UNIT-IV

Collection Overview

Collection

A Collection is a group of individual objects represented as a single unit.

Framework

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

Collection Framework

It is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- Interfaces
- Interface Implementer Classes

Where use Collection Framework

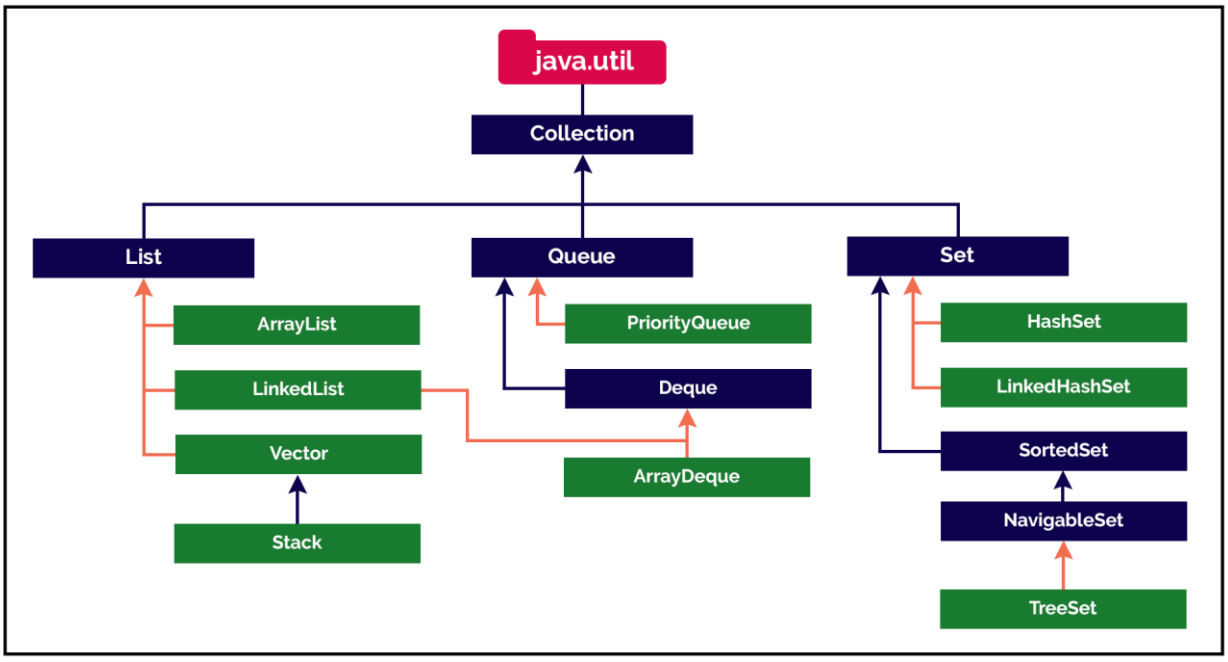
All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

