Unit-4 Object Oriented Programming with Java

Outcome of this unit:-

you should be proficient in using Java's Collection Framework to manage and manipulate groups of objects, understand the underlying data structures, and be able to implement various algorithms for sorting and searching. This will enhance your ability to write efficient and effective Java programs.

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.

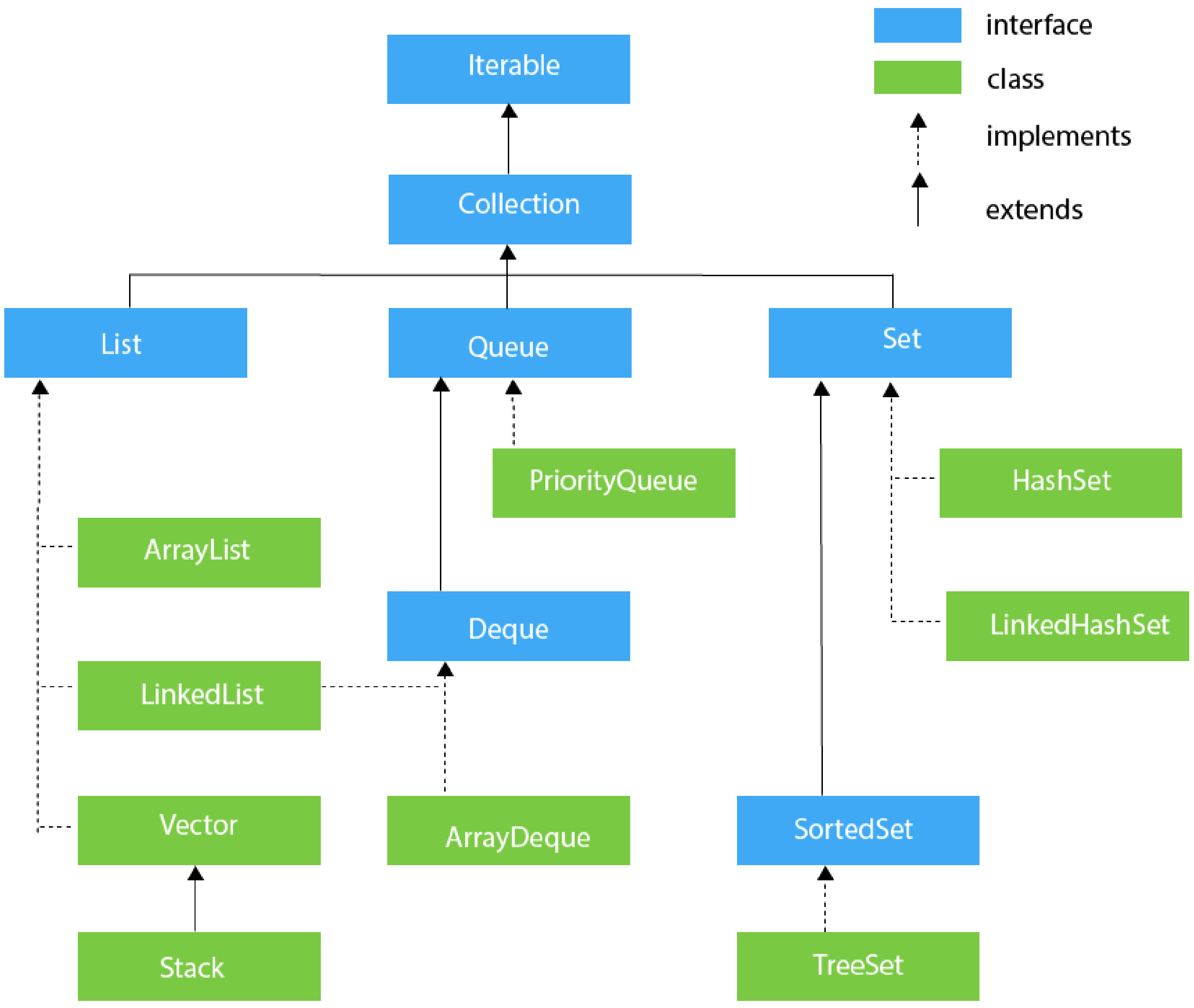
Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit-4 Object Oriented Programming with Java



Iterator interface

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

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

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.

