Genrics

Generics means parameterized types. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

For Example:

```
class gen<g>
{
g obj;
gen(g obj)
{
    this.obj=obj;
}
public g print()
{
return this.obj;
}
}
class hut
{
public static void main(String[] args)
    {
gen<Integer> k=new gen<Integer>(45);
System.out.println(k.print());

gen<String> l=new gen<String>("Advance java");
System.out.println(l.print());
    }
}
```

Types of Java Generics

Generic Classes: A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

Note (We can also pass multiple Type parameters in Generic classes.)

For Example:

```
class gen<g,h>
{
g obj1;
h obj2;
gen(g obj1, h obj2)
{
    this.obj1=obj1;
    this.obj2=obj2;
}
public void print()
{
System.out.print(obj1+"  ");
System.out.println(obj2);

}
}
class hut
{
public static void main(String[] args)
    {
gen<Integer, String> k=new gen<Integer, String>(45,"Advance java");

k.print();
gen<String, Integer> l=new gen<String, Integer>("Advance java", 56);
l.print();
    }
}
```

Generic Functions: We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to the generic method. The compiler handles each method.

For Example:

```
class fun
{
static<f> void display(f value)
    {
```

```
System.out.println(value.getClass().getName() + " = " +value);
    }
public static void main(String[] args)
    {
display(45);

display("Advance java");
display(65.4);
display('J');
display(1234567891.56);
}
}
```

Wildcard 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.

For Example:

```
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 game
{
public static void drawshape(List<? extends shape> l)
    {
for(shape s:l)
        {
s.draw();
            }
    }
public static void main(String[] args)
{
List<rectangle> l1=new ArrayList<rectangle>();
l1.add(new rectangle());

List<circle> l2=new ArrayList<circle>();
l2.add(new circle());
l2.add(new circle());

drawshape(l1);
drawshape(l2);
}
}
```

Upper Bounded Wildcards: The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.

For Example:

```
import java.util.ArrayList;

public class upcl {
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 Second time 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.

For Example:

```
import java.util.Arrays;
import java.util.List;
public class upcl2
{
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

List<? super Integer>

- **? is a wildcard character.**
- **super, is a keyword.**
- **Integer, is a wrapper class.**

Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using List<? super Integer> is suitable for a list of type Integer or any of its superclasses whereas List<Integer> works with the list of type Integer only. So, List<? super Integer> is less restrictive than List<Integer>.

For Example:

```
import java.util.Arrays;
import java.util.List;
public class lps {
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);
}
}
```