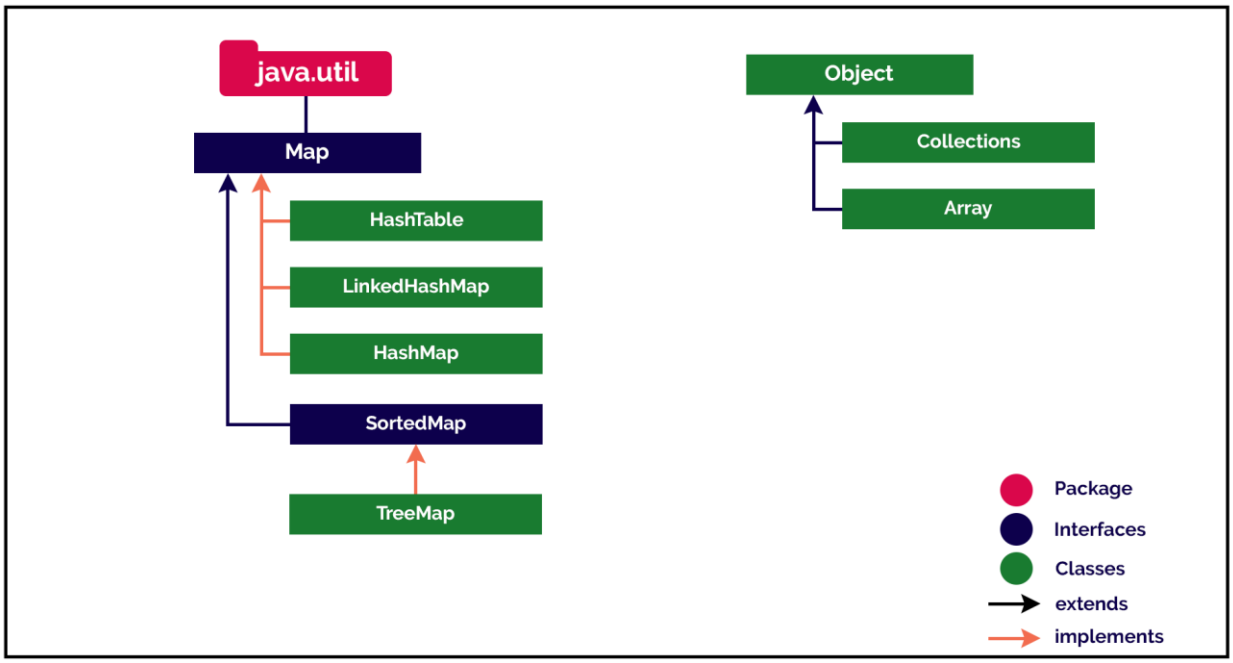
Package

java.util package

Collection Framework Hierarchy

Almost all collections in Java are derived from the java.util.Collection interface





Data Accessing (Or) Retrieving Techniques form Collection Framework

Technique to retrieve elements from Collection object

Java support following technique to retrieve the elements from any collection object.

1. foreach loop
2. Iterator interface

1.foreach

It is similar to for loop used to retrieve elements of collection objects (until the last element)

Syntax

```
for(datatype variable:collection-object)
{
.....
.....
}
```

The above looping statement executed repeatedly several number of time until the elements are available in the collection object, the loop will be terminated if no elements found.

Note: foreach loop always traversing in forward direction.

Example

```
import java.util.*;
class ForeachDemo
{
    public static void main(String args[])
    {
        ArrayList<Integer> al=new ArrayList<Integer>();  // creating arraylist
        al.add(10);
        al.add(20);
        al.add(30);
        for(int i : al)
        {
            System.out.println(i);
        }
    }
}
```

Output

```
10
20
30
```

2.Iterator Interface

- It is one of the predefined interface present in java.util.* package.
- The purpose of this interface is that to extract or retrieve the elements of collection variable only in forward direction but not in backward direction.
- By default an object of iterator is pointing just before the first element of any collection framework variable.

Methods

1. **Public Boolean hasNext()**
   This method return true provided by Iterator interface object is having next element otherwise it returns false.

2. **public object next()**
   This method is used for retrieving next element of any collection framework variable provided public booleanhasNext(). If next elements are available then this method returns true other wise it return false.

3. **public object remove()**
   Remove from collection the last element returned by Iterator.

Example

```
importjava.util.*;
classIteratorDemo
{
   publicstaticvoid main(Stringargs[])
   {
      ArrayList<Integer> al=newArrayList<Integer>();  // creating arraylist
      al.add(10);
      al.add(20);
      al.add(30);
      Iterator itr=al.iterator();   // getting Iterator from arraylist to traverse
       elements
      while(itr.hasNext())
      {
          System.out.println(itr.next());
      }
   }
}
```

Output

```
10
20
30
```

Interfaces

1. Collection Interface
2. List Interface
3. Queue Interface
4. Deque Interface
5. Set Interface
6. Sortedset Interface
7. Map Inetrface

Collection Interface

- The Collection interface is the root interface for most of the interfaces and classes of collection framework.
- The Collection interface is available inside the java.util package. It defines the methods that are commonly used by almost all the collections.

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 in following collection classes

1. ArrayList
2. LinkedList
3. Vector
4. 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> list4 = new Stack();
```

1.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 ArrayListDemo
{
        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
```

2.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
```

3.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
```

4.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 push(),pop().

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.

