Generics

Wildcards

The question mark (?) is known as the wildcard in generic programming . It represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type.

Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

Generics

Types of wildcards in Java:

1- **Upper Bounded Wildcards**: These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < integer >, List < double >, and List < number > , you can do this using an upper bounded wildcard.

To declare an upper-bounded wildcard, use the wildcard character (‘?’), followed by the extends keyword, followed by its upper bound.

```
public static void add(List<? extends Number> list)
```

The following Java program to demonstrate Upper Bounded Wildcards;

Generics

```
import java.util.Arrays;
import java.util.List;
class WildcardDemo{
    public static void main(String[] args){
        List<Integer> list1= Arrays.asList(4,5,6,7);
        System.out.println("Total sum is:"+sum(list1));
        List<Double> list2=Arrays.asList(4.1,5.1,6.1);
        System.out.println("Total sum is:"+sum(list2));
    }
    private static double sum(List<? extends Number>
list)     {
        double sum=0.0;
        for (Number i: list){sum+=i.doubleValue();}
        return sum;
    } }
```

Generics

In the above program, list1 and list2 are objects of the List class. list1 is a collection of Integer and list2 is a collection of Double.

Both of them are being passed to method sum which has a wildcard that extends Number. This means that list being passed can be of any field or subclass of that field. Here, Integer and Double are subclasses of class Number.

2- **Lower Bounded Wildcards**: It is expressed using the wildcard character ('?'), followed by the super keyword, followed by its lower bound.

```
Collectiontype <? super A>
```

The following java program to demonstrate Lower Bounded Wildcards;

Generics

```
import java.util.Arrays;
import java.util.List;
class WildcardDemo{
    public static void main(String[] args){
        //Lower Bounded Integer List
        List<Integer> list1= Arrays.asList(4,5,6,7);
        printIntegerClassorSuperClass(list1);
        List<Number> list2= Arrays.asList(4,5,6,7);
        printIntegerClassorSuperClass(list2);
    }
    static void printIntegerClassorSuperClass(List<?
super Integer> list){
        System.out.println(list);
    }
}
```

Generics

Here arguments can be Integer or superclass of Integer(which is Number). The method printIntegerClassorSuperClass will only take Integer or its superclass objects.

However if we pass list of type Double then we will get compilation error. It is because only the Integer field or its superclass can be passed . Double is not the superclass of Integer.

3- **Unbounded Wildcard**: This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown type. These are useful in the following cases. When writing a method which can be employed using functionality provided in Object class. When the code is using methods in the generic class that don't depend on the type parameter.

Generics

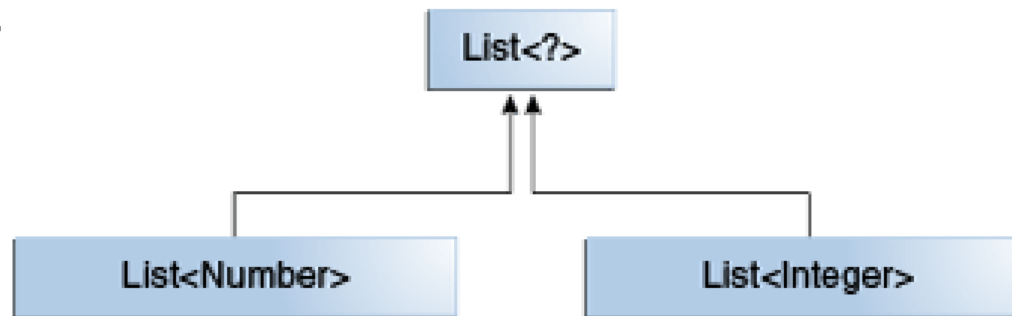
```
import java.util.Arrays;
import java.util.List;
class unboundedwildcardemo{
    public static void main(String[] args) {
        List<Integer> list1= Arrays.asList(1,2,3);
        List<Double> list2=Arrays.asList(1.1,2.2,3.3);
        printlist(list1);
        printlist(list2);
    }
    private static void printlist(List<?> list)
    {
        System.out.println(list);
    }
}
```

Generics

Consider the following method, printList:

```
private static void printlist(List<Object> list)
{
        System.out.println(list);
}
```

The goal of printList is to print a list of any type, but it fails to achieve that goal — it prints only a list of Object instances; it cannot print List<Integer>, List<String>, List<Double>, and so on, because they are not subtypes of List<Object>.



Generics

Type Erasure

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming.

