Java Generics

Introduction

- Generics in Java enable the creation of classes, interfaces, and methods with type parameters.
- This feature allows for type safety, code reusability, better readability and elimination of type casting.
- Generics were introduced in Java 5 and are widely used in the Java Collections Framework.

Generic Class

Example 1: Simple Generic Class

First, let's define a simple generic class Box that can hold an item of any type.

```
// Defining a generic class Box
public class Box<T> {
    private T value;

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

    public T getValue() {
        return value;
    }

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

public class Main {
    public static void main(String[] args) {
        // Creating a Box object for Integer
        Box<Integer> intBox = new Box<>(10);  // Type is Integer
        System.out.println("Integer Box: " + intBox.getValue());

        // Creating a Box object for String
        Box<String> strBox = new Box<>("Hello World");  // Type is String
        System.out.println("String Box: " + strBox.getValue());

        // Creating a Box object for Double
        Box<Double> doubleBox = new Box<>(3.14);  // Type is Double
        System.out.println("Double Box: " + doubleBox.getValue());
    }
}
```

Explanation:

- **Box<Integer> intBox = new Box<>(10)**: Here, we're creating an instance of the Box class where T is replaced by Integer. So Box<Integer> means the Box will hold an Integer.
- **Box<String> strBox = new Box<>("Hello World")**: This creates a Box for String values.
- **Constructor Call (new Box<>(value))**: The constructor is called with the specific type (Integer, String, Double), and the object is initialized.

```
Box<Integer> intBox = new Box<Integer>(10);
```

This is **perfectly valid** and works as expected. Here's a breakdown:

- **Box<Integer>**: This specifies that Box is a generic class and the type parameter T is Integer.
- **new Box<Integer>(10)**: This creates a new instance of the Box class, where T is Integer. The constructor is called with the value 10, which is of type Integer.

The Cleaner, More Common Way: Since Java allows **type inference**, the **Integer** type is already clear from the left side of the declaration (Box<Integer>), so you don't need to repeat it on the right side when creating the object.

Example 2: Multiple Type Parameters
Sometimes, you may want a class to accept multiple type parameters. Let's define a class Pair that holds two different types.

```
// Defining a generic class Pair with two type parameters
public class Pair<K, V> {
    private K key;
    private V value;

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

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating a Pair object with Integer and String
        Pair<Integer, String> pair1 = new Pair<>(1, "One");
        System.out.println("Key: " + pair1.getKey() + ", Value: " +
pair1.getValue());

        // Creating a Pair object with String and Double
        Pair<String, Double> pair2 = new Pair<>("Pi", 3.14159);
        System.out.println("Key: " + pair2.getKey() + ", Value: " +
pair2.getValue());
    }
}
```

Explanation:

- **Pair<Integer, String> pair1 = new Pair<>(1, "One")**: Here, Pair is a generic class with two type parameters: K (for the key) and V (for the value). In this case, we specify K as Integer and V as String.
- **Pair<String, Double> pair2 = new Pair<>("Pi", 3.14159)**: Here, the key is a String and the value is a Double.

Example 3: Generic Class with Bounded Type Parameters
You can also constrain the types a generic class can accept by using **bounded types**. For example, you can limit the class to only accept types that extend Number.

```
// Defining a generic class that only accepts types that extend Number
public class NumericBox<T extends Number> {
    private T value;

    public NumericBox(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        // This works because Integer extends Number
        NumericBox<Integer> intBox = new NumericBox<>(10);
        System.out.println("Integer Box: " + intBox.getValue());

        // This works because Double extends Number
        NumericBox<Double> doubleBox = new NumericBox<>(3.14);
        System.out.println("Double Box: " + doubleBox.getValue());

        // This will result in a compile-time error because String does not
extend Number
        // NumericBox<String> strBox = new NumericBox<>("Hello");
    }
}
```

Explanation:

- **NumericBox<Integer> intBox = new NumericBox<>(10)**: The NumericBox accepts only types that extend Number, so Integer and Double work here.
- **Compile-time error for String**: Trying to create a NumericBox for String would result in an error, because String doesn't extend Number.

Other Variations of Generic Object Creation

You may also encounter **raw types** or **wildcard types** in generic class instantiation. Here are some examples:

1. **Raw Type (Not Recommended)**:

```
Box rawBox = new Box(10);  // No type parameter provided
```

This is possible but **not recommended** because you lose the benefits of type safety with generics.

2. **Wildcard Type**:

If you want to accept any subclass of Number, you could use wildcards:

```
// List of any subclass of Number
public class Box<T> {
    private T value;

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

    public T getValue() {
        return value;
    }
}

// Main method
Box<? extends Number> wildcardBox = new Box<>(10); // Can hold any type of
Number (Integer, Double, etc.)
```

But with this approach, you can only **read** from the wildcardBox, not modify it, because the exact type is unknown.