Queue interface is implemented in following collection class

- PriorityQueue

Queue interface can be instantiated as:

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.

Example

```
import java.util.*;
class PriorityQueueDemo
{
    public static void main(String args[])
    {
            PriorityQueue<String> queue=new PriorityQueue<String>();
            queue.add("Deo");
            queue.add("Smith");
            queue.add("Piter");
            queue.add("Poter");
            System.out.println("Iterating the queue Elements:");
            Iterator itr=queue.iterator();
           while(itr.hasNext())
           {
              System.out.println(itr.next());
           }
    }
 }
```

Output

```
Iterating the queue Elements:
Deo
Smith
Piter
Poter
```

Deque Interface

- Deque interface extends the Queue interface.
- In Deque, we can remove and add the elements from both the side.
- Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque Interface is implemented in following collection class

- ArrayDeque

Deque can be instantiated as:

Deque d = new ArrayDeque();

ArrayDeque

- ArrayDeque class implements the Deque interface.
- It facilitates us to use the Deque.
- Unlike queue, we can add or delete the elements from both the ends.
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Example

```
import java.util.*;
public class TestJavaCollection6
{
    public static void main(String[] args)
    {
        Deque<String> deque = new ArrayDeque<String>();
        deque.add("Gautam");
        deque.add("Karan");
        deque.add("Ajay");
        for (String str : deque)
        {
            System.out.println(str);
        }
    }
}
```

Output:

```
Gautam
Karan
Ajay
```

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 Interface is implemented in following collection classes

1. HashSet
2. LinkedHashSet
3. TreeSet

Set can be instantiated as:

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

1.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.

Example

```
import java.util.*;
public class TestJavaCollection7
{
    public static void main(String args[])
    {
        HashSet<String> set=new HashSet<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:

Vijay
Ravi
Ajay

2.LinkedHashSet

- 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.

Example

```
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.

SortedSet Interface is implemented in following collection class

- TreeSet

The SortedSet can be instantiated as:

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

TreeSet

- 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.

Example

```
import java.util.*;
public class TestJavaCollection9
{
    public static void main(String args[])
    {
        TreeSet<String> set=new TreeSet<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:

Ajay
Ravi
Vijay

Map Interface in Java

- Maps are defined by the java.util.Map interface in Java.
- Map is an Interface which store key, value in which both key and value are objects Key must be unique and value can be duplicate.
- Here values are elements which store in Map.

Map Interface is implemented in the following two classes.

1. HashMap
2. HashTable

Methods of Map Interface

1. **public int size()**
   This method is used for finding the number of entries in any 2-D Collection framework variable.

2. **public boolean isEmpty()**
   This method returns true provided 2-D Collection framework is empty (size=0).Returns false in the case of non-empty (size>0).

3. **public void put(Object,Object)**
   This method is used for adding and modifying the entries of 2-D Collection framework variable. If the (key, value) pair is not existing then such (k,v)pair will be added to the 2-D Collection framework variable and it will be considered as Inserted Entry. In the (k,v) pair ,if the value of key is already present in 2-D collection framework variable then the existing value of value is replaced by new value of key.

1.HashMap

- HashMap is the Implementer class of Map Interface, which store the values based on the key.
- HashMap is not allows to store duplicate elements.
- HashMap is new collection framework class.
- HashMap may have one null key and multiple null values.

- For retrieving elements from HashMap you can use foreach loop, Iterator Interface to retrieve the elements.
- HashMap is not Synchronized means multiple threads can work Simultaneously.

Example

```
import java.util.*;
class HashMapDemo
{
  public static void main(String args[])
  {
       HashMap<Integer,String> hm=new HashMap<Integer,String>();
       hm.put(1,"Deo");
       hm.put(2,"zen");
       hm.put(3,"porter");
       hm.put(4,"piter");
      for(Map.Entry m:hm.entrySet())
      {
          System.out.println(m.getKey()+" "+m.getValue());
      }
   }
}
```

Output