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 :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

The classes that implement the List interface are given below.

Unit-4 Object Oriented Programming with Java

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. Consider the following example.

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

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.

Consider the following example.

Unit-4 Object Oriented Programming with Java

```
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
```

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.

Consider the following 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 example.

```
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
```

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 interface can be instantiated as:

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

There are various classes that implement the Queue interface, some of them are given below.

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.

Consider the following example.

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

Output:

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

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

Unit-4 Object Oriented Programming with Java

1. Set<data-type> s1 = **new** HashSet<data-type>();
2. Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3. Set<data-type> s3 = **new** TreeSet<data-type>();

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Outputs:

```
Vijay
Ravi
Ajay
```

LinkedHashSet

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit-4 Object Oriented Programming with Java

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```java
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ravi
Vijay
Ajay
```

SortedSet Interface

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

The SortedSet can be instantiated as:

1. SortedSet<data-type> set = **new** TreeSet();

TreeSet

Unit-4 Object Oriented Programming with Java

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;
public class TestJavaCollection9{
/public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

```
Ajay
Ravi
Vijay
```

Java Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

Java Map Example: Non-Generic (Old Style)

```
//Non-generic
```

Unit-4 Object Oriented Programming with Java

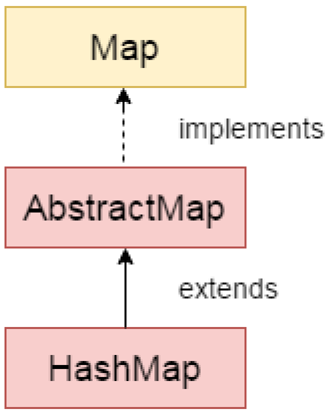
```
import java.util.*;
public class MapExample1 {
public static void main(String[] args) {
    Map map=new HashMap();
    //Adding elements to map
    map.put(1,"Amit");
    map.put(5,"Rahul");
    map.put(2,"Jai");
    map.put(6,"Amit");
    //Traversing Map
    Set set=map.entrySet();//Converting to Set so that we can traverse
    Iterator itr=set.iterator();
    while(itr.hasNext()){
        //Converting to Map.Entry so that we can get key and value separately
        Map.Entry entry=(Map.Entry)itr.next();
        System.out.println(entry.getKey()+" "+entry.getValue());
    }
}
}
```

Output:

```
1 Amit
2 Jai
5 Rahul
6 Amit
```

Java HashMap

Unit-4 Object Oriented Programming with Java



Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

Java HashMap Example

Let's see a simple example of HashMap to store key and value pair.

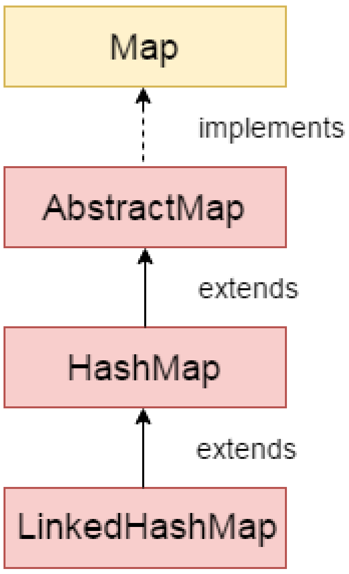
```
import java.util.*;
public class HashMapExample1{
 public static void main(String args[]){
   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating
 HashMap
   map.put(1,"Mango");  //Put elements in Map
   map.put(2,"Apple");
   map.put(3,"Banana");
   map.put(4,"Grapes");

