Unit 4

1] Generics / Generic Methods / Bounded Type Parameters / Generic Classes

It would be nice if we could write a single sort method that could sort the elements in an integer array, a String array or an array of any type that supports ordering.

Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a

set of related methods or with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods -

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Example

Following example illustrates how we can print an array of different type using a single Generic method:-

```
public class GenericMethodTest {
  public static <E> void printArray(E[]
  inputArray) {
    for(E element : inputArray){
      System.out.printf("%s", element);
    }
    System.out.println();
  }
  public static void main(String args[]) {
    Integer[] intArray = {1, 2, 3, 4, 5};
    Double[] doubleArray = {1.1, 2.2, 3.3, 4.4};
    Character[] charArray = {'H', 'E', 'L', 'L',
    'O'};
    System.out.println("Array integerArray
    contains:");
    printArray(intArray);
    System.out.println("\n Array doubleArray
```

```
contains: ");
printArray(doubleArray);
System.out.println("\n Array characterA-
rray contains: ");
printArray(charArray);
}
```

Output

```
Array integerArray contains:
1 2 3 4 5
Array doubleArray contains:
1.1 2.2 3.3 4.4
Array characterArray contains:
H E L L O
```

Bounded Type Parameters

There may be times when you'll want to restrict the kind of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or

"implements" (as in interfaces). This example is Generic method to return the largest of three comparable objects -

```
public class MaximumTest {
public static <T extends Comparable
<T>> T maximum(T x, T y, T z){
  T max = x;
if(y.compareTo(max)>0){
    max=y;
  }
if(z.compareTo(max)>0){
  max=z;
  }
  return max;
 }
public static void main(String args[]){
  System.out.println("Max of %d, %d
and %d is %d\n\n", 3, 4, 5, maximum
(3, 4, 5));
System.out.println("Max of %.1f, %.1f
and %.1f is %.1f\n\n", 6.6, 8.8,
7.7, maximum(6.6, 8.8, 7.7));
System.out.println("Max of %s, %s
and %s is: %s\n", "pear", "apple",
"orange", maximum("pear", "apple",
"orange"));
    }
  }
```

Output

Max of 3, 4 and 5 is 5

Max of 6.6, 8.8 and 7.7 is 8.8
Max of pear, apple and orange is pear

Generic Classes

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.
As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example

```
public class Box<T>{
  private T t;
  public void add(T t){
    this.t = t;
  }
  public T get(){
    return t;
  }
public static void main(String[] args)
{
Box<Integer> integerBox = new Box
<Integer>();
Box<String> stringBox = new Box<String
>();
integerBox.add(new Integer(10));
stringBox.add(new String("Hello World"));
```

```
System.out.println("Integer Value :%d\n
\n",integerBox.get());
System.out.println("String Value :%s\n",
stringBox.get());
    }
}
```

Output

```
Integer Value :10
String Value : Hello World
```

Q) Lambda Expressions

Lambda expressions are introduced in Java 8 and are touted to be the biggest feature of Java 8. Lambda expression facilitates functional programming and simplifies the development a lot.

Syntax

A lambda expression is characterized by the following syntax.

parameter → expression body

Following are the important characteristics of a lambda expression.

- Optional type declaration - No need to declare the type of a parameter. The compiler can inference the same from the value of the parameter.
- Optional parenthesis around parameter - No need to declare a single parameter in parenthesis. For multiple parameters,

parenthesis are required.
-Optional curly braces -No need to use curly braces in expression body if the body contains a single statement.
- Optional return Keyword - The compiler automatically returns the value if the body has a single expression to return the value. Curly braces are required to indicate that expression returns a value.

