Generics in JAVA

Dr. Mainak Adhikari

Assistant Professor,
Department of Computer Science,
IIIT Lucknow, Lucknow, India

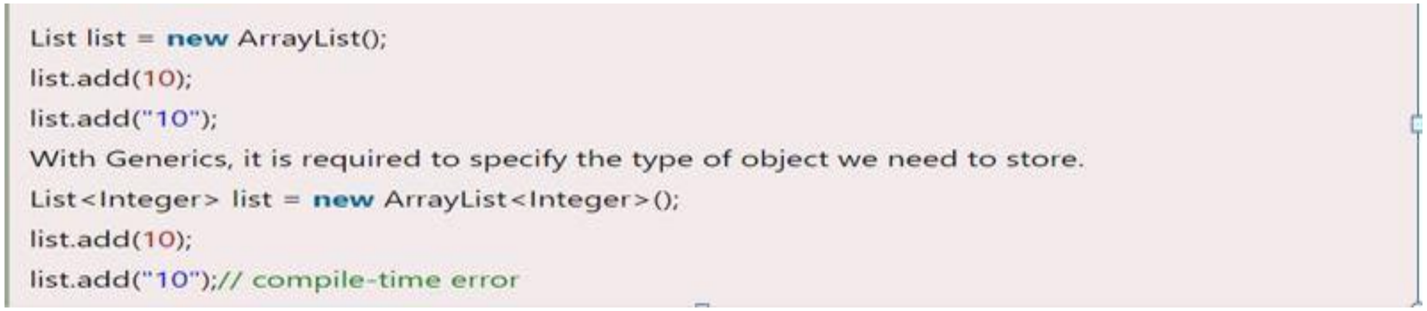
Generics in Java

- The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.
- Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety: We can hold only a single type of objects in generics. It doesn't allow to store other objects.



```
List list = new ArrayList();
list.add(10);
list.add("10");
With Generics, it is required to specify the type of object we need to store.
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

Advantage of Java Generics