   System.out.println("Iterating Hashmap...");
   for(Map.Entry m : map.entrySet()){
    System.out.println(m.getKey()+" "+m.getValue());
   }
}
}
```

Test it Now

Unit-4 Object Oriented Programming with Java

```
Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes
```

Java LinkedHashMap class



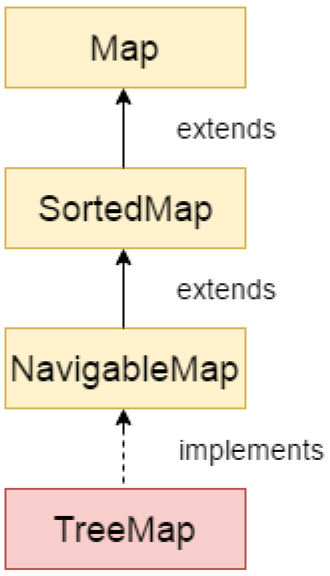
Java LinkedHashMap class is Hashtable and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

Java LinkedHashMap Example

```
import java.util.*;
class LinkedHashMap1{
 public static void main(String args[]){

  LinkedHashMap<Integer,String> hm=new LinkedHashMap<Integer,String>(
);

  hm.put(100,"Amit");
  hm.put(101,"Vijay");
  hm.put(102,"Rahul");

for(Map.Entry m:hm.entrySet()){
  System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

```
Output:100 Amit
       101 Vijay
       102 Rahul
```

Java TreeMap class



Java TreeMap class is a red-black tree based implementation. It provides an efficient means of storing key-value pairs in sorted order.

Java TreeMap Example

```
import java.util.*;
class TreeMap1{
 public static void main(String args[]){
   TreeMap<Integer,String> map=new TreeMap<Integer,String>();
```

```
        map.put(100,"Amit");
        map.put(102,"Ravi");
        map.put(101,"Vijay");
        map.put(103,"Rahul");

        for(Map.Entry m:map.entrySet()){
         System.out.println(m.getKey()+" "+m.getValue());
        }
  }
 }
```

```
Output:100 Amit
       101 Vijay
       102 Ravi
       103 Rahul
```

Java Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

Java Hashtable Example

```
import java.util.*;
class Hashtable1{
 public static void main(String args[]){
  Hashtable<Integer,String> hm=new Hashtable<Integer,String>();

  hm.put(100,"Amit");
  hm.put(102,"Ravi");
  hm.put(101,"Vijay");
  hm.put(103,"Rahul");

  for(Map.Entry m:hm.entrySet()){
   System.out.println(m.getKey()+" "+m.getValue());
  }
 }
}
```

Unit-4 Object Oriented Programming with Java

Test it Now

Output:

```
103 Rahul
102 Ravi
101 Vijay
100 Amit
```

Sorting in Collection

We can sort the elements of:

1. String objects
2. Wrapper class objects
3. User-defined class objects

Collections class provides static methods for sorting the elements of a collection. If collection eleme
we can use TreeSet. However, we cannot sort the elements of List. Collections class provides met
elements of List type elements.

Method of Collections class for sorting List elements

public void sort(List list): is used to sort the elements of List. List elements must be of the Comparable type.

Note: String class and Wrapper classes implement the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.

Example to sort string objects

```
import java.util.*;
class TestSort1{
public static void main(String args[]){

ArrayList<String> al=new ArrayList<String>();
al.add("Viru");
al.add("Saurav");
al.add("Mukesh");
al.add("Tahir");
```

Unit-4 Object Oriented Programming with Java

```

Collections.sort(al);
Iterator itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
 }
}
}
```

Test it Now

```
Mukesh
Saurav
Tahir
Viru
```

Example to sort Wrapper class objects

```
import java.util.*;
class TestSort3{
public static void main(String args[]){

ArrayList al=new ArrayList();
al.add(Integer.valueOf(201));
al.add(Integer.valueOf(101));
al.add(230);//internally will be converted into objects as Integer.valueOf(230)

Collections.sort(al);

Iterator itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
 }
}
}
```

```
101
201
230
```

