Java generics provide a way to write flexible and type-safe code by enabling classes, interfaces, and methods to operate on types specified by the user while ensuring compile-time type safety. Here's a detailed tutorial on Java generics, covering in-built and user-defined generics with examples.

**Why Use Generics in Java?**

1. **Type Safety**: Avoids runtime ClassCastException by catching invalid type usage at compile time.
2. **Code Reusability**: Generic code can work with any type.
3. **Improved Performance**: Eliminates the need for type casting.
4. **Readability and Maintainability**: Makes the code easier to understand.

**Syntax of Generics**

* <T>: A placeholder for a type parameter. Common conventions:
  + T: Type
  + E: Element
  + K, V: Key and Value
  + N: Number

**In-Built Generics in Java**

**1. Generic Collections**

The Java Collections Framework uses generics extensively.

**Example: Generic List**

import java.util.ArrayList;

import java.util.List;

public class GenericListExample {

public static void main(String[] args) {

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

stringList.add("Hello");

stringList.add("World");

// stringList.add(10); // Compile-time error

for (String str : stringList) {

System.out.println(str.toUpperCase());

}

}

}

**Example: Generic Map**

import java.util.HashMap;

import java.util.Map;

public class GenericMapExample {

public static void main(String[] args) {

Map<Integer, String> map = new HashMap<>();

map.put(1, "One");

map.put(2, "Two");

for (Map.Entry<Integer, String> entry : map.entrySet()) {

System.out.println(entry.getKey() + " -> " + entry.getValue());

}

}

}

**3. Wildcard Generics**

* <?>: Unknown type
* <? extends T>: Upper-bounded wildcard
* <? super T>: Lower-bounded wildcard

**Example: Unbounded Wildcard**

import java.util.Arrays;

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 = Arrays.asList(1, 2, 3);

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

printList(intList);

printList(strList);

}

}

**User-Defined Generics**

**1. Generic Class**

A class that can operate on multiple types.

public class GenericBox<T> {

private T item;

public void setItem(T item) {

this.item = item;

}

public T getItem() {

return item;

}

public static void main(String[] args) {

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

stringBox.setItem("Hello");

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

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

intBox.setItem(100);

System.out.println("Integer: " + intBox.getItem());

}

}

**2. Generic Method**

A method with its own type parameter.

public class GenericMethodExample {

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

for (T element : array) {

System.out.print(element + " ");

}

System.out.println();

}

public static void main(String[] args) {

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

String[] strArray = {"A", "B", "C"};

printArray(intArray);

printArray(strArray);

}

}

**3. Generic Interface**

interface GenericInterface<T> {

void display(T item);

}

class GenericInterfaceImpl<T> implements GenericInterface<T> {

@Override

public void display(T item) {

System.out.println("Item: " + item);

}

}

public class GenericInterfaceExample {

public static void main(String[] args) {

GenericInterface<String> stringImpl = new GenericInterfaceImpl<>();

stringImpl.display("Hello");

GenericInterface<Integer> intImpl = new GenericInterfaceImpl<>();

intImpl.display(123);

}

}

**4. Bounded Type Parameters**

Restricts the types that can be passed as type arguments.

**Example: Upper Bound**

public class BoundedTypeExample {

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

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

}

public static void main(String[] args) {

printDouble(10); // Integer

printDouble(10.5); // Double

// printDouble("Test"); // Compile-time error

}

}

Wildcard in Java Generics

The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.

We can use a wildcard as a **type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype**.

import java.util.\*;

**abstract** **class** Shape{

**abstract** **void** draw();

}

**class** Rectangle **extends** Shape{

**void** draw(){System.out.println("drawing rectangle");}

}

**class** Circle **extends** Shape{

**void** draw(){System.out.println("drawing circle");}

}

**class** GenericTest{

//creating a method that accepts only child class of Shape

**public** **static** **void** drawShapes(List<? **extends** Shape> lists){

**for**(Shape s:lists){

s.draw();//calling method of Shape class by child class instance

}

}

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

List<Rectangle> list1=**new** ArrayList<Rectangle>();

list1.add(**new** Rectangle());

List<Circle> list2=**new** ArrayList<Circle>();

list2.add(**new** Circle());

list2.add(**new** Circle());

drawShapes(list1);

drawShapes(list2);

}}

import java.util.ArrayList;