Lambda Expressions Example

Java8Tester.java

```
public class Java8Tester{
  public static void main(String args[]){
    Java8Tester tester=new Java8Tester();

//with type declaration
MathOperation addition=(int a,int b)->
  a+b;
//without type declaration
MathOperation subtraction=(a,b)->a-b;
//with return statement along with
 curly braces
MathOperation multiplication=(int a,int b)
->{return a*b;};
//without return statement and without
 curly braces
MathOperation division=(int a,int b)->
  a/b;

System.out.println("10+5="+tester.opera
```

te (10,5, addition));
System.out.println("10-5= "+tester.operate (10,5, subtraction));
System.out.println("10x5 = "+tester.operate (10,5, multiplication));
System.out.println("10/5 = "+tester.operate (10,5, division));

//without parenthesis
GreetingService greetService1 = message ->
System.out.println("Hello" + message);

// with parenthesis
GreetingService greetService2 = (message) ->
System.out.println("Hello" + message);

greetService1.sayMessage("Mahesh");
greetService2.sayMessage("Suresh");
}

interface MathOperation {
int operation(int a, int b);
}
interface GreetingService {
void sayMessage(String message);
}
private int operate(int a, int b, MathOperation mathOperation) {
return mathOperation.operation
(a,b);
}
}

PAGE No.
DATE / /

Output:-

10+5=15
10-5=5
10x5=2
Hello Mukesh
Hello Suresh

Following are the important points to be considered in the above example.

- lambda expressions are used primarily to define inline implementation of a functional interface i.e an interface with a single method only. In the above example, we've used various types of lambda expressions to define the operation method of MathOperation interface. Then we have defined the implementation of sayMessage of GreetingService.
- lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

Scope

Using lambda expression, you can refer to any final variable or effectively final variable (which is assigned only once). lambda expression throws a compilation error, if a variable is assigned a value the second time.

PAGE No.
DATE / /

Scope Example

- Java8Tester.java

```
public class Java8Tester {
   final static String salutation = "Hello! ";
   public static void main(String args[]){
   GreetingService greetService1 = message ->
   System.out.println(salutation + mess-
age);
   greetService1.sayMessage("Mahesh");
   }
interface GreetingService {
   void sayMessage(String message);
   }
}
```

Output

Hello! Mahesh

Q) Collections in Java

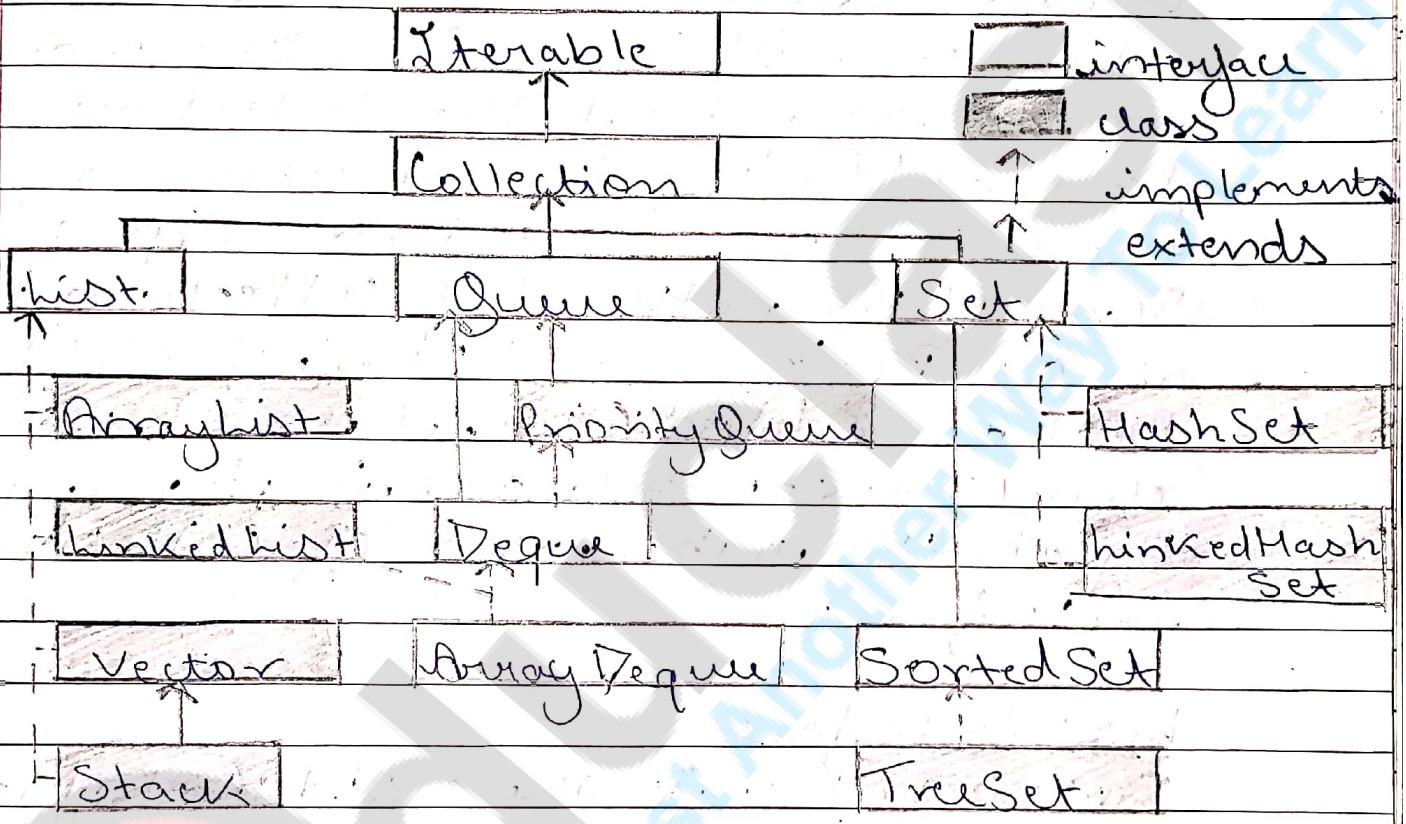
The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation and deletion.