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
After Generics, we don't need to typecast the object.
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

Syntax to use generic collection

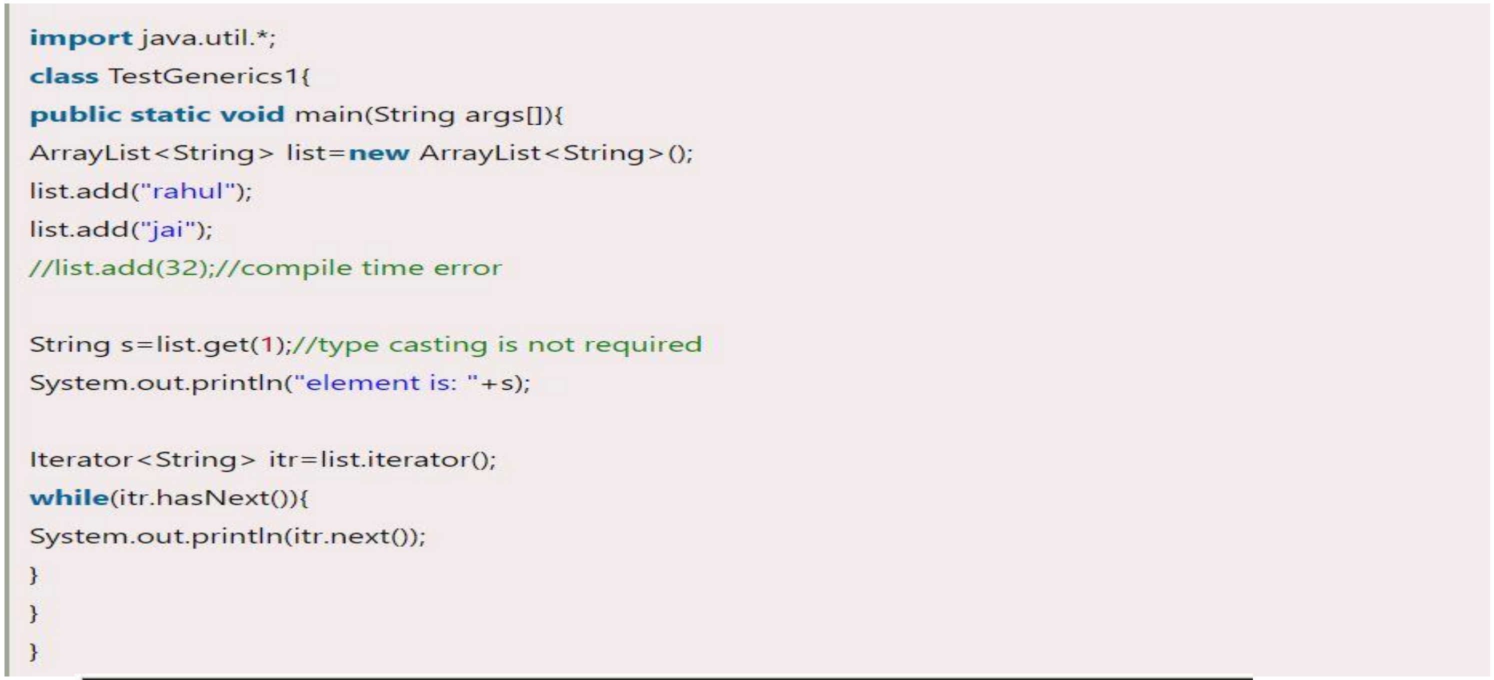
```
ClassOrInterface<Type>
```

Example to use Generics in java

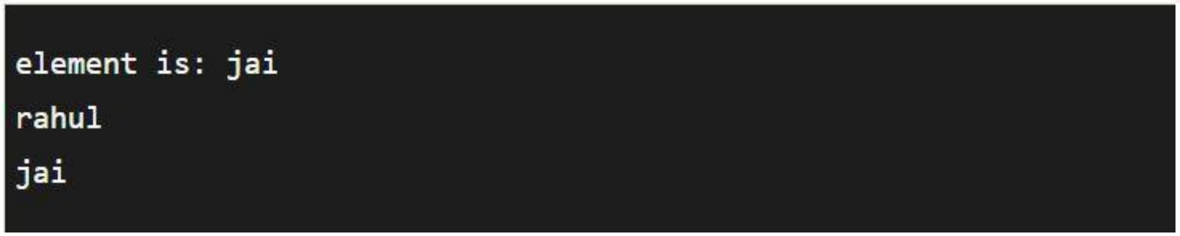
```
ArrayList<String>
```

Example of Generics in Java

Here, we are using the ArrayList class, but you can use any collection class such as ArrayList, LinkedList, HashSet, TreeSet, HashMap, Comparator etc.



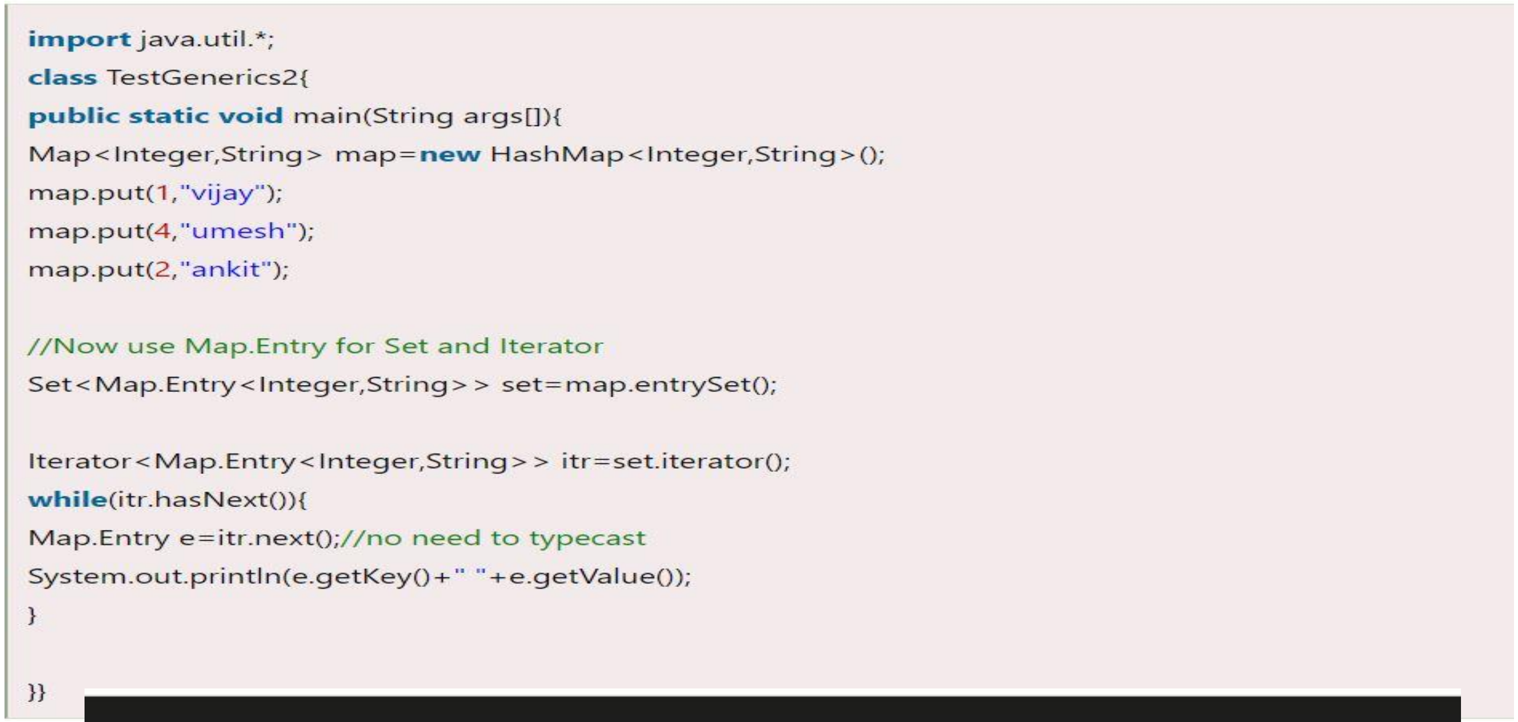
```java
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error

