```
import java.util.*;

// ======================
// 1. Generic Class
// ======================
class Box<T> {
    private T value;

    public Box(T value) {       // Generic constructor
        this.value = value;
    }

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

    public T get() {
        return value;
    }
}

// ======================
// 2. Generic Interface
// ======================
interface Pair<K, V> {
    K getKey();
    V getValue();
}
```

```
class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() { return key; }
    public V getValue() { return value; }
}

// ======================
// 3. Generic Method
// ======================
class Utils {
    public static <T> void printItem(T item) {
        System.out.println("Item: " + item);
    }

    // ======================
    // 4. Bounded Generics
    // ======================
    public static <T extends Number> void showNumber(T num) {
        System.out.println("Number: " + num);
    }
```

```
    // ======================
    // 5. Wildcards
    // ======================
    public static void printListWildcard(List<?> list) {
        System.out.println("Wildcard list: " + list);
    }

    public static void printUpperBound(List<? extends Number> list) {
        System.out.println("Upper bounded list: " + list);
    }

    public static void printLowerBound(List<? super Integer> list) {
        System.out.println("Lower bounded list: " + list);
    }
}

// ======================
// 6. MAIN — FULL DEMO
// ======================
public class GenericsDemo {
    public static void main(String[] args) {

        // --- Generic Class ---
        Box<String> box1 = new Box<>("Hello");
        Box<Integer> box2 = new Box<>(123);
        System.out.println("Box1: " + box1.get());
        System.out.println("Box2: " + box2.get());
```

```
        // --- Generic Interface ---
        Pair<String, Integer> p = new OrderedPair<>("Age", 25);
        System.out.println("Key: " + p.getKey() + ", Value: " + p.getValue());

        // --- Generic Method ---
        Utils.printItem("Generic Method Example");
        Utils.printItem(999);

        // --- Bounded Generics ---
        Utils.showNumber(55);       // Integer
        Utils.showNumber(12.5);     // Double

        // --- Wildcards ---
        List<String> list1 = Arrays.asList("A", "B", "C");
        List<Integer> list2 = Arrays.asList(10, 20, 30);
        List<Number> list3 = Arrays.asList(1, 2.2, 3.3);

        Utils.printListWildcard(list1);
        Utils.printListWildcard(list2);

        Utils.printUpperBound(list2);    // OK: Integer extends Number
        Utils.printUpperBound(list3);    // OK: Number extends Number

        Utils.printLowerBound(list3);    // OK: Number is super of Integer
        Utils.printLowerBound(new ArrayList<Object>()); // OK
    }
}
```

📘 THEORETICAL NOTES ON GENERICS (FULL EXAM MATERIAL)

Below is a **well-structured theory section**, exactly how you would write in your exam.

🔶 1. Introduction to Generics

Generics were introduced in **Java 5** to provide **type safety** and **compile-time checking** in Java programs.
A Generic allows you to create **classes, interfaces, and methods** that can operate on any data type without sacrificing type safety.

Key Idea:

Generics allow "parameterized types," meaning you can pass a data type as a parameter.

Example:

List<String> list = new ArrayList<>();

🔶 2. Why Generics? (VERY IMPORTANT for exams)

✓ 1. Type Safety

Generics prevent inserting wrong data types at compile time.

✓ 2. No Need for Type Casting

Without generics:

Object obj = list.get(0);

String s = (String) obj;

With generics:

String s = list.get(0);

✓ 3. Code Reusability

A single class works for multiple data types.

✓ 4. Compile-time Checking

Errors are detected early (before runtime).

🔶 3. Generic Class

A Generic class is a class that takes a type parameter.

Syntax:

```
class ClassName<T> { }
```

Example from the program:

```
class Box<T> {
   private T value;
}
```

Here, T is a placeholder for any data type (String, Integer, etc.).

🔶 4. Generic Constructors

A constructor can also be generic:

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

🔶 5. Generic Interface

You can define type parameters for interfaces too.