Java collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, Link-

ed HashSet, TreeSet).

Hierarchy of Collection framework

The java.util package contains all the classes and interfaces for the Collection framework.



Iterable Interface

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method i.e.

Iterator<T> iterator()

It returns the iterator over the elements

of type T.

Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the collection interface builds the foundation on which the collection framework depends. Some of the methods of collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of collection interface.

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values. List interface is implemented by the classes ArrayList, LinkedList, Vector and Stack.

To instantiate the List interface, we must use:-

```
List <data-type> list1 = new ArrayList();
List <data-type> list2 = new LinkedList();
List <data-type> list3 = new Vector();
```

List <data-type> list = new Stack();

There are various methods in list interface that can be used to insert, dilute and access the elements from the list.
The classes that implement the list interface are given below.

Arraylist

The Arraylist class implements the list interface. It uses a dynamic array to store the duplicate element of different data types. The Arraylist class maintains the insertion order and is non-synchronized. The elements stored in the Arraylist class can be randomly accessed.

Example

```
import java.util.*;
class TestJavaCollection1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<Str-
  ing>();
 list.add("Ravi");
 list.add("Vijay");
 list.add("Ravi");
 list.add("Ajay");
Iterator itr=list.iterator();
while(itr.hasNext()){
 System.out.println(itr.next());
      }
    }
}
```

Output:
Ravi
Vijay
Ravi
Ajay

LinkedList
LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Example

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList
<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output: Ravi
Vijay
Ravi
Ajay

PAGE No
DATE / /

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, it is synchronized and contains many methods that are not the part of Collection framework.

Example

```
import java.util.*;
public class TestJavaCollection3{
 public static void main(String args[]){
 Vector<String> v=new Vector<String>();
  v.add("Ayush");
  v.add("Amit");
  v.add("Ashish");
  v.add("Garima");
 Iterator<String> itr=v.iterator();
 while(itr.hasNext()){
  System.out.println(itr.next());
    }
   }
}
```

Output

```
Ayush
Amit
Ashish
Garima
```

Stack

The stack is the subclass of Vector. It implements the last-in-first-out data structure i.e Stack. The stack contains all of the methods of Vector class and

also provides its methods like boolean push(), boolean peek(), boolean push(Object o), which defines its properties.

