", name, age);
```

Reading and Writing Data from Various Sources

Java provides multiple ways to **read (input) and write (output) data** from various sources. This is essential for interacting with external systems such as files, databases, and networks.

Common Data Sources for Java I/O

Java can read and write data from different sources, such as:

1. **Files** – Read and write data stored in text or binary files.
2. **Console (Standard Input/Output)** – Take user input from the keyboard and display output on the screen.
3. **Network Connections** – Send and receive data over the internet or local network.
4. **Databases (via JDBC)** – Store and retrieve structured data from databases like MySQL, PostgreSQL, etc.
5. **Memory Buffers** – Store temporary data in memory instead of using files or databases.

Reading and Writing Data from Different Sources

1. Files (Reading and Writing to a File)

Files are one of the most common ways to store data. Java allows reading from and writing to files using **streams**.

Reading Data from a File (Text File)

We use `FileReader` or `BufferedReader` to read character data from a text file.
Example:

```java
import java.io.*;                                             //Ummed Singh
public class FileReadExample {
   public static void main(String[] args) {
       try (BufferedReader reader = new BufferedReader(new
FileReader("example.txt"))) {
           String line;
           while ((line = reader.readLine()) != null) {
               System.out.println(line);
           }
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

👉 This reads the content of `example.txt` **line by line** and prints it to the console.

Writing Data to a File

We use `FileWriter` or `BufferedWriter` to write text to a file.

```java
import java.io.*;                                             // Ummed Singh
public class FileWriteExample {
   public static void main(String[] args) {
       try (BufferedWriter writer = new BufferedWriter(new
FileWriter("output.txt"))) {
           writer.write("Hello, Java File Handling!");
           System.out.println("Data written successfully!");
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

👉 This creates a file `output.txt` and writes "Hello, Java File Handling!" into it.

---

2. Console (Reading and Writing from the Keyboard and Screen)

The **console** (or terminal) is the standard input and output for a program.

Reading Input from the User

We can use `Scanner` or `BufferedReader` to take user input.

```
import java.util.Scanner;                                    // Ummed Singh
public class ConsoleInputExample {
   public static void main(String[] args) {
       Scanner scanner = new Scanner(System.in);
       System.out.print("Enter your name: ");
       String name = scanner.nextLine();
       System.out.println("Hello, " + name + "!");
       scanner.close();
   }
}
```

👉 This asks the user for their name and prints a greeting.

Writing Output to the Console

We use `System.out.println()` to display messages.

```
public class ConsoleOutput {                                 // Ummed Singh

   public static void main(String[] args) {

       System.out.println("Hello, World!"); //Writing output to
the console

   }
}
```

---

3. Network Connections (Reading and Writing Data Over a Network)

Java allows communication over a **network** (e.g., the internet) using **Sockets**.

Reading Data from a Website (URL)

We can fetch web page data using `URLConnection`.

```
import java.io.*;                                          // Ummed Singh
import java.net.*;
public class NetworkReadExample {
   public static void main(String[] args) {
       try {
           URL url = new URL("https://www.example.com");
           BufferedReader reader = new BufferedReader(new
InputStreamReader(url.openStream()));

           String line;
           while ((line = reader.readLine()) != null) {
               System.out.println(line);
           }
           reader.close();
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

👉 This program fetches and prints data from https://www.example.com.

Sending Data Over the Network

We use **Sockets** to send data over the network.
 Example: Sending a simple message to a server.

```
import java.io.*;                                          // Ummed Singh
import java.net.*;
```

```
public class NetworkWriteExample {
   public static void main(String[] args) {
       try (Socket socket = new Socket("example.com", 80);
            PrintWriter writer = new
PrintWriter(socket.getOutputStream(), true)) {
           writer.println("GET / HTTP/1.1\r\nHost:
example.com\r\n\r\n");
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

👉 This sends a basic **HTTP GET request** to `example.com`.

---

4. Databases (Reading and Writing Data using JDBC)

JDBC (Java Database Connectivity) allows Java programs to interact with **databases** like MySQL, PostgreSQL, etc.

Reading Data from a Database

Example: Fetching data from a **MySQL database**.

