Module - 6

Generics

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

- We 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.

Rules

All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precede the method's return type (eg: < E >).

```
public static <TypeParameter> ReturnType methodName(TypeParameter parameter)
{    // Method body
}
```

(Commonly used type parameter names are E, T, K, and V, which stand for Element, Type, Key, and Value, respectively. However, you can use any valid identifier as a type parameter name).

- **T stands for "Type". It's commonly used when there's only one type parameter.**
- **E stands for "Element". It's often used in the context of collections, like List<E>, Set<E>, etc.**
- **K stands for "Key" and V stands for "Value". These are often used in the context of maps (Map<K, V>).**

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.

Java Wrapper Classes

Java Wrapper Classes

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Count the number of occurrences of a string in a given string array

```
public static int count(String[] array, String item) {
  int count = 0;

      for (String s : array) {
          if (item.equals(s)) {
              count++;
          }
      }
      return count;
}
```

```
public static void main(String args[]){
String[] helloWorld = {"h", "e", "l", "l", "o", "w", "o", "r", "l", "d"};
int count = count(helloWorld, "l");
System.out.println("#occurrences of l: " + count);}
```

Output

```
#occurrences of l: 3
```

```
public static <T> int count(T[] array, T item) {
    int count = 0;
 for (T element : array) {
            if (item.equals(element)) {
                count++;
            }
        }
return count;
}
```

```
public static void main(String args[])
{
String[] st={"a","b","c","d"};

Integer[] integers = {1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0};
Double[] db={12.3, 4.5, 6.7, 4.5};
Character[] ch={'a','b','c','c','d'};
int count = count(ch,'c');
System.out.println("#occurrences of c: " + count);}
}
```

```
#occurrences of c: 2
```

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.println(element);
      }
      System.out.println();
   }
public static void main(String args[]) {
    // Create arrays of Integer, Double and Character
    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);   // pass an Integer array

    System.out.println("\nArray doubleArray contains:");
    printArray(doubleArray);   // pass a Double array

    System.out.println("\nArray characterArray contains:");
    printArray(charArray);   // pass a Character array
  }
}
```

```
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
```

- By convention, type parameter names are named as single, uppercase letters so that a type parameter can be distinguished easily from an ordinary class or interface name. Following is the list of commonly used type parameter names –
- **E** – Element, and is mainly used by Java Collections framework.
- **K** – Key, and is mainly used to represent parameter type of key of a map.
- **V** – Value, and is mainly used to represent parameter type of value of a map.
- **N** – Number, and is mainly used to represent numbers.
- **T** – Type, and is mainly used to represent first generic type parameter.
- **S** – Type, and is mainly used to represent second generic type parameter.
- **U** – Type, and is mainly used to represent third generic type parameter.
- **V** – Type, and is mainly used to represent fourth generic type parameter.

Generic class

A generic class is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the type parameters (also called type variables) T1, T2, ..., and Tn.

```
class Test<T>
{
// An object of type T is declared
    T obj;
    Test(T obj) {  this.obj = obj;  }  // constructor
    public T getObject()  { return this.obj; }
}
class Main
{
    public static void main (String[] args)
    {
        // instance of Integer type
        Test <Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());
           // instance of String type
        Test <String> sObj =  new Test<String>(“VIT");
        System.out.println(sObj.getObject());
    }
}
```

15

VIT

```
class Test<T, U>
{
   T obj1;  // An object of type T
   U obj2;  // An object of type U
   // constructor
   Test(T obj1, U obj2)
   {        this.obj1 = obj1;
      this.obj2 = obj2;    }
   // To print objects of T and U
   public void print()
   {        System.out.println(obj1);
      System.out.println(obj2);
   } }
class Main {
   public static void main (String[] args)
   {        Test <String, Integer> obj = new Test<String, Integer>(“ABC", 15);
      obj.print();
   } }
```

ABC

15

```
public class Pair<T, U> {
   T first;
   U second;
    public Pair(T first, U second) {
      this.first = first;
      this.second = second;
   }
    public T getFirst() {
      return first;
   }
    public U getSecond() {
      return second;
   }
}
```

```
class Dimension<T>
{
   private T length;
   private T width;
   private T height;

