



JAVA Programming

Course Instructor: Dr. Suvojit Dhara

School of Computer Science

UPES Dehradun

TOPICS to be discussed

- Concept of Java Generics
- Key Features of Generics
- Different Generics
 - ❑ Generic Method
 - ❑ Generic Class
 - ❑ Generic Interface
- Bounded Type Parameters
- Wild Cards
- Limitations of Generics

Let's START ...!!!



Concept of Generics

- **Generics** in **Java** allow us to write code that can operate on objects of various types while providing compile-time type safety.
- Introduced in **Java-5**, **Generics** enables us to define **methods**, **classes**, and **interfaces** with a placeholder for a type parameter.
- Use of **generics** in **Java** avoids the need for casting and reduces runtime errors.

Key Features of Generics

- ❑ **Type Safety:** **Generics** ensure that the code is type-checked at the compile time
- ❑ **Eliminates Casting:** **Generics** reduces the need for explicit casting in our code.
- ❑ **Code Reusability:** **Generics** allow us to write **generic algorithms** or **classes**, that work with different data types.

Generic Method

- We can write a single **generic method** declaration that can be called with arguments of different data types.
- Based on the types of arguments passed to the **generic method**, the compiler handles each method call appropriately.

```
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, 4};  
        String[] strArray = {"A", "B", "C"};  
        printArray(intArray);  
        printArray(strArray);  
    }  
}
```

Output: 1 2 3 4
 A B C

Generic Class

- A **generic class** declaration looks like a normal/regular **class** declaration, except that the **class** name is followed by a **type parameter** section.
- The **type parameter** is defined within angle brackets <T> and this can be replaced with any specific type when an **object** is created.

```
class Box<T> {  
    private T item;  
    public Box(T t){ item = t; }  
    public T get(){ return item; }  
    public static void main(String[] args) {  
        Box<Integer> intBox = new Box<Integer>(10);  
        Box<Double> dblBox = new Box<Double>(10.73);  
  
        System.out.println(intBox.get());  
        System.out.println(dblBox.get());  
    }  
}
```

Output: 10
10.73

```
class Box<T, U> {  
    private T item1; private U item2;  
    public Box(T t, U u){ item1 = t; item2 = u; }  
    public T getObj1(){ return item1; }  
    public U getObj2(){ return item2; }  
    public static void main(String[] args) {  
        Box<Integer, String> Box = new  
            Box<Integer, String>(10, "Hello");  
  
        System.out.println("Obj1: " + Box.getObj1());  
        System.out.println("Obj2: " + Box.getObj2());  
    }  
}
```

Output: Obj1: 10
Obj2: Hello

Generic Interface

- A **generic interface** in **Java** allows you to define an **interface** with type parameters, making it flexible and reusable for different data types.

```
interface Storage<T> {  
    void addItem(T item);  
    T getItem();  
}  
  
//Implement the generic interface  
class Box<T> implements Storage<T> {  
    private T item;  
  
    @Override  
    public void addItem(T item){  
        this.item = item;  
    }  
  
    @Override  
    public T getItem(){  
        return item;  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Storage<String> strBox = new Box<>();  
        strBox.addItem("Hello, Generics!");  
        System.out.println("String in box: " +  
                             strBox.getItem());  
  
        Storage<Integer> intBox = new Box<>();  
        intBox.addItem(123);  
        System.out.println("Integer in box: " +  
                             intBox.getItem());  
    }  
}
```

Output:

String in box: Hello, Generics!

Integer in box: 123

Bounded Type Parameters

- There may be times when we want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates only on Numbers might expect **instances** of the **Number class** or its **subclasses** as parameters.
- A **bounded type parameter** in **Java Generics** allows you to restrict the type parameters to a specific range of types, ensuring that the parameterized type fulfils certain requirements.
- This is achieved using the “**extends**” keyword for both **classes** and **interfaces**.

Syntax:

`<T extends ClassName>`

`<T extends InterfaceName>`

Upper Bound (Example)

```
class Box<T extends Number> {  
    private T value;  
    public Box(T value){ this.value = value; }  
    public double square(){  
        return value.doubleValue()*value.doubleValue();  
    }  
    public T getValue(){ return value; }  
    public void setValue(T value){ this.value = value; }  
  
    public static void main(String[] args) {  
        //Create a Box for Integer  
        Box<Integer> intBox = new Box<>(5);  
        System.out.println("Integer square: " + intBox.square());  
  
        //Create a Box for Double  
        Box<Double> doubleBox = new Box<>(7.5);  
        System.out.println("Double square: " + doubleBox.square());  
  
        //Create a Box for Float  
        Box<Float> floatBox = new Box<>(3.5f);  
        System.out.println("Float square: " + floatBox.square());  
    }  
}
```

Output:

Integer square: 25.0

Double square: 56.25

Float square: 12.25

Multiple Bounds (Example)

```
class Box<T extends Number & Comparable<T>> {  
    private T value;  
    public Box(T value){    this.value = value;    }  
    public T getValue(){    return value;    }  
    public void setValue(T value){    this.value = value;    }  
  
    //A method that compares the value with another Box's value  
    public boolean isGreaterThan(Box<T> otherBox){  
        return this.value.compareTo(otherBox.getValue()) > 0;  
    }  
  
    public static void main(String[] args) {  
        //Create Box instances with Integer values  
        Box<Integer> box1 = new Box<>(10);  
        Box<Integer> box2 = new Box<>(5);  
        System.out.println("Box1 is greater than Box2: " +  
                            box1.isGreaterThan(box2));  
  
        //Create Box instances with Double values  
        Box<Double> box3 = new Box<>(15.5);  
        Box<Double> box4 = new Box<>(20.0);  
        System.out.println("Box3 is greater than Box4: " +  
                            box3.isGreaterThan(box4));  
    }  
}
```