**public** **class** UpperBoundWildcard {

**private** **static** Double add(ArrayList<? **extends** Number> num) {

**double** sum=0.0;

**for**(Number n:num)

        {

            sum = sum+n.doubleValue();

        }

**return** sum;

    }

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

        ArrayList<Integer> l1=**new** ArrayList<Integer>();

        l1.add(10);

        l1.add(20);

        System.out.println("displaying the sum= "+add(l1));

       ArrayList<Double> l2=**new** ArrayList<Double>();

        l2.add(30.0);

        l2.add(40.0);

        System.out.println("displaying the sum= "+add(l2));

    }

}

**Unbounded Wildcards**

The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -

* When the given method is implemented by using the functionality provided in the Object class.
* When the generic class contains the methods that don't depend on the type parameter.

Example of Unbounded Wildcards

import java.util.Arrays;

import java.util.List;

public class UnboundedWildcard {

public static void display(List<?> list)

{

for(Object o:list)

{

System.out.println(o);

}

}

public static void main(String[] args) {

List<Integer> l1=Arrays.asList(1,2,3);

System.out.println("displaying the Integer values");

display(l1);

List<String> l2=Arrays.asList("One","Two","Three");

System.out.println("displaying the String values");

display(l2);

}

}

Lower Bounded Wildcards

The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.

Syntax

import java.util.Arrays;

import java.util.List;

public class LowerBoundWildcard {

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

for(Object n:list)

{ System.out.println(n);

}

}

public static void main(String[] args) {

List<Integer> l1=Arrays.asList(1,2,3);

System.out.println("displaying the Integer values");

addNumbers(l1);

List<Number> l2=Arrays.asList(1.0,2.0,3.0);

System.out.println("displaying the Number values");

addNumbers(l2);

}

}

**Key Points**

1. **Type Erasure**: Generics in Java are implemented using type erasure, meaning the generic type information is erased during runtime, and raw types are used instead.
2. **Raw Types**: Using a generic type without specifying the type parameter (e.g., List instead of List<String>) is discouraged but possible.
3. **Restrictions**:
   * Cannot instantiate generic types directly (e.g., new T()).
   * Cannot create arrays of generic types (e.g., T[] array).

Would you like to explore any specific concept in detail or need further examples?

4o

import java.util.Arrays;

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 = Arrays.asList(1, 2, 3);

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

printList(intList);

printList(strList);

}

}

**User-Defined Generics**

**1. Generic Class**

A class that can operate on multiple types.

public class GenericBox<T> {

private T item;

public void setItem(T item) {

this.item = item;

}

public T getItem() {

return item;

}

public static void main(String[] args) {

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

stringBox.setItem("Hello");

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

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

intBox.setItem(100);

System.out.println("Integer: " + intBox.getItem());

}

}

**2. Generic Method**

A method with its own type parameter.

public class GenericMethodExample {

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

for (T element : array) {

System.out.print(element + " ");

}

System.out.println();

}

public static void main(String[] args) {

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

String[] strArray = {"A", "B", "C"};

printArray(intArray);

printArray(strArray);

}

}

GenericInterface:

interface GenericInterface<T> {

void display(T item);

}

class GenericInterfaceImpl<T> implements GenericInterface<T> {

@Override

public void display(T item) {

System.out.println("Item: " + item);

}

}

public class GenericInterfaceExample {

public static void main(String[] args) {

GenericInterface<String> stringImpl = new GenericInterfaceImpl<>();

stringImpl.display("Hello");

GenericInterface<Integer> intImpl = new GenericInterfaceImpl<>();

intImpl.display(123);

}

}

**4. Bounded Type Parameters**

Restricts the types that can be passed as type arguments.

**Example: Upper Bound**

public class BoundedTypeExample {

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

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

}

public static void main(String[] args) {

printDouble(10); // Integer

printDouble(10.5); // Double

// printDouble("Test"); // Compile-time error

}

}

 **Type Erasure**: Generics in Java are implemented using type erasure, meaning the generic type information is erased during runtime, and raw types are used instead.

 **Raw Types**: Using a generic type without specifying the type parameter (e.g., List instead of List<String>) is discouraged but possible.

 **Restrictions**:

* Cannot instantiate generic types directly (e.g., new T()).
* Cannot create arrays of generic types (e.g., T[] array).