Summary of Syntax

- **Simple Generic Class**: ClassName<Type> object = new ClassName<Type>(value);
- **Multiple Type Parameters**: ClassName<Type1, Type2> object = new ClassName<Type1, Type2>(value1, value2);
- **Bounded Type**: ClassName<Type extends SuperType> object = new ClassName<Type>(value);
- **Wildcard Type**: ClassName<?> object = new ClassName<>(value);

1. ArrayList<?> l2 = new ArrayList<String>();

- **ArrayList<?>**: This is a **wildcard type**. It represents an **ArrayList** of an unknown type. You don't know or care about the type that the list contains. It's useful when you're working with collections that may have various types, but you're only interested in reading from them (you can't add elements to them except null).
- **new ArrayList<String>()**: This creates an ArrayList that holds String elements.
- **Why it's valid**:
    - **ArrayList<?>** can hold any type of object. Even though we create an ArrayList<String>, it can be assigned to an ArrayList<?> because ? represents an unknown type, and a list of String is valid for that unknown type.

2. ArrayList<?> l3 = new ArrayList<Integer>();

- **ArrayList<?>**: Again, this is a **wildcard** that can hold any type of object.
- **new ArrayList<Integer>()**: This creates an ArrayList that holds Integer elements.
- **Why it's valid**:
    - Just like in the first case, **ArrayList<?>** can hold any type of object, so assigning an ArrayList<Integer> to an ArrayList<?> is perfectly fine. The wildcard ? can represent Integer in this case.

3. ArrayList<? extends Number> l5 = new ArrayList<String>();

- **ArrayList<? extends Number>**: This is a **bounded wildcard**. It means that the list can hold any type that is a subclass of Number (like Integer, Double, etc.), but **not String** because String is not a subclass of Number.
- **new ArrayList<String>()**: This creates an ArrayList that holds String elements.
- **Why it's **invalid**:
    - String does not extend Number. Since ArrayList<? extends Number> expects a list that holds Number or its subclasses (like Integer, Double), **you cannot assign a list of String to it**. This is a **type mismatch**, and will result in a compile-time error.

4. ArrayList<?> l6 = new ArrayList<? extends Number>();

- **ArrayList<?>**: A list that can hold any type of object (wildcard).
- **new ArrayList<? extends Number>()**: This is trying to create an ArrayList with a wildcard type that extends Number.
- **Why it's **invalid**:
    - This is invalid because **you cannot use a wildcard type (? extends T) as the actual type argument when creating a new object**. The type parameter (? extends Number) is used to specify the type of a collection, but **when creating a new instance**, the type must be concrete (like Integer, String, or Double), not a wildcard.
    - The correct way would be to use a concrete type like ArrayList<Number> or ArrayList<Integer>, but you can't instantiate an ArrayList using a wildcard directly like this.

5. ArrayList<?> l7 = new ArrayList<?>();

- **ArrayList<?>**: A list that can hold any type of object (wildcard).
- **new ArrayList<?>()**: This is trying to create an ArrayList with a wildcard type.
- **Why it's **invalid**:
    - Similar to the previous case, you **cannot instantiate a generic type with a wildcard**. Wildcards (?) are only valid for **referencing** the type, not for creating a new instance.
    - You must provide a concrete type, such as ArrayList<Object>, ArrayList<Integer>, or ArrayList<String>, when creating an ArrayList.

Key Takeaways:

1. **Wildcards (?)** are for flexibility in referencing types, but you can't use them when creating an object. You need a concrete type (like Integer, String, etc.) to instantiate a generic class.
2. **Bounded Wildcards (? extends T)** are useful when you want to restrict the types to a specific range (like all subclasses of Number), but they cannot be used for object instantiation.
3. **Wildcards are useful for reading from a collection**, but not for writing (except null).

Generic method

Generic methods allow you to define a method with a placeholder for the type of object it will operate on, making the method flexible and reusable for different data types. This approach is a part of Java's **Generics** mechanism, which enables type-safe code without the need to cast objects.

The syntax for defining a generic method is as follows:

```
<typeParameter> returnType methodName(parameters)
```

- **<typeParameter>**: This is the placeholder for the type that will be used in the method.
- **returnType**: The type of the value the method returns (which can also be a generic type).
- **methodName**: The name of the method.
- **parameters**: The parameters the method will accept (can also be of generic types).

The type parameter is usually denoted by a single letter, commonly T (for Type), but it can be anything (like E, K, V, etc.).