Output:

Box1 is greater than Box2: true
Box3 is greater than Box4: false

Wild Cards

- Wildcards in **Java Generics** are represented by the symbol `?` and are used to represent an unknown type.
- They make **generics** more flexible by allowing **methods**, **classes**, or **interfaces** to work with a variety of data types without knowing their exact types at compile time.
- Wildcards are especially useful when working with **collections** and **APIs** where type flexibility is required.
- Wildcards can be of following types:
 - ❑ **Unbounded Wildcard** (`<?>`)
 - ❑ **Upper-bounded Wildcard** (`<? extends Type>`)
 - ❑ **Lower-bounded Wildcard** (`<? super Type>`)

Unbounded Wild Cards (<?>)

- The **unbounded wildcard** (<?>) represents any type. It is used when the exact type is unknown or irrelevant.
- **Unbounded wildcards** are typically used when you want to read or operate on a collection, but do not care about its specific type.

Syntax:

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

```
import java.util.*;
class UnboundedWildcardExample {
    public static void printList(List<?> list){
        for(Object obj : list){
            System.out.print(obj + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        List<?> List1 =
            Arrays.asList("A", "B", "C", 1, 2);
        List<?> List2 =
            Arrays.asList(1, 2, 3, "A", "B");

        printList(List1);
        printList(List2);
    }
}
```

Output: A B C 1 2
 1 2 3 A B

Upper-bounded Wild Cards

<? extends Type>

- The **upper-bounded wildcard** restricts the type to be **Type** or any **subclass** of **Type**.
- It is useful when you want to read data from a **collection**, but you are unsure of the exact type.

Syntax:

List<? extends Number> list;

```
import java.util.*;
class Main {
    public static double sum(List<? extends Number> list){
        double sum = 0.0;
        for(Number num : list){ sum += num.doubleValue(); }
        return sum;
    }
    public static void main(String[] args) {
        List<? extends Number> List1 =
            Arrays.asList(1, 2, 3, 4.5, 9.6);
        List<? extends Number> List2 =
            Arrays.asList(1.1, 2.2, 3.3, 5, 7, 8);

        System.out.println("List1 Sum: "+sum(List1));
        System.out.println("List2 Sum: "+sum(List2));
    }
}
```

Output: List1 Sum: 20.1
List2 Sum: 26.6

Lower-bounded Wild Cards

<? super Type>

- The **lower-bounded wildcard** restricts the type to be **Type** or any **superclass** of **Type**.
- It is useful when you want to write data to a collection but do not care about its exact type.

Syntax:

List<? super Integer> list;

```
import java.util.*;
class LowerboundedWildcardExample {
    public static void addNums(List<? super Integer> list){
        list.add(10);
        list.add(20);
        list.add(30);
    }
    public static void main(String[] args) {
        List<Number> numberList = new ArrayList<>();
        addNums(numberList);

        System.out.println("The list is: "+numberList);
    }
}
```

Output:

The list is: [10, 20, 30]

Key Points to Remember

- **Unbounded Wildcards (?):**

Allow any type but are read-only.

- **Upper-Bounded Wildcards (<? extends Type>):**

Allow reading elements as a specific type but restrict writing.

- **Lower-Bounded Wildcards (<? super Type>):**

Allow writing elements but restrict reading as Object.

Limitations of Generics

- **Generics** in **Java** work only with reference types (**objects**), not primitive types. For example, you cannot directly use `int`, `double`, or `char` with generics.

```
List<int> intList = new ArrayList<>();    //Compile-time error
List<Integer> intList = new ArrayList<>(); //Correct
```

- **Generics** in **Java** use **type erasure**, meaning that all type information is removed at run-time and replaced with **Object** (or the specified bound). As a result, the runtime does not retain the **generic type** information.

```
List<Integer> intList = new ArrayList<>();
List<String> strList = new ArrayList<>();
System.out.println(intList.getClass() == strList.getClass()); //Output: true
System.out.println(intList.getClass().getName()); //Output: java.util.ArrayList
System.out.println(strList.getClass().getName()); //Output: java.util.ArrayList
```


Limitations of Generics

- You cannot create **arrays** of **generic types** directly because of **type erasure**. The **Java** runtime cannot ensure type safety for such arrays.

```
List<String>[] array = new ArrayList<String>[10];    //Compile-time error  
List<List<String>> listOfLists = new ArrayList<>(); //Correct
```

- **Generics** cannot be used in **static fields** or **methods** because **generics** are resolved at the **class level**, not the **instance level**. **Static** members belong to the **class** and are shared across all **instances**, so they cannot depend on instance-specific type parameters.

```
public class GenericClass<T> {  
    private static T staticField;    //Compile-time error  
  
    public static T getStaticField(){ //Compile-time error  
        return staticField;  
    }  
}
```

Summary

Today, we learned about

- Concept of Generics and Their key features
- Generic methods, classes, and interfaces
- Bounded type parameters (upper-bound, multiple bounds)
- Wild Cards (Unbounded, upper-bounded and lower-bounded wild cards)
- Limitations of Generics



Thank You!