Example to sort user-defined class objects

```java
import java.util.*;

class Student implements Comparable<Student> {
    public String name;
  public Student(String name) {
    this.name = name;
  }
  public int compareTo(Student person) {
    return name.compareTo(person.name);

  }
}
public class TestSort4 {
  public static void main(String[] args) {
      ArrayList<Student> al=new ArrayList<Student>();
      al.add(new Student("Viru"));
      al.add(new Student("Saurav"));
      al.add(new Student("Mukesh"));
      al.add(new Student("Tahir"));

    Collections.sort(al);
    for (Student s : al) {
      System.out.println(s.name);
    }
  }
}
```

```
Mukesh
Saurav
Tahir
Viru
```

java Comparable interface

Unit-4 Object Oriented Programming with Java

Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

Collections class

Collections class provides static methods for sorting the elements of collections. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements.

Method of Collections class for sorting List elements

public void sort(List list): It is used to sort the elements of List. List elements must be of the Comparable type.

Note: String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.

Java Comparable Example

Let's see the example of the Comparable interface that sorts the list elements on the basis of age.

File: Student.java

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

public int compareTo(Student st){
if(age==st.age)
return 0;
else if(age>st.age)
return 1;
else
return -1;
}
}
```

Java Comparator interface

Java Comparator interface is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Methods of Java Comparator Interface

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Collections class

Collections class provides static methods for sorting the elements of a collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

Method of Collections class for sorting List elements

public void sort(List list, Comparator c): is used to sort the elements of List by the given Comparator.

Java Comparator Example (Non-generic Old Style)

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

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

AgeComparator.java

This class defines comparison logic based on the age. If the age of the first object is greater than the second, we are returning a positive value. It can be anyone such as 1,

Unit-4 Object Oriented Programming with Java

2, 10. If the age of the first object is less than the second object, we are returning a negative value, it can be any negative value, and if the age of both objects is equal, we are returning 0.

```java
import java.util.*;
class AgeComparator implements Comparator{
public int compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
}
```

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

```java
import java.util.*;
class NameComparator implements Comparator{
public int compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;

return s1.name.compareTo(s2.name);
}
}
```

Simple.java

Unit-4 Object Oriented Programming with Java

In this class, we are printing the values of the object by sorting on the basis of name and age.

```
import java.util.*;

import java.io.*;

class Simple{
public static void main(String args[]){

ArrayList al=new ArrayList();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

System.out.println("Sorting by Name");

Collections.sort(al,new NameComparator());
Iterator itr=al.iterator();
while(itr.hasNext()){
Student st=(Student)itr.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}

System.out.println("Sorting by age");

Collections.sort(al,new AgeComparator());
Iterator itr2=al.iterator();
while(itr2.hasNext()){
Student st=(Student)itr2.next();
System.out.println(st.rollno+" "+st.name+" "+st.age);
}


}
```

```java
}
```

```
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23

Sorting by age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Java Comparator Example (Generic)

Student.java

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

AgeComparator.java

```java
import java.util.*;
class AgeComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
return -1;
}
}
```

NameComparator.java

Unit-4 Object Oriented Programming with Java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

```java
import java.util.*;
class NameComparator implements Comparator<Student>{
public int compare(Student s1,Student s2){
return s1.name.compareTo(s2.name);
}
}
```