String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```



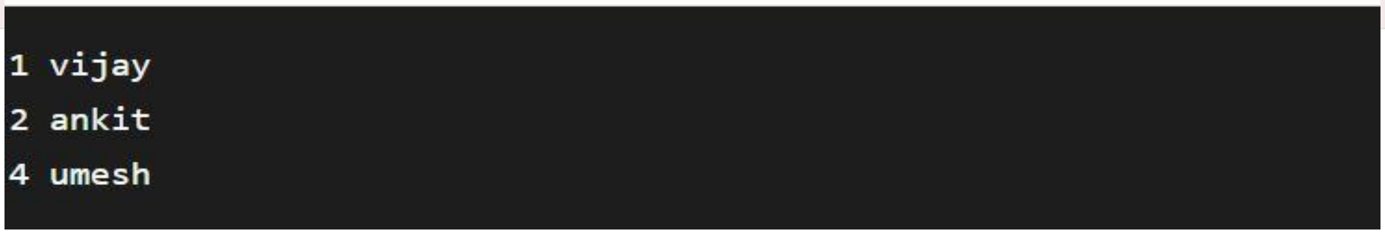
```
element is: jai
rahul
jai
```

Example of Java Generics using Map

- Now we are going to use map elements using generics. Here, we need to pass key and value.



```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");

//Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}

}}
```

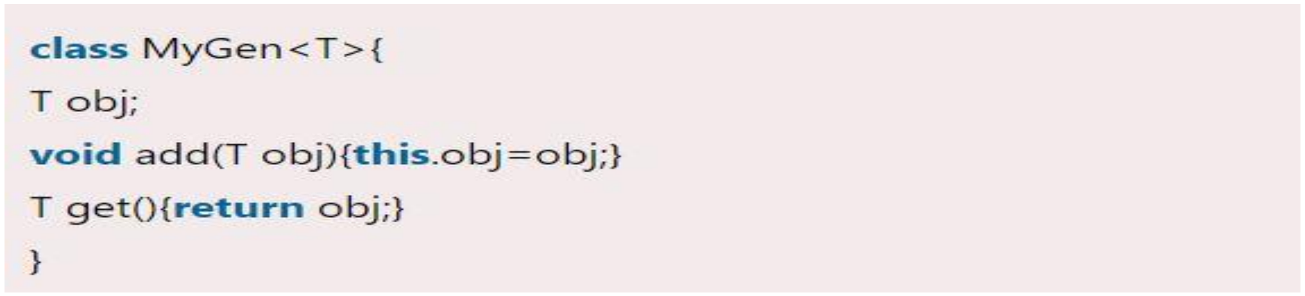


```
1 vijay
2 ankit
4 umesh
```

Generic class

- A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.
- Let's see a simple example to create and use the generic class.