Example 1: Basic Generic Method

Let's look at an example where we have a method that prints out the value of any type.

```
public class GenericExample {

    // Generic method to print any type of data
    public static <T> void print(T data) {
        System.out.println(data);
    }

    public static void main(String[] args) {
        print(10);           // Integer
        print("Hello!");     // String
        print(3.14);         // Double
    }
}
```

Explanation:

1. **<T>**: This specifies a type parameter T (you can choose any letter here). It means the method can work with any type.
2. **print(T data)**: The method takes a parameter data of type T and prints it. The type T will be determined at runtime based on what is passed into the method.
3. **Method calls**: In the main method, we call print() with different data types (Integer, String, Double). The compiler automatically infers the type when the method is called.

Example 2: Generic Method with Multiple Parameters

You can also have multiple type parameters in a generic method.

```
public class GenericExample {

    // Generic method with two type parameters
    public static <T, U> void printPair(T first, U second) {
        System.out.println("First: " + first + ", Second: " + second);
    }

    public static void main(String[] args) {
        printPair(10, "Hello");     // Integer and String
        printPair(3.14, 'A');       // Double and Character
    }
}
```

Explanation:

1. **<T, U>**: Two type parameters, T and U, are defined. This means that the method can accept parameters of different types.

2. **printPair(T first, U second)**: The method takes two parameters of types T and U, respectively.
3. **Method calls**: We pass different pairs of data types like Integer and String, Double and Character.

Example 3: Generic Method with Bound
You can restrict the type that a generic method accepts using **bounded types**. For example, you can specify that the type must be a subclass of a particular class.

```
public class GenericExample {

    // Generic method with a bounded type parameter
    public static <T extends Number> void printNumber(T number) {
        System.out.println("Number: " + number);
    }

    public static void main(String[] args) {
        printNumber(10);      // Integer
        printNumber(3.14);    // Double
        // printNumber("Hello");  // Error: String is not a subclass of Number
    }
}
```

Explanation:
1. **<T extends Number>**: This bounds the generic type T to be a subclass of Number. This ensures that the method can only accept types like Integer, Double, Float, etc.
2. **Method call**: The method works with Integer and Double, but trying to pass a String would result in a compile-time error.

Example 4: Generic Method Returning a Value
Generic methods can also return a value.

```
public class GenericExample {

    // Generic method that returns the same type
    public static <T> T identity(T value) {
        return value;
    }

    public static void main(String[] args) {
        Integer x = identity(10);        // Integer
        String y = identity("Hello");    // String
        System.out.println(x);            // Prints 10
        System.out.println(y);            // Prints Hello
    }
}
```

Explanation:
1. **<T> T identity(T value)**: The method returns the same type T that is passed as a parameter.
2. **Method call**: We pass different types (Integer, String), and the method returns them back, demonstrating the flexibility of the generic method.

Key Points to Remember:
- **Type Safety**: Generics ensure that you avoid ClassCastException at runtime because type checking is done at compile-time.
- **Wildcards**: You can also use wildcards (?) for more flexible method signatures, such as ? extends T (upper-bounded wildcard) or ? super T (lower-bounded wildcard).
- **Reusability**: Generic methods allow code to be reused with any type, without needing to write type-specific versions of the method.

Example 5: Generic Methods with Wildcards

```
public class GenericExample {

    // Method that accepts any list of objects that are either a type of Number
or its subclass
    public static void printNumbersList(List<? extends Number> list) {
        for (Number number : list) {
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        List<Integer> intList = List.of(1, 2, 3);
        List<Double> doubleList = List.of(1.1, 2.2, 3.3);

        printNumbersList(intList);   // Works for Integer
        printNumbersList(doubleList);  // Works for Double
    }
}
```

Explanation:

1. **List<? extends Number>**: This is an example of an upper-bounded wildcard. The method accepts any list of objects that are a subtype of Number.
2. **Method call**: We can pass a list of Integer or Double, but not a list of String because String is not a subtype of Number.

Wildcards

In Java, **wildcards** are a special kind of type parameter that provides flexibility when working with generics. They allow you to represent unknown types, enabling more general and flexible methods or classes while maintaining some level of type safety.

Wildcards are represented by the ? symbol, and they are different from **type parameters** (like T, E, K, etc.) in that wildcards are used when you don't know or don't care about the specific type. In contrast, type parameters allow you to define specific, reusable types for a method or class.

Difference Between Type Parameters (T) and Wildcards (?)