Simple.java

In this class, we are printing the values of the object by sorting on the basis of name and age.

```java
import java.util.*;
import java.io.*;
class Simple{
public static void main(String args[]){

ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));

System.out.println("Sorting by Name");

Collections.sort(al,new NameComparator());
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}

System.out.println("Sorting by age");

Collections.sort(al,new AgeComparator());
for(Student st: al){
System.out.println(st.rollno+" "+st.name+" "+st.age);
}
}
}
```

```
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23

Sorting by age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Java 8 Comparator interface

Java 8 Comparator interface is a functional interface that contains only one abstract method. Now, we can use the Comparator interface as the assignment target for a lambda expression or method reference.

Methods of Java 8 Comparator Interface

Method	Description
int compare(T o1, T o2)	It compares the first object with second object.
static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor)	It accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator that compares by that sort key.
static <T,U> Comparator<T> comparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	It accepts a function that extracts a sort key from a type T, and returns a Comparator that compares by that sort key using the specified Comparator.

Unit-4 Object Oriented Programming with Java

static <T> Comparator<T> comparingDouble(ToDoubleFunction<? super T> keyExtractor)	It accepts a function that extracts a double sort key from a type T, and returns a Comparator that compares by that sort key.
static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)	It accepts a function that extracts an int sort key from a type T, and returns a Comparator that compares by that sort key.
static <T> Comparator<T> comparingLong(ToLongFunction<? super T> keyExtractor)	It accepts a function that extracts a long sort key from a type T, and returns a Comparator that compares by that sort key.
boolean equals(Object obj)	It is used to compare the current object with the specified object.
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()	It returns a comparator that compares Comparable objects in natural order.
static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)	It returns a comparator that treats null to be less than non-null elements.
static <T> Comparator<T> nullsLast(Comparator<? super T> comparator)	It returns a comparator that treats null to be greater than non-null elements.
default Comparator<T> reversed()	It returns comparator that contains reverse ordering of the provided comparator.
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()	It returns comparator that contains reverse of natural ordering.

default Comparator<T> thenComparing(Comparator<? super T> other)	It returns a lexicographic-order comparator with another comparator.
default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a Comparable sort key.
default <U> Comparator<T> thenComparing(Function<? super T,? extends U> keyExtractor, Comparator<? super U> keyComparator)	It returns a lexicographic-order comparator with a function that extracts a key to be compared with the given Comparator.
default Comparator<T> thenComparingDouble(ToDoubleFunction<? super T> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a double sort key.
default Comparator<T> thenComparingInt(ToIntFunction<? super T> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a int sort key.
default Comparator<T> thenComparingLong(ToLongFunction<? super T> keyExtractor)	It returns a lexicographic-order comparator with a function that extracts a long sort key.

Java 8 Comparator Example

Let's see the example of sorting the elements of List on the basis of age and name.

File: Student.java

```
class Student {
   int rollno;
```

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

   public int getRollno() {
       return rollno;
   }

   public void setRollno(int rollno) {
       this.rollno = rollno;
   }

   public String getName() {
       return name;
   }

   public void setName(String name) {
       this.name = name;
   }

   public int getAge() {
       return age;
   }

   public void setAge(int age) {
       this.age = age;
   }

   }
```