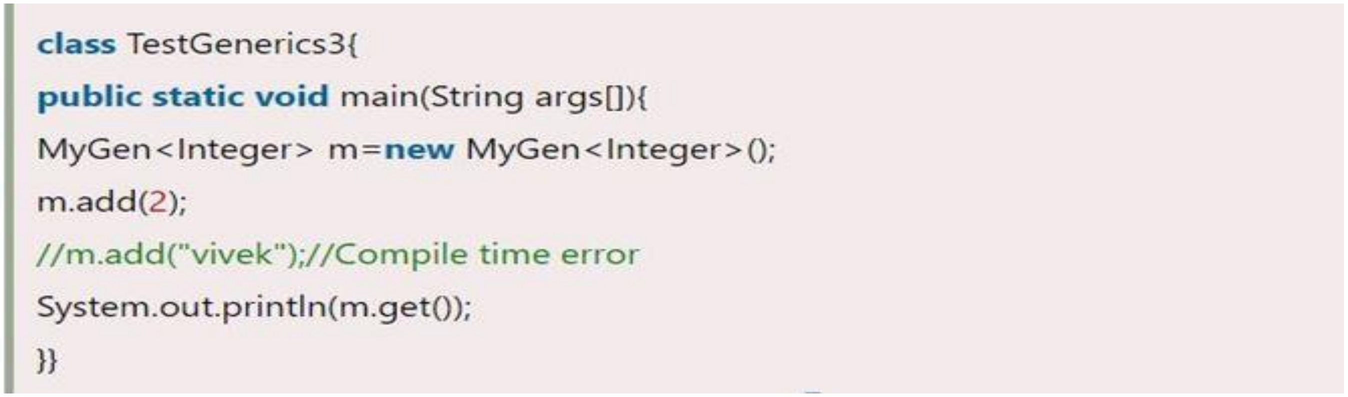
Creating a generic class:



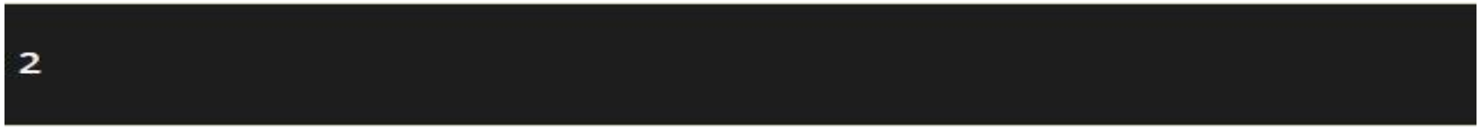
```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

Using generic class