1. **Type Parameters (T)**:
    - A **type parameter** is a placeholder that you define when creating a generic method or class.
    - The type parameter is resolved when you use the method or class, and it is used to define specific types that the method/class operates on.
    - Example: <T>, <E>, <K, V>, etc.
2. **Wildcards (?)**:
    - A **wildcard** represents an unknown type and is used when you don't need to specify a particular type, but still want to allow for some flexibility with types.
    - Wildcards are typically used in method arguments (especially collections) where you don't need to know the exact type of the elements, but just that they fall within a certain type hierarchy (or that they are of a certain type).

Types of Wildcards

There are mainly **three types** of wildcards:

1. **Unbounded Wildcard (?)**:
    - This means "any type". The method or class can accept any type, but you can't make assumptions about what type it is.
    - **Example**: List<?> can be a List<Integer>, List<String>, List<Double>, etc.
2. **Upper Bounded Wildcard (? extends T)**:
    - This means "any type that is a subtype of T" (including T itself).
    - **Example**: List<? extends Number> can be a List<Integer>, List<Double>, List<Float>, etc. It allows you to pass any list where the elements are of type Number or its subclasses.

3. **Lower Bounded Wildcard (? super T)**:
    - This means "any type that is a supertype of T" (including T itself).
    - **Example**: List<? super Integer> can be a List<Number> or List<Object>, but not a List<Double>. This restricts the list to be of Integer or its superclasses.

Example 1: Unbounded Wildcard (?)
This is useful when you don't care about the type.

```
import java.util.List;

public class WildcardExample {

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

    public static void main(String[] args) {
        List<Integer> intList = List.of(1, 2, 3);
        List<String> stringList = List.of("A", "B", "C");

        printList(intList);    // Works with List<Integer>
        printList(stringList); // Works with List<String>
    }
}
```

Explanation:

- **List<?>**: This means the list can hold any type of object, but you cannot perform any operations that depend on the specific type (like adding elements, unless they are null).
- **Flexibility**: This is a very general form where we simply iterate over the list, and the type of elements doesn't matter.

Example 2: Upper Bounded Wildcard (? extends T)
This is used when you want to limit the accepted type to be a subclass of a specific type.

```
import java.util.List;

public class WildcardExample {

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

    public static void main(String[] args) {
        List<Integer> intList = List.of(1, 2, 3);
        List<Double> doubleList = List.of(1.1, 2.2, 3.3);

        printNumbers(intList);    // Works with List<Integer>
        printNumbers(doubleList); // Works with List<Double>
    }
}
```

Explanation:

- **List<? extends Number>**: This means the list can be a List of any type that extends Number (like Integer, Double, Float, etc.).

- **Limiting**: You can read elements of the list safely as Number, but you cannot add elements to the list because you don't know the specific subclass of Number (you can't add just any Number because it might be a Double list, for example).

Example 3: Lower Bounded Wildcard (? super T)
This is used when you want to allow types that are superclasses of a certain type.

```
import java.util.List;

public class WildcardExample {

    public static void addNumbers(List<? super Integer> list) {
        list.add(10);  // We can safely add an Integer
    }

    public static void main(String[] args) {
        List<Number> numList = List.of(1, 2, 3);
        List<Object> objList = List.of("A", "B");

        addNumbers(numList);   // Works with List<Number>
        addNumbers(objList);   // Works with List<Object>
    }
}
```

Explanation:
- **List<? super Integer>**: This means the list can be a List of Integer or any superclass of Integer (like Number or Object).
- **Adding Safely**: Since Integer is a subclass of Number and Object, you can add an Integer to such lists, but you can't read elements as anything other than Object (because we don't know the exact type of the list).

Why Use Wildcards Instead of Type Parameters (T)?
- **Flexibility**: Wildcards allow you to create more flexible APIs. They allow the method to accept a broader range of types, unlike a type parameter like T, which would restrict the method to a specific type.
- **Read vs. Write**: With wildcards, the direction of type safety can be controlled:
  - **? extends T** (upper bound) is for **reading**. You can read elements but not modify the collection.
  - **? super T** (lower bound) is for **writing**. You can add elements of type T, but you can't safely read them as anything other than Object.

Summary of Key Differences:

Feature	Type Parameters (T)	Wildcards (?)
Purpose	Represents a specific type, to be determined later	Represents an unknown type, for more flexible APIs
Usage Context	Defined when creating a class or method	Used when you don't care about the type or want more generality
Read/Write	You can read and write with T	Depends on the bound, e.g., ? extends T (read), ? super T (write)
Flexibility	Specific to the type you define	More flexible, allows handling different types in one method

- **Use type parameters (like T) when defining** your generic classes or methods (you decide what the type will be).
- **Use wildcards (? extends Type) when declaring** variables or parameters when you want to allow any type that fits a certain condition but don't need to know the specific type.