### ****Generics****

Java Generics provide a way to define classes, interfaces, and methods with type parameters, enabling the creation of reusable and type-safe code. This allows you to write more flexible and maintainable code, as well as reduce runtime errors.

### ****1. Introduction to Generics****

Generics allow you to create classes, interfaces, and methods that work with any data type while ensuring type safety at compile-time. This is achieved by using type parameters, which are placeholders for specific types that will be defined when the class, interface, or method is instantiated or invoked.

Generics are introduced in Java 5. They improve code reusability and type safety, ensuring that objects are type-checked at compile time.

### ****2. Syntax of Generics****

The syntax for generics involves the use of angle brackets (<>) to specify type parameters. These parameters are placeholders for types that will be defined when the class, interface, or method is instantiated.

Example:

// A generic class that works with any data type T

class Box<T> {

private T value;

public void setValue(T value) {

this.value = value;

}

public T getValue() {

return value;

}

}

Here, T is a type parameter that can be replaced with any valid type during instantiation.

### ****3. Generic Classes****

A generic class allows you to define a class with a type parameter that can be replaced by any object type.

Example:

// A generic Box class

class Box<T> {

private T value;

public void setValue(T value) {

this.value = value;

}

public T getValue() {

return value;

}

}

public class P1GenericClassExample {

public static void main(String[] args) {

// Using Box with Integer type

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

intBox.setValue(10);

System.out.println("Value in intBox: " + intBox.getValue());

// Using Box with String type

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

strBox.setValue("Hello, Generics!");

System.out.println("Value in strBox: " + strBox.getValue());

}

}

**Output:**

Value in intBox: 10

Value in strBox: Hello, Generics!

### ****4. Generic Methods****

A generic method is a method that can operate on objects of various types while providing compile-time type safety.

Example:

// Generic method to print the array of any type

public class P2GenericMethodExample {

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

for (T element : array) {

System.out.println(element);

}

}

public static void main(String[] args) {

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

String[] strArray = {"apple", "banana", "cherry"};

// Print Integer array

System.out.println("Integer Array:");

printArray(intArray);

// Print String array

System.out.println("String Array:");

printArray(strArray);

}

}

**Output:**

Integer Array:

1

2

3

4

String Array:

apple

banana

cherry

### ****5. Bounded Type Parameters****

In some cases, you might want to restrict the types that can be used as type parameters. This is known as bounding the type parameter. The bound is specified using the extends keyword.

Example:

// Bounded type parameter that accepts only objects that are instances of Number or its subclasses

public class BoundedTypeExample {

public static <T extends Number> void printDouble(T value) {

System.out.println(value.doubleValue());

}

public static void main(String[] args) {

printDouble(10); // Integer

printDouble(15.5); // Double

}

}

**Output:**

10.0

15.5

**Explanation:** The printDouble method accepts any subclass of Number, such as Integer, Double, etc.

### ****6. Wildcards in Generics****

Sometimes you don’t know the exact type that a parameter will have, but you want to restrict the range of types that can be used. This is where wildcards come into play.

#### 6.1 Upper Bounded Wildcards

You can use an upper-bounded wildcard (? extends Type) to specify that a parameter can accept objects of a specific type or its subclasses.

Example:

// A method to calculate the sum of elements in a list of Numbers or its subclasses

import java.util.List;

public class P3UpperBoundedWildcardExample {

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

double sum = 0.0;

for (Number number : list) {

sum += number.doubleValue();

}

return sum;

}

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);

System.out.println("Sum of Integer list: " + sum(intList));

System.out.println("Sum of Double list: " + sum(doubleList));

}

}

**Output:**

Sum of Integer list: 6.0

Sum of Double list: 6.6

#### 6.2 Lower Bounded Wildcards

A lower-bounded wildcard (? super Type) allows you to specify that a parameter can accept a type or any of its superclasses.

Example:

// A method to add a value to a list of Numbers or any of its supertypes

import java.util.List;

public class P4LowerBoundedWildcardExample {

public static void addToList(List<? super Integer> list) {

list.add(10);

System.out.println(list);

}

public static void main(String[] args) {

List<Number> numberList = new ArrayList<>();

addToList(numberList); // Works with Integer or any superclass of Integer

}

}

**Output:**

[10]

### ****7. Generic Interfaces****

Interfaces can also be made generic. Just like classes, you can define interfaces with type parameters.

Example:

// A generic interface for a repository

interface Repository<T> {

void save(T item);

T find(int id);

}

class ProductRepository implements Repository<Product> {

@Override

public void save(Product item) {

System.out.println("Saving product: " + item);

}

@Override

public Product find(int id) {

return new Product(id, "Product" + id); // Just a mock product

}

}

class Product {

int id;

String name;

Product(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "Product{id=" + id + ", name='" + name + "'}";

}

}

public class GenericInterfaceExample {

public static void main(String[] args) {

Repository<Product> productRepo = new ProductRepository();

Product product = productRepo.find(1);

System.out.println(product);

productRepo.save(product);

}

}

**Output:**

Product{id=1, name='Product1'}

Saving product: Product{id=1, name='Product1'}

### ****8. Generic Collections****

Java's collection framework is designed with generics. By using generics in collections, you can avoid ClassCastException and ensure type safety.

Example:

import java.util.\*;

public class GenericCollectionExample {

public static void main(String[] args) {

// A List of Integer values

List<Integer> numbers = new ArrayList<>();

numbers.add(10);

numbers.add(20);

numbers.add(30);

// No need for casting when retrieving values

for (Integer number : numbers) {

System.out.println(number);

}

}

}

**Output:**

10

20

30