Consider the following eg

```
import java.util.*;
public class TestJavaCollection4 {
 public static void main(String args[]){
 Stack<String> stack = new Stack<String>();
 stack.push("Ayush");
 stack.push("Garvit");
 stack.push("Amit");
 stack.push("Ashish");
  stack.push("Garima");
 stack.pop();
 Iterator<String> itr = stack.iterator();
 while(itr.hasNext()){
 System.out.println(itr.next());
   }
   }
 }
```

Output:

Ayush
Garvit
Amit
Ashish

8] Wildcards in Java

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.

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>

Implementation:

```
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.print("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;
  }
}
```

Output:

```
Total sum is: 22.0
Total sum is: 15.299999999999997
```

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. Thus, 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: <? super A>

Syntax: CollectionType<? super A>

Implementation:

```
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);
printOnlyIntegerClassOrSuperClass(
 list1);
List<Number> list2 = Arrays.asList
(4,5,6,7);
printOnlyIntegerClassOrSuperClass
(list2); }
 public static void printOnlyInteger
ClassOrSuperClass(List<? super In-
teger> list){
  System.out.println(list);
 }
}
```

Output:

[4, 5, 6, 7]
[4, 5, 6, 7]

Here arguments can be Integer or superclass of Integer (which is Number). The method printOnly IntegerClass or SuperClass 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. Use extend wildcard when you want to get values out of a structure and super wildcard when you put values in a structure. Don't use wildcard when you get and put values in a structure.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

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 dont depend on the type parameter.

Implementation:-

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

Output:

```
[1,2,3]
[1.1,2.2,3.3]
```

Q] Maps / HashMap

A map in java is an object that maps keys to values and is designed for the faster lookups. Data is stored in key-value pairs and every key is unique. Each key maps to a value hence the name map. These key-value pairs are called map entries.

In the JDK, java.util.Map is an interface that includes signatures for insertion, removal, and retrieval of elements based on a key. With such methods, it's a perfect tool to use for key-value association mapping. Such as for dictionaries.

Characteristics of Map Interface

- The Map interface is not a true subtype of Collection interface, therefore its characteristics and behaviors are different from the rest of the collection types.
- It provides three collection views - set of keys, set of key-value mappings and collection of values.
- A Map cannot contain duplicate keys and each key can

map to at most one value. Some implementations allow null key and null value (HashMap and LinkedHashMap) but some does not (TreeMap).
- The Map interface doesn't guarantee the order of mappings, however it depends on the implementation. For instance, HashMap doesn't guarantee the order of mappings but TreeMap does.
- AbstractMap class provides a skeletal implementation of the Java Map interface and most of the Map concrete classes extend AbstractMap class and implement required methods.

Implementations of Map

There are several classes that implement the Java Map but three major and general-purpose implementations are HashMap, TreeMap and LinkedHashMap.

HashMap class

- HashMap class extends AbstractMap and implements Map interface.
- It uses a hashtable to store the map. This allows the execution time of get() and put() to

remain same.

- HashMap has four constructor.
  HashMap()
  HashMap(Map<? extends K, ? extends V> m)
  HashMap(int capacity)
  HashMap(int capacity, float fill-ratio)
- HashMap does not maintain order of its element.

Example

```
import java.util.*;
class HashMapDemo {
  public static void main(String
  args[]) {
    HashMap<String, Integer> hm =
    new HashMap<String, Integer>();
    hm.put("a", new Integer(100));
    hm.put("b", new Integer(200));
    hm.put("c", new Integer(300));
    hm.put("d", new Integer(400));
Set<Map.Entry<String, Integer>>
st = hm.entrySet();
for(Map.Entry<String, Integer>
me: st)
{
System.out.print(me.getKey()+
":");
```

```
System.out.println(me.getValue());
    }
}
}
```

Output

```
c 300
a 100
d 400
b 200
```