```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```



```
2
```

Type Parameters

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

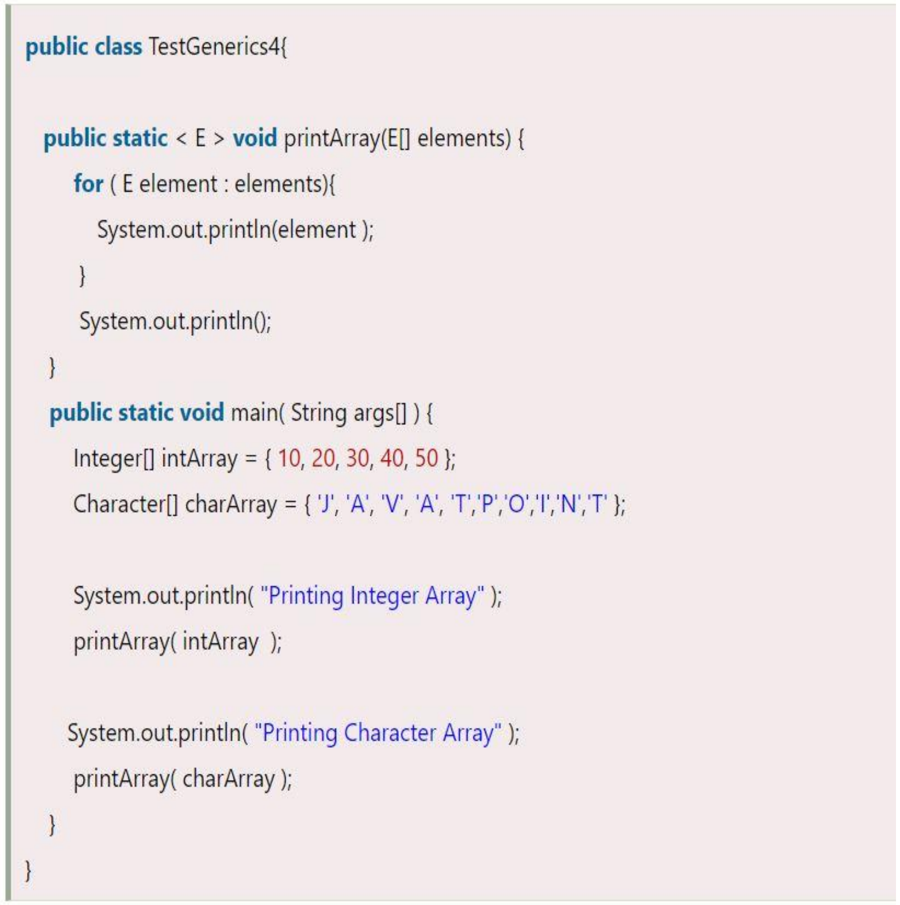
- T - Type
- E - Element
- K - Key
- N - Number
- V - Value

Generic Method

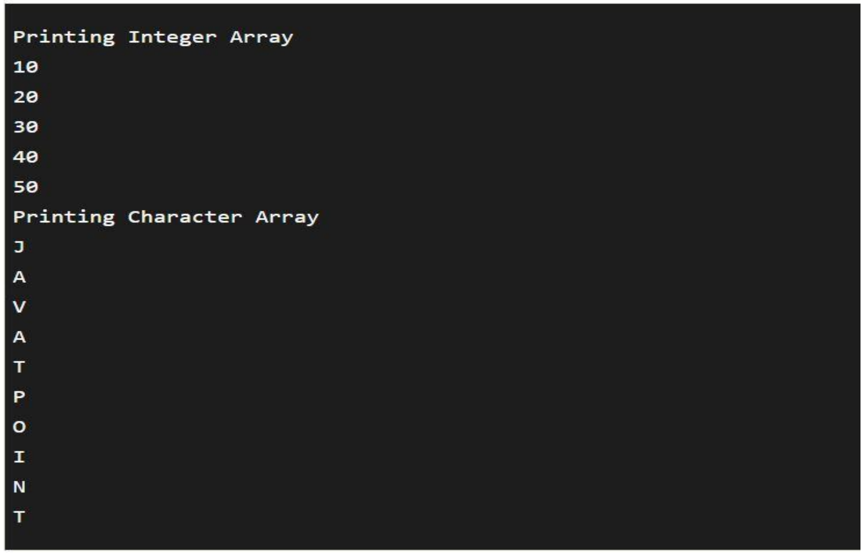
Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

Simple example of java generic method

Example of java generic method to print array elements. We are using here **E** to denote the element.



```
public class TestGenerics4{

  public static < E > void printArray(E[] elements) {
      for ( E element : elements){
         System.out.println(element );
      }
      System.out.println();
  }
  public static void main( String args[] ) {
      Integer[] intArray = { 10, 20, 30, 40, 50 };
      Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

      System.out.println( "Printing Integer Array" );
      printArray( intArray  );

      System.out.println( "Printing Character Array" );
      printArray( charArray );
  }
}
```