Example:

```
interface Pair<K, V> {
   K getKey();
   V getValue();
}
```

This allows implementations to define custom data types:

```
Pair<String, Integer> p = new OrderedPair<>("Age", 25);
```

🔶 6. Generic Methods

A method can be made generic even if the class is NOT generic.

Syntax:

public <T> void methodName(T obj) {}

Example:

public static <T> void printItem(T item)

This increases flexibility.

🔶 7. Bounded Generics (Important Topic)

Bounded generics restrict the type of data allowed.

✓ Upper Bound

<T extends Number>

Means T can be: Number, Integer, Float, Double, etc.

✓ Lower Bound

<? super Integer>

Means the type can be: Integer's parent class
(Number, Object)

🔶 8. Wildcards

Wildcards are used when the exact type is not known.

✓ Unbounded Wildcard

List<?> list

Accepts any List.

✓ Upper Bounded Wildcard

List<? extends Number>

Accepts Number or its subclasses.

✓ Lower Bounded Wildcard

List<? super Integer>

Accepts Integer or its superclasses.

🔶 9. Type Erasure (Theory Question!)

Java generics use **type erasure** internally.
This means:

1. Generic type information is removed at compile time.
2. Compiler replaces type parameters with their bounds:
    - For <T extends Number> → replaced with Number
    - For <T> without bounds → replaced with Object
3. Ensures backward compatibility with older versions of Java.

Example after erasure:

Box<Integer> becomes Box<Object>

🔶 10. Advantages of Generics

1. Type safety
2. No explicit casting
3. Easy-to-maintain code
4. Compile-time checks
5. Code reusability

🔶 11. Disadvantages of Generics

1. Cannot use primitive types directly
   Example: List<int> is not allowed
2. Type erasure can cause issues during reflection
3. Complex syntax

🔶 12. Real-life Uses of Generics

✓ Collection Framework (List<T>, Map<K,V>, Set<T>)
✓ Comparator and Comparable interfaces
✓ Custom Data Structures
✓ Algorithms libraries

🎯 Perfect Exam-Oriented Summary (Write this in your answer)

Generics in Java provide a mechanism for writing type-safe and reusable code by allowing classes, interfaces, and methods to operate on user-specified types. They offer compile-time type checking, remove the need for casting, and make code more readable and maintainable. Key components include generic classes, generic methods, bounded types, and wildcards. Internally, Java implements generics through **type erasure**, where generic information is removed at compile time to maintain backward compatibility. Generics form the foundation of the Java Collection Framework and many modern Java libraries.

Below is a **complete, exam-ready explanation of WILDCARDS in Java Generics**, written in simple language with definitions, types, examples, diagrams, and important theory points.

⭐ WILDCARDS IN JAVA GENERICS (FULL THEORY + EXAMPLES)

A **wildcard** in Java generics is represented by a **question mark (?)** and is used when **the exact type parameter is not known** or **we want to allow flexibility in accepting multiple types**.

🔶 Definition (Exam Answer)

A **wildcard (?)** in Java generics is a special type argument that represents an *unknown type*. It is used when we want a generic method or class to work with **a range of types**, not just one specific type.

Wildcards provide flexibility while still maintaining type safety.

📌 Why are Wildcards needed?

✓ To read data from generic collections without knowing exact type

✓ **To allow polymorphism in generics (since List<Object> ≠ List<String>)**

✓ **To handle parent-child relationships in generics**

✓ **To restrict what types are acceptable (bounded wildcards)**

⭐ TYPES OF WILDCARDS

There are **three types**:

1️⃣ Unbounded Wildcard (?)

✓ **Used when we don't care about the type**

✓ **Can read data, but cannot add anything except null**

Syntax:

```
List<?> list;
```

Example:

```
public static void printList(List<?> list) {
   System.out.println(list);
}
```

Usage:
Works with List<Integer>, List<String>, List<Dog>, etc.