```
import java.sql.*;                                      // Ummed Singh

public class DatabaseReadExample {
   public static void main(String[] args) {
       try {
           Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
"user", "password");
           Statement stmt = con.createStatement();
           ResultSet rs = stmt.executeQuery("SELECT * FROM
students");

           while (rs.next()) {
               System.out.println(rs.getString("name") + " - " +
rs.getInt("age"));
           }
```

```
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

👉 This connects to a **MySQL database**, retrieves data from the `students` table, and prints it.

Writing Data to a Database

```
import java.sql.*;                                              // Ummed Singh
public class DatabaseWriteExample {
    public static void main(String[] args) {
        try {
            Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb",
"user", "password");
            PreparedStatement pstmt =
con.prepareStatement("INSERT INTO students (name, age) VALUES
(?, ?)");
            pstmt.setString(1, "Aman");
            pstmt.setInt(2, 22);
            pstmt.executeUpdate();
            System.out.println("Data inserted successfully!");

            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

👉 This inserts a new student **"Aman, 22"** into the database.

---

5. Memory Buffers (Storing Temporary Data in Memory)

Memory buffers store data **temporarily in RAM** instead of a file or database.

Reading from a Memory Buffer

We use `ByteArrayInputStream` for byte data.

```
import java.io.*;                                            // Ummed Singh
public class MemoryReadExample {
   public static void main(String[] args) {
       byte[] data = "Hello, Memory!".getBytes();
       ByteArrayInputStream bais = new
ByteArrayInputStream(data);

       int content;
       while ((content = bais.read()) != -1) {
           System.out.print((char) content);
       }
   }
}
```

👉 This reads data from memory instead of a file.

Writing Data to a Memory Buffer

We use `ByteArrayOutputStream` to store data in memory.

```
import java.io.*;                                            // Ummed Singh
public class MemoryWriteExample {
   public static void main(String[] args) {
       ByteArrayOutputStream baos = new ByteArrayOutputStream();
       String message = "Stored in memory!";

       try {
           baos.write(message.getBytes());
           System.out.println("Data stored: " +
baos.toString());
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

👉 This writes a message into memory instead of a file.

Using Streams to Read and Write Files

Stream

A **stream** in Java is a sequence of data elements that can be read from or written to. It acts as an abstraction to represent the flow of data between a program and an I/O source.

Types of I/O Streams

Java provides several types of I/O streams, but they mainly fall into two categories when working with files: **Byte Streams** and **Character Streams**.

Byte Streams vs. Character Streams

- **Byte Streams** handle data in bytes (8 bits at a time). These are ideal for reading or writing binary data like images, audio files, or any non-text content.

- **Character Streams** process data in characters (16 bits at a time). These are designed specifically for handling text, especially when dealing with different human languages that require more than one byte to represent a single character. Using Character Streams ensures that each character is properly interpreted, avoiding data loss or misrepresentation.

When to Use Which Stream

- Use **Byte Streams** for non-text data where character encoding is not a concern.

- Use **Character Streams** for text data, especially when working with human languages and their specific characters.

How to Identify Stream Types

You can distinguish between Byte and Character streams by their class names in Java:

- **Byte Streams** typically include the word **`Stream`** in the class name (e.g., `FileInputStream`, `BufferedOutputStream`).

- **Character Streams** include **`Reader`** or **`Writer`** in their names (e.g., `FileReader`, `BufferedWriter`).

Identifying the Type of Stream

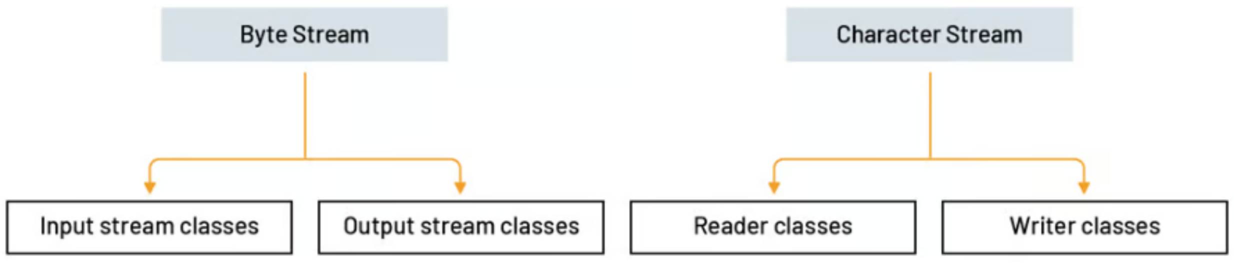
In Java, you can easily recognize whether a stream is a Byte stream or a Character stream by looking at the class name:

- **Byte Stream** classes always contain the word **`Stream`** in their names (e.g., `InputStream`, `OutputStream`, `FileOutputStream`).