   //Generic constructor
   public Dimension(T length, T width, T height)
   {
      this.length = length;
      this.width = width;
      this.height = height;
   }
}
```

Generic Interface

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public OrderedPair(K key, V value) {
            this.key = key;
            this.value = value;
    }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

Collections Framework in Java

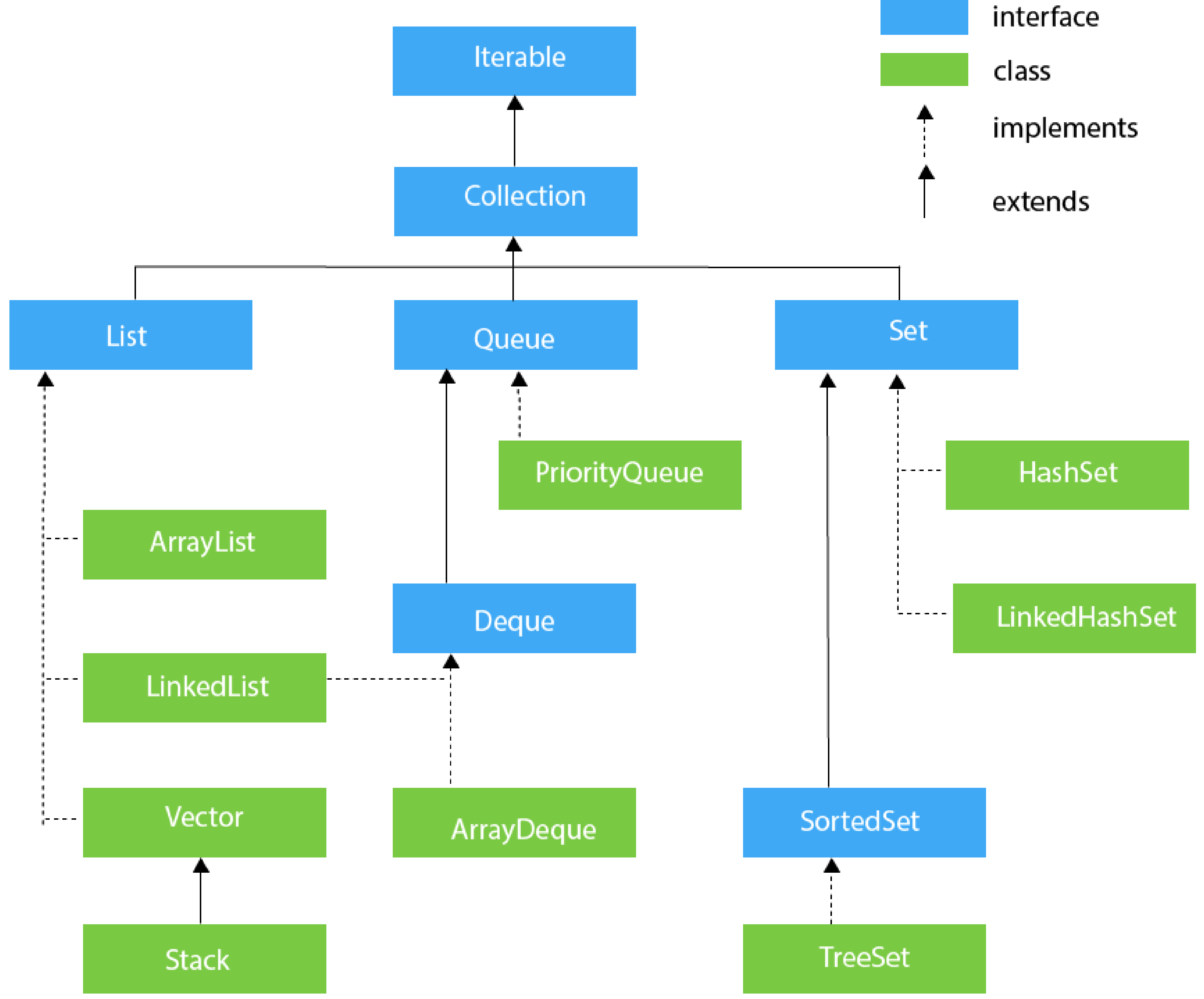
- A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework, **which defines several classes and interfaces** to represent a group of objects as a single unit.(**java.util)**
- The framework provides **an architecture** to store and manipulate the 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 (**Collection, List, Set, Queue, and Map**) and classes (**ArrayList, LinkedList, HashSet, LinkedHashSet, TreeSet, PriorityQueue, HashMap, LinkedHashMap, and TreeMap**).

What is a framework in Java?

- It provides readymade architecture.
- It represents a set of classes and interfaces.

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

- Interfaces and its implementations, i.e., classes
- Algorithm


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSet
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet

- **Collection:** It's the root of the collection hierarchy. A collection represents a group of objects known as its elements. The Collection interface provides methods to perform basic operations like addition/removal of elements, check if an element is present, etc.
- **List:** It's an ordered collection that can contain duplicate elements. You can access elements by their index. ArrayList and LinkedList are popular List implementations.
- **Set:** It's a collection that cannot contain duplicate elements. It models the mathematical set abstraction. HashSet, LinkedHashSet, and TreeSet are popular Set implementations.
- **Queue:** It typically orders elements in a FIFO (first-in-first-out) manner, but exceptions exist. It's used when multiple elements are to be processed. PriorityQueue is a popular Queue implementation.
- **Map:** It's an object that maps keys to values. No duplicate keys can exist in a Map, each key can map to at most one value. It's not actually a subtype of Collection, but it's part of the collections framework. HashMap, LinkedHashMap, and TreeMap are popular Map implementations.

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last elements returned by the iterator. It is less used.

ArrayList

- The ArrayList class implements the List interface.
- It uses a dynamic array(resizable array) to store the duplicate element of different data types.
- The elements stored in the ArrayList class can be randomly accessed.
- Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