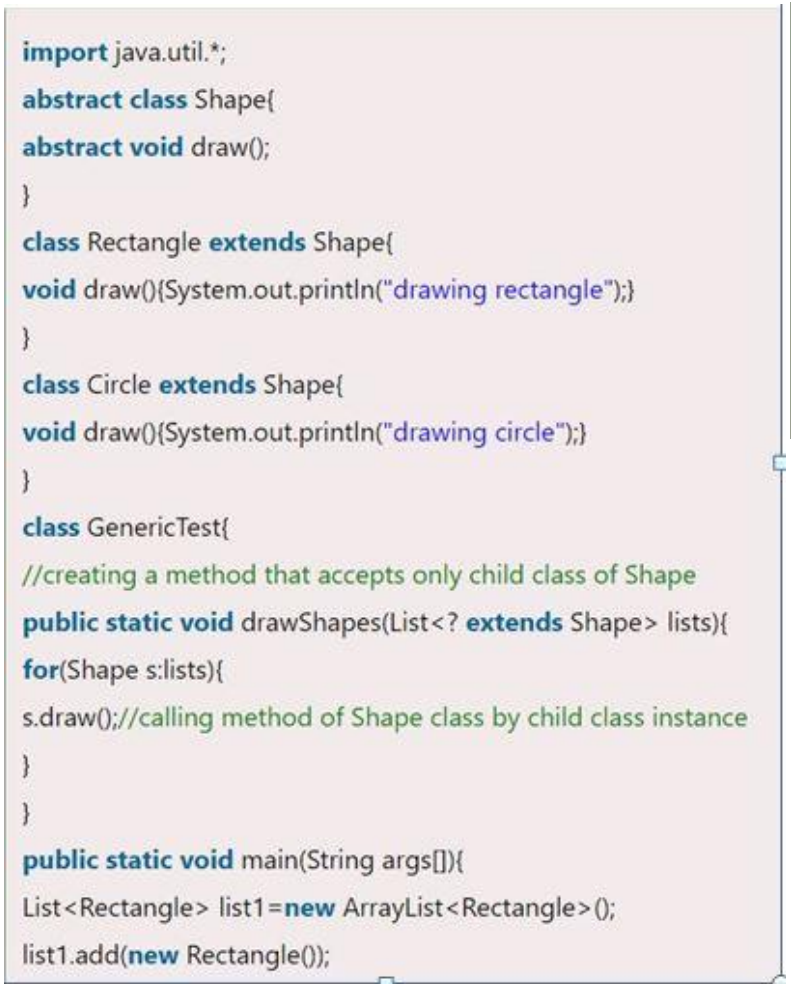


```
Printing Integer Array
10
20
30
40
50
Printing Character Array
J
A
V
A
T
P
O
I
N
T
```

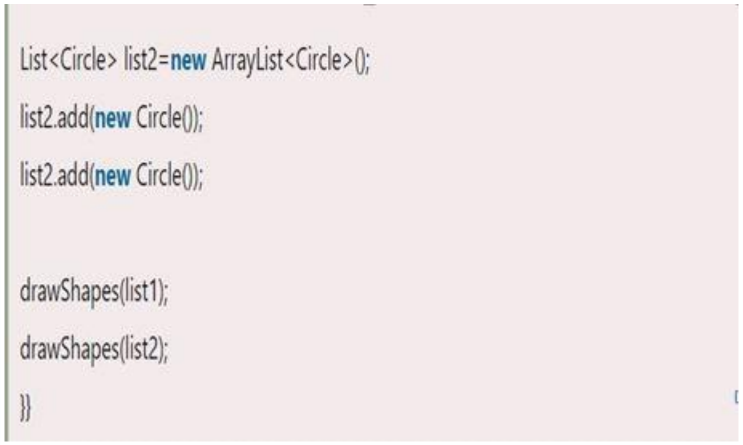
Wildcard in Java Generics

- 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 super type.

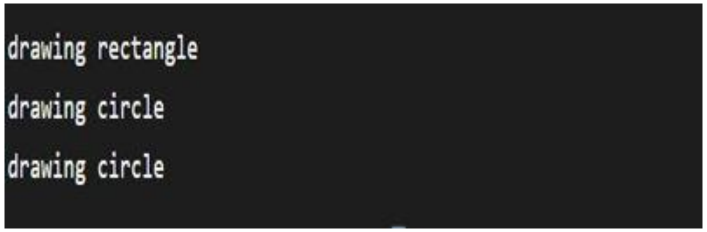
Example of wildcard in Java Generics



```
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);
}}
```



```
drawing rectangle
drawing circle
drawing circle
```

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.