- **Character Stream** classes include either **`Reader`** or **`Writer`** in their names (e.g., `FileReader`, `BufferedWriter`, `InputStreamReader`).

This naming convention helps you quickly identify the type of stream being used.

Below is a simple classification of commonly used classes for each type of stream:



Types of Streams

There are two fundamental types of streams in Java:

- **Input Stream**: Used to read data from a source (e.g., `FileInputStream`, `BufferedReader`).

- **Output Stream**: Used to write data to a destination (e.g., `FileOutputStream`, `BufferedWriter`).

Java provides various classes in the `java.io` and `java.nio.file` packages for file handling.

Note: Each type of stream is further categorized based on the kind of data it handles.

1. Input Streams

Using FileInputStream (Byte Stream)

```
import java.io.*;                                            // Ummed Singh
public class FileReadDemo {
   public static void main(String[] args) {
       try (FileInputStream fis = new
FileInputStream("input.txt")) {
           int content;
           while ((content = fis.read()) != -1) {
               System.out.print((char) content);
           }
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

Using BufferedReader (Character Stream)

```
import java.io.*;                                            // Ummed Singh
public class BufferedFileReadExample {
   public static void main(String[] args) {
       try (BufferedReader reader = new BufferedReader(new
FileReader("input.txt"))) {
           String line;
           while ((line = reader.readLine()) != null) {
               System.out.println(line);
           }
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

2. Output Streams

Using FileOutputStream (Byte Stream)

```
import java.io.*;                                          // Ummed Singh
public class FileWriteDemo {
   public static void main(String[] args) {
       try (FileOutputStream fos = new
FileOutputStream("output.txt")) {
           String data = "Hello, File I/O!";
           fos.write(data.getBytes());
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

Using BufferedWriter (Character Stream)

```
import java.io.*;                                          // Ummed Singh
public class BufferedFileWriteExample {
   public static void main(String[] args) {
       try (BufferedWriter writer = new BufferedWriter(new
FileWriter("output.txt"))) {
           writer.write("Java File Handling Example");
       } catch (IOException e) {
           e.printStackTrace();
       }
   }
}
```

Object Serialization

Serialization

Serialization is the process of converting an object into a byte stream. This byte stream can be:
✅ **Stored** in a file or database
✅ **Sent** over a network
✅ **Transferred** between different parts of a system

Deserialization is the reverse process—converting the byte stream back into an object.

Real-World Example:

📌 Use Case: Saving User Data in an Application

Imagine you're developing a **game application**. When a player quits the game, you want to **save their progress** (name, score, level, etc.).

- 🔹 **Serialization** helps store the player's progress into a file/database.
- 🔹 When the player opens the game again, **deserialization** restores their progress.

📍 **Other Use Cases:**
✅ **Sending Objects over a Network:** Online multiplayer games, API communication, and remote method invocation (RMI).
✅ **Caching Data:** Store Java objects in memory for quick access.
✅ **Database Persistence:** Store Java objects in NoSQL databases like MongoDB.

How to Make a Class Serializable

To make a class serializable:
✅ **Implement `Serializable` interface** (It's a marker interface, so no methods need to be implemented).
✅ **Use `ObjectOutputStream`** to serialize (write) an object.
✅ **Use `ObjectInputStream`** to deserialize (read) an object.

Java Code Example: Writing & Reading Objects

The following program demonstrates how to serialize and deserialize a `Person` object.

Example of Writing and Reading Objects

```
import java.io.*;                                         // Ummed Singh

// Define a Serializable Class
class Person implements Serializable {
   private static final long serialVersionUID = 1L;
   String name;
   int age;

   public Person(String name, int age) {
       this.name = name;
       this.age = age;
```

```java
   }

   public void display() {
       System.out.println("Name: " + name + ", Age: " + age);
   }
}

public class SerializationExample {
   public static void main(String[] args) {
       Person p1 = new Person("Alice", 30);