2️⃣ Upper Bounded Wildcard (? extends T)

✓ **Means "T or any subclass of T"**

✓ **Used when only *reading* is allowed**

✓ **You cannot add items (except null)**

Syntax:

```
List<? extends Number>
```

This means the list may contain:

- Integer

- Float
- Double
- Number

Example:

```
public static void processNumbers(List<? extends Number> list) {
   for (Number n : list) {
      System.out.println(n);
   }
}
```

Rule:

PECS rule: Producer Extends
If a list produces values → use extends.

3️⃣ Lower Bounded Wildcard (? super T)

✓ Means "T or any superclass of T"

✓ Used when we want to *add* items

✓ Reading gives only Object

Syntax:

List<? super Integer>

This means the list may contain:

- Integer
- Number
- Object

Example:

```
public static void addNumbers(List<? super Integer> list) {
   list.add(10);  // allowed
   list.add(20);  // allowed
}
```

Rule:

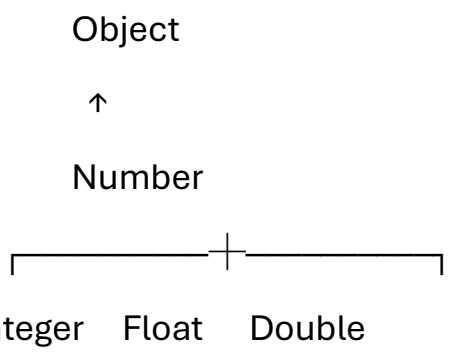
PECS rule: Consumer Super
If a list consumes values → use super.

🌟 **PECS Rule (VERY IMPORTANT FOR EXAMS!)**

PECS = Producer Extends, Consumer Super

Condition	Wildcard to Use
List is only **producing** (reading)	? extends T
List is **consuming** (adding)	? super T

🔷 **DIAGRAM (VERY SIMPLIFIED)**



✓ **List<? extends Number>**

Can refer to lists of: Number, Integer, Float, Double

✓ **List<? super Number>**

Can refer to lists of: Number, Object

⭐ **EXAMPLES OF EACH WILDCARD**

✓ **Unbounded Wildcard**

```
List<?> list = new ArrayList<String>();
```

✓ **Upper Bounded**

```
List<? extends Number> nums = new ArrayList<Integer>();
```

✓ Lower Bounded

```
List<? super Integer> values = new ArrayList<Object>();
```

⭐ COMPLETE PROGRAM USING ALL WILDCARDS

```
import java.util.*;

class WildcardDemo {

  // Unbounded
  public static void printList(List<?> list) {
    System.out.println(list);
  }

  // Upper bounded: reading allowed
  public static void showNumbers(List<? extends Number> list) {
    for (Number n : list) {
      System.out.println(n);
    }
  }

  // Lower bounded: adding allowed
  public static void addValues(List<? super Integer> list) {
    list.add(10);
    list.add(20);
  }
```

```
    public static void main(String[] args) {
        List<String> list1 = Arrays.asList("A", "B", "C");
        List<Integer> list2 = Arrays.asList(1, 2, 3);

        printList(list1);    // works
        printList(list2);    // works

        showNumbers(list2);  // upper bound

        List<Object> list3 = new ArrayList<>();
        addValues(list3);    // lower bound
        System.out.println(list3);
    }
}
```

⭐ ADVANTAGES OF WILDCARDS

1. Provide flexibility in generics
2. Allow methods to accept a wide range of types
3. Support polymorphism in generics
4. Help with reading or writing restrictions
5. Increase reusability of code

⭐ DISADVANTAGES OF WILDCARDS

1. Cannot always add items to wildcard lists
2. Harder to understand for beginners
3. Complicated when mixed with multiple bounds

4. Sometimes requires type casting

🎯 SHORT EXAM SUMMARY (WRITE THIS IN 3 MARKS)

Wildcards in Java Generics are represented by (?) and denote an unknown type.
They are used to increase flexibility in methods and classes.
Three types of wildcards exist:
(1) Unbounded wildcard (?) – any type allowed;
(2)Upper bounded wildcard (? extends T) – T and its subclasses, used for reading;
(3) Lower bounded wildcard (? super T) – T and its superclasses, used for adding.
Wildcards follow the PECS rule: Producer Extends, Consumer Super.