Syntax

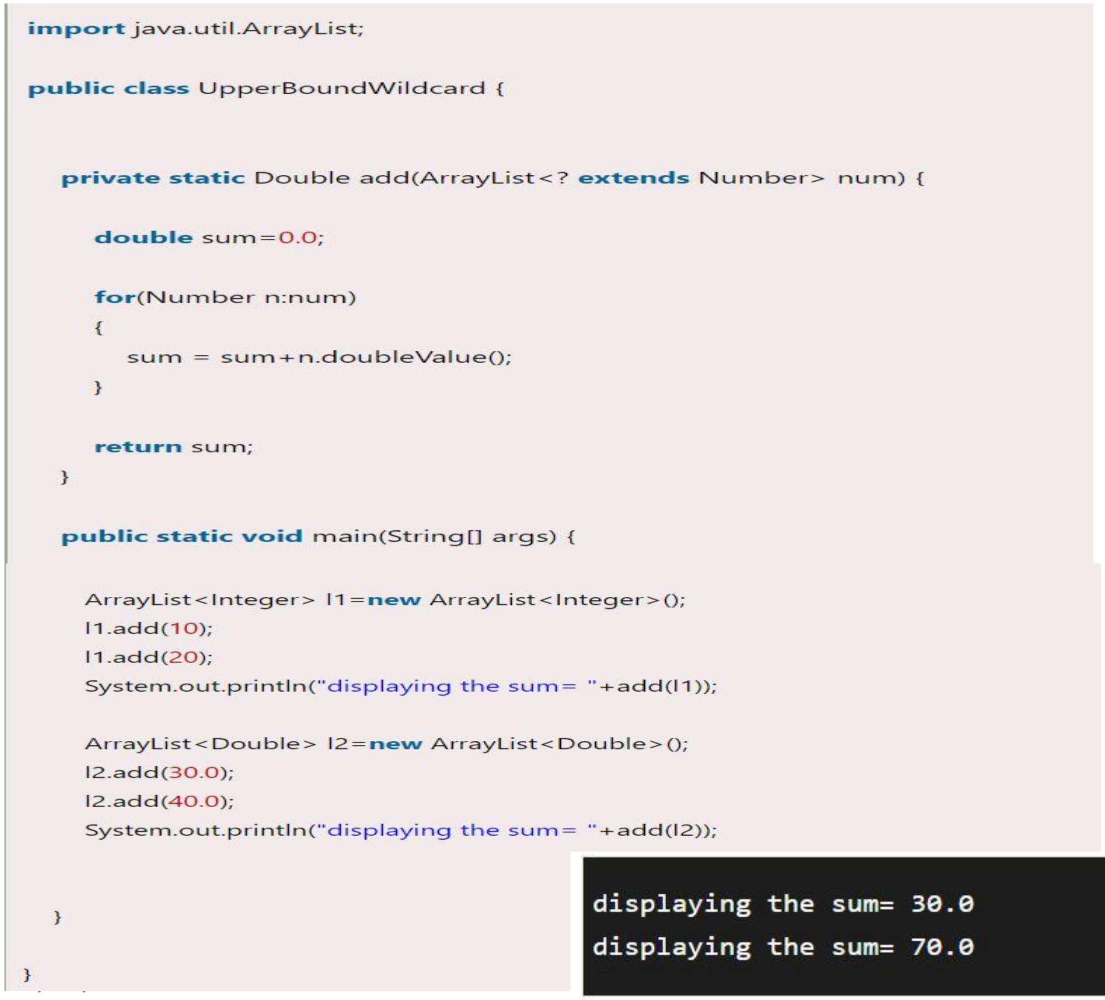
List<? **extends** Number>

- **?** is a wildcard character.
- **extends**, is a keyword.
- **Number**, is a class present in java.lang package

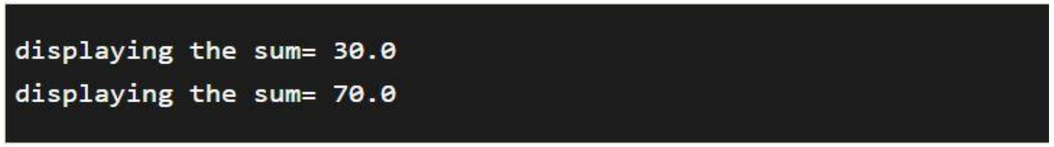
Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using **List<? extends Number>** is suitable for a list of type Number or any of its subclasses whereas **List<Number>** works with the list of type Number only. So, **List<? extends Number>** is less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