       // Serialize Object
       try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("person.dat"))) {
           oos.writeObject(p1);
           System.out.println("Object Serialized
Successfully.");
       } catch (IOException e) {
           e.printStackTrace();
       }

       // Deserialize Object
       try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("person.dat"))) {
           Person p2 = (Person) ois.readObject();
           System.out.println("Object Deserialized
Successfully.");
           p2.display();
       } catch (IOException | ClassNotFoundException e) {
           e.printStackTrace();
       }
   }
}
```

Explanation of the Code:

📝 Serialization (Saving the Object)

- `ObjectOutputStream` writes the object to a file called **`person.dat`**.

- The object is converted into a byte stream and stored.

📝 **Deserialization (Restoring the Object)**

- `ObjectInputStream` reads the **byte stream** and recreates the `Person` object.
- The object's data (`name` and `age`) is **restored and displayed**.

Important Points About Serialization

✅ `serialVersionUID`

- It helps with version control. If a class changes, the system can detect if it's compatible with an older serialized version.
- If not defined, Java generates one automatically, which can cause compatibility issues if the class is modified later.

✅ Not Everything Can Be Serialized

- **Transient variables** (marked with `transient`) are **not serialized**.
- **Static variables** are **not serialized** (since they belong to the class, not an instance).

✅ Alternatives to Java Serialization

- JSON Serialization (using libraries like Jackson, Gson) is often preferred for web APIs.
- XML-based serialization (used in configurations and data storage).

Generics

Generic in Java

Generics were introduced in Java 5 to enable **compile-time type checking** and eliminate the risk of **ClassCastException**, which was a common issue when working with collection classes. To improve type safety, the entire Collection Framework was updated to support generics. Let's explore how generics allow us to work with collection classes in a safer and more reliable way.

```
public class WithoutGenericDemo {                    // Ummed Singh
   public static void main(String[] args) {
       List list = new ArrayList();
       list.add("abc");
       list.add(5); //OK

       for(Object obj : list){
           //type casting leading to ClassCastException at
runtime
           String str=(String) obj;
       }
   }
}
```

Above code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the elements is of type Integer. After Java 5, we use collection classes like below.

```
List<String> list1 = new ArrayList<String>(); // java 7 ?
List<String> list1 = new ArrayList<>();
       list1.add("abc");
//list1.add(5); //compiler error

       for(String str : list1){
           //no type casting needed, avoids ClassCastException
       }
```

Notice that at the time of list creation, we have specified that the type of elements in the list will be String. So if we try to add any other type of object in the list, the program will throw a compile-time error. Also notice that in for loop, we don't need typecasting of the element in the list, hence removing the ClassCastException at runtime.

Java Generic Class

We can define our own classes with generic types. A generic type is a class or interface that is parameterized over types. We use angle brackets (<>) to specify the type parameter. To understand the benefit, let's say we have a simple class as:

```
public class GenericsTypeOld {                          // Ummed Singh
   private Object t;
   public Object get() {
       return t;
   }
   public void set(Object t) {
       this.t = t;
   }
   public static void main(String args[]){
       GenericsTypeOld type = new GenericsTypeOld();
       type.set("Ummed");
       String str = (String) type.get(); //type casting, error
prone and can cause ClassCastException

       type.set(1);
       String str1 = (String) type.get();

   }
}

Output:
Exception in thread "main" java.lang.ClassCastException: class
java.lang.Integer cannot be cast to class java.lang.String
```

Notice that while using this class, we have to use type casting and it can produce ClassCastException at runtime. Now we will use java generic class to rewrite the same class as shown below.

Creating a Custom Generic Class:

A **generic class** allows you to define a class with a type parameter. The type parameter is specified using angle brackets (<>).

```
// Defining a generic class with type parameter T
class Box<T> {
   private T value;                                     // Ummed Singh
```

```
    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
public class CustomGenericClassDemo {

    public static void main(String[] args) {
        // Creating an instance of Box for Integer
        Box<Integer> intBox = new Box<>();
        intBox.setValue(100);
        System.out.println("Integer Value: " +
intBox.getValue());

        Box type1 = new Box(); //raw type
        type1.setValue("Ummed"); //valid
        type1.setValue(24); //valid and autoboxing support

