**Java Generics**

Java Generics allow you to create classes, interfaces, and methods that operate on a specific type without needing to know the exact type during design time. Generics are a powerful feature in Java, enabling code reusability and type safety.

**Table of Contents**

1. **Introduction to Generics**
2. **Generic Classes**
3. **Generic Methods**
4. **Bounded Types**
5. **Wildcards in Generics**
6. **Generics and Inheritance**
7. **Conclusion**

**1. Introduction to Generics**

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces, and methods. The key benefit of Generics is to ensure type safety and to eliminate the need for casting.

**Example Without Generics**

import java.util.ArrayList;

public class WithoutGenerics {

public static void main(String[] args) {

ArrayList list = new ArrayList();

list.add("Hello");

list.add(123); // This will compile, but it can cause runtime errors

String str = (String) list.get(0); // This cast is necessary

System.out.println(str);

}

}

**Example With Generics**

import java.util.ArrayList;

public class WithGenerics {

public static void main(String[] args) {

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

list.add("Hello");

// list.add(123); // Compile-time error

String str = list.get(0); // No cast needed

System.out.println(str);

}

}

In the above example, with Generics, the type is specified as String, ensuring that only String objects can be added to the list.

**2. Generic Classes**

A generic class can work with any type specified at instantiation time.

**Example of a Generic Class**

class Box<T> {

private T t;

public void set(T t) {

this.t = t;

}

public T get() {

return t;

}

}

public class GenericClassExample {

public static void main(String[] args) {

Box<Integer> integerBox = new Box<>();

integerBox.set(10);

System.out.println("Integer Value: " + integerBox.get());

Box<String> stringBox = new Box<>();

stringBox.set("Hello");

System.out.println("String Value: " + stringBox.get());

}

}

In this example, Box<T> is a generic class where T is a type parameter. You can replace T with any type when creating an object of the Box class.

**3. Generic Methods**

Generic methods can be declared in both generic and non-generic classes.

**Example of a Generic Method**

public class GenericMethodExample {

public static <T> void printArray(T[] inputArray) {

for (T element : inputArray) {

System.out.printf("%s ", element);

}

System.out.println();

}

public static void main(String[] args) {

Integer[] intArray = { 1, 2, 3, 4, 5 };

String[] strArray = { "Hello", "World", "!" };

printArray(intArray);

printArray(strArray);

}

}

Here, <T> is the generic type parameter. The printArray method can accept an array of any type.

**4. Bounded Types**

You can restrict the types that can be used as arguments for a type parameter using bounded types.

**Example of a Bounded Type**

class NumberBox<T extends Number> {

private T t;

public void set(T t) {

this.t = t;

}

public T get() {

return t;

}

public double doubleValue() {

return t.doubleValue();

}

}

public class BoundedTypeExample {

public static void main(String[] args) {

NumberBox<Integer> intBox = new NumberBox<>();

intBox.set(10);

System.out.println("Double value: " + intBox.doubleValue());

NumberBox<Double> doubleBox = new NumberBox<>();

doubleBox.set(3.14);

System.out.println("Double value: " + doubleBox.doubleValue());

}

}

Here, T extends Number means that T can only be of type Number or its subclasses (e.g., Integer, Double).

**5. Wildcards in Generics**

Wildcards (?) represent an unknown type. They can be used in scenarios where you do not know the exact type.

**Example of Wildcards**

import java.util.List;

public class WildcardExample {

public static void printList(List<?> list) {

for (Object elem : list) {

System.out.println(elem);

}

}

public static void main(String[] args) {

List<Integer> intList = List.of(1, 2, 3, 4);

List<String> strList = List.of("A", "B", "C");

printList(intList);

printList(strList);

}

}

Here, List<?> can represent a list of any type. The wildcard allows the method to accept lists of any type.

**Bounded Wildcards**

You can also use bounds with wildcards.

public class BoundedWildcardExample {

public static double sumOfList(List<? extends Number> list) {

double sum = 0.0;

for (Number num : list) {

sum += num.doubleValue();

}

return sum;

}

public static void main(String[] args) {

List<Integer> intList = List.of(1, 2, 3, 4);

System.out.println("Sum: " + sumOfList(intList));

List<Double> doubleList = List.of(1.5, 2.5, 3.5);

System.out.println("Sum: " + sumOfList(doubleList));

}

}

In this example, List<? extends Number> allows any list whose elements are of type Number or a subclass of Number.

**6. Generics and Inheritance**

Generics follow the principle of invariance. For instance, List<String> is not a subtype of List<Object>.

**Example of Invariance**

public class InvarianceExample {

public static void main(String[] args) {

List<String> stringList = List.of("Hello", "World");

// List<Object> objectList = stringList; // Compile-time error

}

}

However, wildcards can be used to overcome this limitation.

**Covariance with Wildcards**

public class CovarianceExample {

public static void printList(List<? extends Object> list) {

for (Object elem : list) {

System.out.println(elem);

}

}

public static void main(String[] args) {

List<String> stringList = List.of("Hello", "World");

printList(stringList);

}

}

Here, List<? extends Object> allows the method to accept a List<String>.

**7. Conclusion**

Java Generics provide a way to create classes, methods, and interfaces that are type-safe and reusable. Understanding how to use generics can greatly improve the robustness and readability of your code. By using bounded types, wildcards, and generic methods, you can create flexible and reusable components that work with any data type.