Example of Upper bound Wildcards

- In this example, we are using the upper bound wildcards to write the method for List<Integer> and List<Double>



```java
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));


    }

}
```



```
displaying the sum= 30.0
displaying the sum= 70.0
```

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

```java
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);
   }

}
```

Output

```
displaying the Integer values
1
2
3
displaying the String values
One
Two
Three
```

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

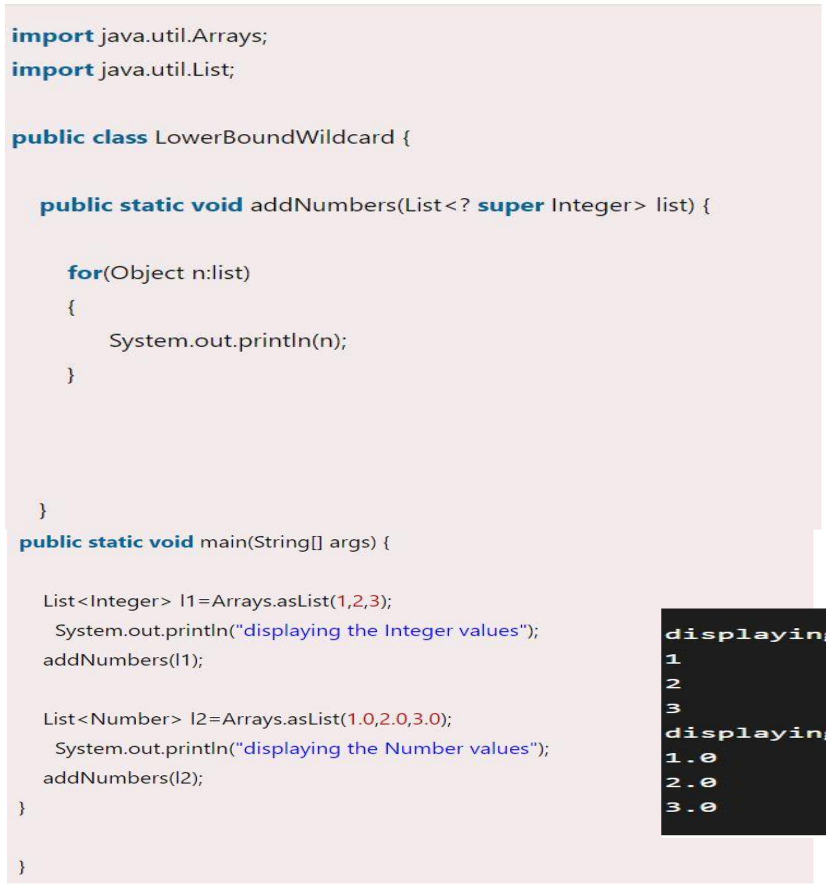
List<? **super** Integer>

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

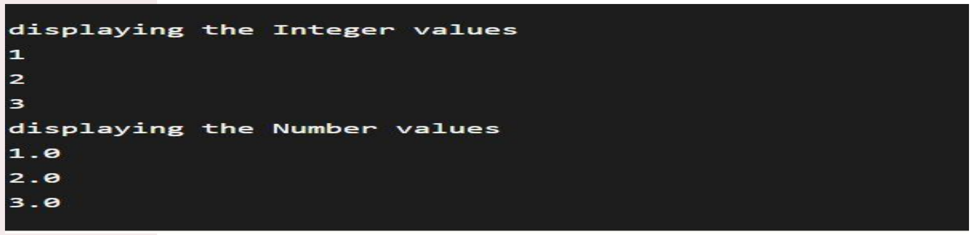
Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses whereas **List<Integer>** works with the list of type Integer only. So, **List<? super Integer>** is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

Example of Lower bound Wildcards

- In this example, we are using the lower bound wildcards to write the method for List<Integer> and List<Number>



```java
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);
  }

  }
```



```
displaying the Integer values
1
2
3
displaying the Number values
1.0
2.0
3.0
```

Thanks