    }
}
```

The Box class is a generic class that allows type-safe operations without the need for type casting. In the main method, we create an instance of Box<Integer>, ensuring only integer values can be assigned. However, when we use a raw type (Box type1 = new Box();), the compiler generates a warning: “Box is a raw type. References to generic type Box<T> should be parameterized.” Using a raw type allows different object types (String, Integer) to be assigned, which defeats the purpose of generics and may lead to runtime errors due to type mismatch. It is recommended to always use parameterized types to ensure type safety.

Java Generic Type:

Java Generic Type Naming convention helps us understand code easily and having a naming convention is one of the best practices of the Java programming language. So generics also come with its own naming conventions. Usually, type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)

- K - Key (Used in Map)
- N - Number
- T - Type
- V - Value (Used in Map)
- S,U,V etc. - 2nd, 3rd, 4th types

Using the Type Inference Diamond (<>) to Create an Object

Type Inference and the Diamond Operator (<>)

Before Java 7, when working with **generics**, we had to explicitly specify the type on **both** sides of the object declaration:

```
Box<Integer> boxObj = new Box<Integer>();  // Explicit type declaration (before Java 7)
```

From **Java 7 onwards**, the **diamond operator (<>)** was introduced, which allows the compiler to infer the type automatically:

```
Box<Integer> boxObj = new Box<>();  // Compiler infers that it's Box<Integer>
```

🔹 **Type Inference** means that Java can automatically determine the type from the context, so we don't need to **repeat** it on the right-hand side.

Why is the Diamond Operator (<>) Useful

✅ **Reduces Code Repetition:** Before Java 7, we had to specify the type twice. The diamond operator removes redundancy.
✅ **Improves Readability:** The code becomes cleaner and easier to read.
✅ **Prevents Type Errors:** Java still enforces type safety, preventing accidental errors.

What Problem Did the Diamond Operator Solve

🔴 Problem Before Java 7

If we forgot to specify the type on the right side, Java **assumed Object type**, which led to unsafe type casting:

```
Box<Integer> intBox = new Box();   // No type specified (RAW type)

intBox.setValue(10);

int value = (Integer) intBox.getValue();   // Needs explicit casting
```

This **could cause runtime errors** if the wrong type was used.

🟢 Solution in Java 7+ (Using <>)

With the diamond operator, Java **enforces type safety** at compile-time, preventing errors:

```
Box<Integer> intBox = new Box<>();   // Compiler infers Integer type

intBox.setValue(10);

int value = intBox.getValue();   // No explicit casting needed!
```

✅ **No casting required**
✅ **Compile-time type checking**

Real-World Example: Where is the Diamond Operator Used?

📌 Example : E-Commerce Product Inventory (Using Generics)

Imagine you're developing an **E-commerce application** that manages a list of **product prices**.

Before Java 7:

```
List<Double> productPrices = new ArrayList<Double>();  // Old
way
```

After Java 7 (Using <>):

```
List<Double> productPrices = new ArrayList<>();  // Cleaner and
safer
```

Java Code Example: Diamond Operator in Action

```
// Defining a generic class with type parameter T
class Box<T> {
   private T value;

   public void setValue(T value) {
       this.value = value;
   }

   public T getValue() {
       return value;
   }
}
public class DiamondOperatorExample
{

   public static void main(String[] args) {
       // Creating an instance of Box for Integer
       Box<Integer> intBox = new Box<>();
       intBox.setValue(100);
       System.out.println("Integer Value: " +
intBox.getValue());

       // Using List with Diamond Operator
       List<String> names = new ArrayList<>();  // Compiler
infers String type
      names.add("Ummed");
      names.add("Singh");
      System.out.println("Names: " + names);

   }
}

Output:
Integer Value: 100
Names: [Ummed, Singh]
```

Using Bounded Types (extends)

Bounded Type Parameter (extends)?

In Java, **generics** allow you to create classes, interfaces, and methods that operate on **different types**.
 However, sometimes you want to **restrict** the allowed types to a **specific group** (e.g., only numbers, only animals, etc.).
 This is where **bounded type parameters** (extends) come in.

🔹 **Syntax:**

```
class ClassName<T extends SuperClass> { ... }
```

This means **T can only be a subclass of SuperClass**.

2. What Problem Did Bounded Types Solve?

Before Java Generics, we had to rely on Object, which meant we could pass **any** type, leading to **runtime errors**.
 With bounded generics, the compiler **restricts types at compile-time**, preventing incorrect type usage.

🔴 **Problem Before Bounded Types**

```
public class NumericBox<T> {  // No restrictions!

   private T num;
```

```
    public NumericBox(T num) {

        this.num = num;

    }

    public double square() {

        return num * num;  // ERROR: Cannot multiply an Object!