File: TestSort1.java

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

Unit-4 Object Oriented Programming with Java

```
import java.util.*;
public class TestSort1{
public static void main(String args[]){
ArrayList<Student> al=new ArrayList<Student>();
al.add(new Student(101,"Vijay",23));
al.add(new Student(106,"Ajay",27));
al.add(new Student(105,"Jai",21));
/Sorting elements on the basis of name
Comparator<Student> cm1=Comparator.comparing(Student::getName);
 Collections.sort(al,cm1);
 System.out.println("Sorting by Name");
 for(Student st: al){
   System.out.println(st.rollno+" "+st.name+" "+st.age);
   }
 //Sorting elements on the basis of age
  Comparator<Student> cm2=Comparator.comparing(Student::getAge);
Collections.sort(al,cm2);
 System.out.println("Sorting by Age");
 for(Student st: al){
   System.out.println(st.rollno+" "+st.name+" "+st.age);
   }
}
}
```

```
Sorting by Name
106 Ajay 27
105 Jai 21
101 Vijay 23
Sorting by Age
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Properties class in Java

The **properties** object contains key and value pair both as a string. The java.util.Properties class is the subclass of Hashtable.

Faculty: SHAHRUKH KAMAL
Shahrukhkamal7@gmail.com

It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

Recompilation is not required if the information is changed from a properties file: If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

Constructors of Properties class

Method	Description
Properties()	It creates an empty property list with no default values.
Properties(Properties defaults)	It creates an empty property list with the specified defau

Methods of Properties class

The commonly used methods of Properties class are given below.

Method	Description
public void load(Reader r)	It loads data from the Reader object.
public void load(InputStream is)	It loads data from the InputStream object
public void loadFromXML(InputStream in)	It is used to load all of the properties represented by the XML document on the specified input stream into this properties table.
public String getProperty(String key)	It returns value based on the key.
public String getProperty(String key, String defaultValue)	It searches for the property with the specified key.
public void setProperty(String key, String value)	It calls the put method of Hashtable.

public void list(PrintStream out)	It is used to print the property list out to the specified output stream.
public void list(PrintWriter out))	It is used to print the property list out to the specified output stream.
public Enumeration<?> propertyNames())	It returns an enumeration of all the keys from the property list.
public Set<String> stringPropertyNames()	It returns a set of keys in from property list where the key and its corresponding value are strings.
public void store(Writer w, String comment)	It writes the properties in the writer object.
public void store(OutputStream os, String comment)	It writes the properties in the OutputStream object.
public void storeToXML(OutputStream os, String comment)	It writes the properties in the writer object for generating XML document.
public void storeToXML(Writer w, String comment, String encoding)	It writes the properties in the writer object for generating XML document with the specified encoding.

Example of Properties class to get information from the properties file

To get information from the properties file, create the properties file first.

db.properties

1. user=system
2. password=oracle

Now, let's create the java class to read the data from the properties file.

Test.java

Unit-4 Object Oriented Programming with Java

```
import java.util.*;
import java.io.*;
public class Test {
public static void main(String[] args)throws Exception{
    FileReader reader=new FileReader("db.properties");

    Properties p=new Properties();
    p.load(reader);

    System.out.println(p.getProperty("user"));
    System.out.println(p.getProperty("password"));
}
}
```

```
Output:system
       oracle
```

Now if you change the value of the properties file, you don't need to recompile the java class. That means no maintenance problem.

Example of Properties class to get all the system properties

By System.getProperties() method we can get all the properties of the system. Let's create the class that gets information from the system properties.

Test.java

```
import java.util.*;
import java.io.*;
public class Test {
public static void main(String[] args)throws Exception{

Properties p=System.getProperties();
Set set=p.entrySet();

Iterator itr=set.iterator();
while(itr.hasNext()){
```

Unit-4 Object Oriented Programming with Java

```
Map.Entry entry=(Map.Entry)itr.next();
System.out.println(entry.getKey()+" = "+entry.getValue());
}

}
}
```

```
Output:
java.runtime.name = Java(TM) SE Runtime Environment
sun.boot.library.path = C:\Program Files\Java\jdk1.7.0_01\jre\bin
java.vm.version = 21.1-b02
java.vm.vendor = Oracle Corporation
java.vendor.url = http://java.oracle.com/
path.separator = ;
java.vm.name = Java HotSpot(TM) Client VM
file.encoding.pkg = sun.io
user.country = US
user.script =
sun.java.launcher = SUN_STANDARD
...........
```

Example of Properties class to create the properties file

Now let's write the code to create the properties file.

Test.java

```java
import java.util.*;
import java.io.*;
public class Test {
public static void main(String[] args)throws Exception{

Properties p=new Properties();
p.setProperty("name","Sonoo Jaiswal");
p.setProperty("email","sonoojaiswal@javatpoint.com");

p.store(new FileWriter("info.properties"),"Javatpoint Properties Example");

}
}
```

Let's see the generated properties file.

Unit-4 Object Oriented Programming with Java

ADVERTISEMENT

info.properties

```
#Javatpoint Properties Example
#Thu Oct 03 22:35:53 IST 2013
email=sonoojaiswal@javatpoint.com
name=Sonoo Jaiswal
```