```
ArrayList<String> list = new ArrayList<String>();  // Java 7 or earlier
ArrayList<String> list = new ArrayList<>();  // Java 7 or later (diamond operator)
```

```
ArrayList<String> list = new ArrayList<>();
list.add("Java"); // add element
list.add("Python"); // add element
System.out.println(list.get(0)); // get element at index 0
System.out.println(list.size()); // print size of the list
list.remove("Java"); // remove element "Java“
System.out.println(list.contains("Java")); // check if "Java" is in the list
```

```
Java
2
false
```

```
import java.util.*;
  class TestJavaCollection1{
  public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Ravi");//Adding object in arraylist
  list.add("Vijay");
  list.add("Ravi");
  list.add("Ajay");
  //Traversing list through Iterator
  Iterator itr=list.iterator();
  while(itr.hasNext()){
  System.out.println(itr.next());
  }
  }
  }
```

Ravi

Vijay

Ravi

Ajay

Iterating Collection through the for-each loop

```
import java.util.*;
class ArrayList3{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
   //Traversing list through for-each loop
  for(String obj:al)
    System.out.println(obj);
 }
}
```

User-defined class objects in Java ArrayList

```
class Student{
  int rollno;
  String name;
  int age;
  Student(int rollno,String name,int age){
   this.rollno=rollno;
   this.name=name;
   this.age=age;
  }
}
```

```
import java.util.*;
 class ArrayList5{
 public static void main(String args[]){
  //Creating user-defined class objects
  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);
  //creating arraylist
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(s1);//adding Student class object
  al.add(s2);
  al.add(s3);
  //Getting Iterator
  Iterator itr=al.iterator();
  //traversing elements of ArrayList object
  while(itr.hasNext()){
    Student st=(Student)itr.next();
    System.out.println(st.rollno+" "+st.name+" "+st.age);
  }
 }
}
```

Arrays and ArrayLists in Java are both used to store data, but they have some key differences:

Size:

Array: The size of an array is static. This means that an array has a fixed length and you cannot change its size once it is created.

ArrayList: The size of an ArrayList is dynamic. You can add or remove elements from an ArrayList, and its size changes automatically.

Type Safety:

Array: Arrays can store both primitives and objects (except for the exception of autoboxing/unboxing for primitives).

ArrayList: ArrayLists can only store objects. They cannot store primitive data types such as int, char, etc. However, they can store their wrapper classes such as Integer, Character, etc.

Performance:

Array: Access to an array element is faster because it's index based.

ArrayList: ArrayList's get and set methods are slower compared to arrays, as they have a bit of overhead, but the difference is negligible for small amounts of data.

Flexibility:

Array: Arrays do not have built-in methods for adding, removing, or other common operations, so you must implement these operations yourself if you need them.

ArrayList: ArrayLists come with built-in methods for many common operations, such as adding elements, removing elements, and searching for elements. This can make ArrayLists more convenient to use than arrays for many tasks.

Multidimensional:

Array: Arrays can be multidimensional (i.e., an array of arrays).

ArrayList: ArrayLists support only a single dimension. However, you could technically have an ArrayList of ArrayLists.

Memory:

Array: An array is more memory-efficient because it does not have the extra overhead of object methods.

ArrayList: An ArrayList consumes more memory as it contains extra methods and also because each Integer object takes substantially more space than an int.

```
int[] array = new int[10]; // Array
ArrayList<Integer> arrayList = new ArrayList<>(); // ArrayList
```

Autoboxing/ Unboxing

ArrayList<Integer> list = new ArrayList<>(); // ArrayList can only hold objects

int num = 5; // num is a primitive type

list.add(num); // num is autoboxed from int to Integer

int num2 = list.get(0); // the Integer in the ArrayList is unboxed to int

(Primitive to Wrapper –autoboxing and vice versa-unboxing)

- int to Integer
- Integer to int

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements.

In LinkedList, the manipulation is fast because no shifting is required.

```
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());
  }
  }
  }
```

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.

```
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());
  }
  }
  }
```

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.

```
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());
  }
  }
  }
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
after removing two elements:
Raj
Vijay Raj
```

The **Queue** interface defines some methods for acting on the first element of the list, which differ in the way they behave, and the result they provide.

The **peek()** method retrieves the value of the first element of the queue without removing it from the queue. For each invocation of the method we always get the same value and its execution does not affect the size of the queue. If the queue is empty the peek() method returns null.

The **element()** method behaves like peek(), so it again retrieves the value of the first element without removing it. Unlike peek), however, if the list is empty element() throws a NoSuchElementException.

The **poll()** method retrieves the value of the first element of the queue by removing it from the queue. At each invocation it removes the first element of the list and if the list is already empty it returns null but does not throw any exception.

The **remove()** method behaves as the poll() method, so it removes the first element of the list and if the list is empty it throws a NoSuchElementException.