    }

    public static void main(String[] args) {

    }

}
```

💥 **Issue:** The compiler doesn't know if T supports mathematical operations like multiplication.

🟢 **Solution: Use extends Number to Restrict to Numeric Types**

```
class NumericDemo<T extends Number> {  // Now T must be a Number
(Integer, Double, Float, etc.)
    private T num;
    public NumericDemo(T num) {
        this.num = num;
    }
    public double square() {
        return num.doubleValue() * num.doubleValue();  // Safe
operation
    }
}
```

✅ **Now, T is guaranteed to have doubleValue(), making the code safe!**

3. Real-World Examples of Bounded Generics

📌 Example 1: Financial Application (Calculating Interest)

A **banking application** calculates interest for **savings accounts, loans, or credit cards**.
We want to ensure that **only numeric values** are allowed in calculations.

```
class InterestCalculator<T extends Number> {       // Ummed Singh

   private T principal;

   public InterestCalculator(T principal) {

       this.principal = principal;

   }

   public double calculateInterest(double rate) {

       return principal.doubleValue() * rate / 100;

   }

}


public class BankApplication {

   public static void main(String[] args) {

       InterestCalculator<Integer> savingsAccount = new
InterestCalculator<>(5000);

       System.out.println("Interest: ₹" +
savingsAccount.calculateInterest(5));



       InterestCalculator<Double> loan = new
InterestCalculator<>(12000.75);

       System.out.println("Interest: ₹" +
loan.calculateInterest(3.5));
```

```
        // Error: String is not allowed

        // InterestCalculator<String> invalid = new
InterestCalculator<>("Ten Thousand");

    }

}

Output:

Interest: ₹250.0

Interest: ₹420.02625
```

✅ **Prevents invalid data (like String) from being passed, ensuring correctness.**

📌 **Example 2: E-Commerce (Calculating Discount on Products)**

An e-commerce system applies discounts **only to numeric prices**.

```
class DiscountCalculator<T extends Number> {

    private T price;

    public DiscountCalculator(T price) {

        this.price = price;

    }

    public double applyDiscount(double discountPercent) {

        return price.doubleValue() - (price.doubleValue() *
discountPercent / 100);

    }

}
```

```
public class EcommerceApplication {

   public static void main(String[] args) {

       DiscountCalculator<Integer> product1 = new
DiscountCalculator<>(100);

       System.out.println("Discounted Price: ₹" +
product1.applyDiscount(10));



       DiscountCalculator<Double> product2 = new
DiscountCalculator<>(499.99);

       System.out.println("Discounted Price: ₹" +
product2.applyDiscount(15));

   }

}



Output:

Discounted Price: ₹90.0

Discounted Price: ₹424.9915
```

✅ **Ensures that only valid number types are used for price calculations.**

Using Wildcards (?)

1. What Are Wildcards (?) in Java Generics?

Wildcards (?) in generics allow us to work with **unknown types**. They make generic classes and methods **more flexible** by letting them accept a **range of different types** rather than being restricted to a single type.

🔹 **Why Use Wildcards?**

- **Provides Flexibility**: Allows working with **different types** without changing the method signature.
- **Improves Code Reusability**: A single method can handle **multiple types**.
- **Ensures Type Safety**: Prevents unsafe operations while still supporting different data types.

🔹 **Types of Wildcards in Java**

Wildcard	Meaning	Example Usage
`?` (Unbounded Wildcard)	Accepts **any** type.	`List<?>` → Can hold `List<Integer>`, `List<String>`, etc.
`? extends Type` (Upper Bounded Wildcard)	Accepts `Type` and **its subclasses**.	`List<? extends Number>` → Accepts `List<Integer>`, `List<Double>`, but **not `List<String>`**.
`? super Type` (Lower Bounded Wildcard)	Accepts `Type` and **its superclasses**.	`List<? super Integer>` → Accepts `List<Number>`, `List<Object>`, but **not `List<Double>`**.

Problems Solved by Wildcards in Java Generics

🔴 **Problem Before Wildcards:**
Without wildcards, generic methods are **too restrictive**. They require **exact type matches**, which limits flexibility.

🔹 **Example: Restrictive Method Without Wildcards**