To implement generics, the Java compiler applies type erasure to:

1. Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
2. Insert type casts if necessary to preserve type safety.
3. Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure, ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

Generics

During the type erasure process, the Java compiler erases all type parameters and replaces each with its first bound if the type parameter is bounded, or Object if the type parameter is unbounded.

Consider the following generic class that represents a node in a singly linked list:

```
public class Node<T> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
}
```

Generics

Because the type parameter T is unbounded, the Java compiler replaces it with Object:

```
public class Node {
    private Object data;
    private Node next;
    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Object getData() { return data; }
}
```

Generics

In the following example, the generic Node class uses a bounded type parameter:

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
}
```

Generics

The Java compiler replaces the bounded type parameter T with the first bound class, Comparable:

```
public class Node {
    private Comparable data;
    private Node next;
    public Node(Comparable data, Node next) {
        this.data = data;
        this.next = next;
    }
    public Comparable getData() { return data; }
}
```

Generics and Inheritance

- A generic class can extend a non-generic class.

```
class NonGenericClass
{
     //Non Generic Class
}

class GenericClass<T> extends NonGenericClass
{
    //Generic class extending non-generic class
}
```

Generics and Inheritance

- Generic class can also extend another generic class. When generic class extends another generic class, sub class should have at least same type and same number of type parameters and at most can have any number and any type of parameters.

```
class GenericSuperClass<T>
{
    //Generic super class with one type parameter
}

class GenericSubClass1<T> extends GenericSuperClass<T>
{
    //sub class with same type parameter
}
```

Generics and Inheritance

```
class GenericSubClass2<T, V> extends
GenericSuperClass<T>
{
    //sub class with two type parameters
}

class GenericSubClass3<T1, T2> extends
GenericSuperClass<T>
{
    /*Compile time error, sub class having different
type of parameters*/
}
```

Generics and Inheritance

- Non-generic class can’t extend generic class except of those generic classes which have already pre defined types as their type parameters.

```
class GenericSuperClass<T>
{
    //Generic class with one type parameter
}

class NonGenericClass extends GenericSuperClass<T>
{
    /*Compile time error, non-generic class can't
extend generic class*/
}
```

Generics and Inheritance

```
class GenericSuperClass1<A>
{
    /*Generic class with pre defined type 'A' as type
parameter*/
}

class NonGenericClass1 extends GenericSuperClass1<A>
{
    //No compile time error, It is legal
}
```

Generics and Inheritance

- Non-generic class can extend generic class by removing the type parameters. i.e as a raw type. But, it gives a warning.

```
class GenericClass<T>
{
    T t;
     public GenericClass(T t)    {        this.t = t;
}
}
 class NonGenericClass extends GenericClass  //Warning
{
    public NonGenericClass(String s) {
       super(s);           //Warning    }
}
```

Generics and Inheritance

- While extending a generic class having bounded type parameter, type parameter must be replaced by either upper bound or it's sub classes.

```
class GenericSuperClass<T extends Number>
{
        //Generic super class with bounded type
parameter
}

class GenericSubClass1 extends
GenericSuperClass<Number>
{
        //type parameter replaced by upper bound
}
```

Generics and Inheritance

- While extending a generic class having bounded type parameter, type parameter must be replaced by either upper bound or it's sub classes.

```
class GenericSuperClass<T extends Number>
{
    //Generic super class with bounded type parameter
}
class GenericSubClass2 extends GenericSuperClass<Integer>
{
    //type parameter replaced by sub class of upper bound
}
class GenericSubClass3 extends GenericSuperClass<T extends
Number>
{
    //Compile time error }
```

Guidelines for Wildcard Use

One of the more confusing aspects when learning to program with generics is determining when to use an upper bounded wildcard and when to use a lower bounded wildcard.

It is helpful to think of variables as providing one of two functions:

An "In" Variable

An "in" variable serves up data to the code. Imagine a copy method with two arguments: `copy(src, dest)`. The `src` argument provides the data to be copied, so it is the "in" parameter.

An "Out" Variable

An "out" variable holds data for use elsewhere. In the copy example, `copy(src, dest)`, the `dest` argument accepts data, so it is the "out" parameter.

Guidelines for Wildcard Use

Wildcard Guidelines:

- An "in" variable is defined with an upper bounded wildcard, using the extends keyword.
- An "out" variable is defined with a lower bounded wildcard, using the super keyword.
- In the case where the "in" variable can be accessed using methods defined in the Object class, use an unbounded wildcard.
- In the case where the code needs to access the variable as both an "in" and an "out" variable, do not use a wildcard.
- These guidelines do not apply to a method's return type. Using a wildcard as a return type should be avoided because it forces programmers using the code to deal with wildcards.