```
public class WithoutWildcardsDemo {                    // Ummed Singh

    public static void printList(List<Object> list) {   // Accepts only List<Object>
        for (Object obj : list) {
            System.out.println(obj);
        }
    }
```

```
   public static void main(String[] args) {

   }
}
```

💥 **Issue:** This method **only** works with List<Object>, **not** List<Integer> or List<String>.

✅ **Solution:** Use **wildcards** (?) to allow any type.

---

3. Types of Wildcards and Their Real-World Use Cases

3.1 Unbounded Wildcard (?)

🔹 **When to Use?**

- When **the type doesn't matter** (we only need to **read** from the list).
- Useful for **printing, logging, and debugging**.

🔹 **Real-World Use Case:**
Imagine a **logging system** that needs to print **any** list of values (Strings, Integers, Doubles, etc.).

```
class Printer {
   public static void printList(List<?> list) {  // Accepts any
type of list
       for (Object obj : list) {
           System.out.println(obj);
       }
   }
}
public class UnboundedWildcardDemo {
       public static void main(String[] args) {
           List<Integer> intList = Arrays.asList(1, 2, 3);
           List<String> strList = Arrays.asList("A", "B", "C");

           Printer.printList(intList);  // Works with Integer
List
```

```
            Printer.printList(strList);   // Works with String
List

    }

}

Output:
1
2
3
A
B
C
```

🔹 **What Problem Does It Solve?**

- Allows a **single method** to print **any type** of list.
- Eliminates the need for **overloading the method** for different types.

3.2 Upper Bounded Wildcard (`? extends Type`)

🔹 **When to Use?**

- When **we need to read** from a collection but **not modify it**.
- When a method works on **subtypes** of a given class.

🔹 **Real-World Use Case:**
A **mathematical utility** that calculates the sum of **any list of numbers (Integer, Double, Float, etc.).**

```
class MathUtils {                                  // Ummed Singh

   public static double sumNumbers(List<? extends Number>
numbers) {  // Accepts Numbers and its subclasses

       double sum = 0.0;
```

```
        for (Number num : numbers) {
            sum += num.doubleValue();  // Safe operation
        }
        return sum;
    }
}
public class UpperBoundedWildcardDemo {
    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1, 2, 3);
        List<Double> doubleList = Arrays.asList(1.5, 2.5, 3.5);

        System.out.println("Sum (Integers): " +
MathUtils.sumNumbers(intList));
        System.out.println("Sum (Doubles): " +
MathUtils.sumNumbers(doubleList));
    }
}


Output:
Sum (Integers): 6.0
Sum (Doubles): 7.5
```

🔹 **What Problem Does It Solve?**

- Ensures the method works **only** with Number subclasses (Integer, Double, etc.).
- Prevents **non-numeric types** (like String) from being passed.

3.3 Lower Bounded Wildcard (? super Type)

🔹 **When to Use?**

- When we **need to add elements** to a collection.
- When we need to accept a **parent type and its subclasses**.

🔹 **Real-World Use Case:**
A **data storage system** that only allows adding **Integer values** but can store them in a List<Number> or List<Object>.

```
class DataStore {                                          // Ummed Singh
   public static void addNumbers(List<? super Integer> list) {
// Accepts Integer and its superclasses
       list.add(10);
       list.add(20);
       // list.add(3.5);  // Error: Double is not allowed
   }
}
public class LowerBoundedWildcardDemo {
   public static void main(String[] args) {
       List<Number> numList = new ArrayList<>();
       DataStore.addNumbers(numList);
       System.out.println(numList);
   }

}

Output:
[10, 20]
```

🔹 **What Problem Does It Solve?**

- Allows adding **Integer values** to a collection **without breaking type safety**.
- Ensures that `List<Number>` and `List<Object>` can **store Integer values safely**.

Key Benefits of Using Wildcards in Java

✅ **Flexibility**: Wildcards make methods work with **multiple data types**.
✅ **Type Safety**: Prevents **unsafe type conversions** and **runtime errors**.
✅ **Reusability**: A **single method** can handle multiple types instead of writing duplicate methods.