```
1 Deo
2 zen
3 porter
4 piter
```

2.HashTable in Java

- HashTable is **Implementer** class of Map interface and extends Dictionary class. HashTable does not allows null key and null values, these elements will be stored in a random order.
- HashTable is a legacy class, which will uses hashing technique (the elements will be stored in unsorted or un-order format).
- HashTable stored the elements in key values format.
- HashTable does not allows null keys and null values.
- HashTable is Synchronized.
- HashTable allows heterogeneous elements.
- For retrieve elements from HashTable you can use foreach loop, Iterator Interface

Note: HashTable class also contains same methods like HashMap.

Example

```
import java.util.*;
class HashTableDemo
{
    public static void main(String args[])
    {
        HashTable<Integer,String> ht=new HashTable<Integer,String>();
        ht.put(1,"Deo");
        ht.put(2,"zen");
        ht.put(3,"porter");
        ht.put(4,"piter");
        System.out.println(ht);
     }
}
```

Output

[1=Deo, 2=zen, 3=porter ,4=piter]

Comparator in java

- The Comparator is an interface available in the **java.util** package.
- The java **Comparator** is used to order the objects of user-defined classes. The java Comparator can compare two objects from two different classes.
- Using the java Comparator, we can sort the elements based on data members of a class.
- For example, we can sort based on roolNo, age, salary, marks, etc.

The Comparator interface has the following methods.

Method	Description
int compare(Object obj1, Object obj2)	It is used to compares the obj1 with obj2 .
boolean equals(Object obj)	It is used to check the equity between current object and argumented object.

We use the following steps to use Comparator with a seperate class.

Step - 1: Create the user-defined class.

Step - 2: Create a class that implements Comparator interface.

Step -3: Implement the comapare( ) method of Comparator interface inside the above defined class(step - 2).

Step -4: Create the actual class where we use the Compatator object with sort method of Collections class.

Step - 5: Create the object of Compatator interface using the class crearted in step - 2.

Step - 6: Call the sort method of Collections class by passing the object created in step - 6.

Step - 7: Use a for-each (any loop) to print the sorted information.

Let's consider an example program to illustrate Comparator using a separate class.

How does Collections.Sort() work?

Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method returns -1, 0, or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for their sort.

Example

```
import java.util.*;
class Student
{
        String name;
        float percentage;
        Student(String name, float percentage)
        {
                this.name = name;
                this.percentage = percentage;
        }
}
class PercentageComparator implements Comparator<Student>
{
        public int compare(Student stud1, Student stud2)
        {
                if(stud1.percentage < stud2.percentage)
                        return 1;
                return -1;
        }
```

```
}
public class StudentCompare
{
	public static void main(String args[])
	{
		ArrayList<Student> studList = new ArrayList<Student>();
		studList.add(new Student("Gouthami", 90.61f));
		studList.add(new Student("Raja", 83.55f));
		studList.add(new Student("Honey", 85.55f));
		studList.add(new Student("Teja", 77.56f));
		studList.add(new Student("Varshith", 80.89f));
		Comparator<Student> com = new PercentageComparator();
		Collections.sort(studList, com);
		System.out.println("Percentage% --> Name");
		for(Student stud:studList)
		{
			System.out.println(stud.percentage + " --> " + stud.name);
		}
	}
}
```

Output:

```
Percentage% --> Name
90.61 --> Gouthami
85.55 --> Honey
83.55 --> Raja
80.89 --> Varshith
77.56 --> Teja
```

Collection algorithms in java

- The java collection framework defines several algorithms as static methods that can be used with collections and map objects.
- All the collection algorithms in the java are defined in a class called **Collections** which defined in the **java.util** package.
- All these algorithms are highly efficient and make coding very easy. It is better to use them than trying to re-implement them.

The collection framework has the following methods as algorithms.

Method	Description
void sort(List list)	Sorts the elements of the list as determined by their natural ordering.
void sort(List list, Comparator comp)	Sorts the elements of the list as determined by Comparator comp.
void reverse(List list)	Reverses all the elements sequence in list.
void rotate(List list, int n)	Rotates list by n places to the right. To rotate left, use a negative value for n.
void shuffle(List list)	Shuffles the elements in list.
void shuffle(List list, Random r)	Shuffles the elements in the list by using r as a source of random numbers.
void copy(List list1, List list2)	Copies the elements of list2 to list1.
void swap(List list, int idx1, int idx2)	Exchanges the elements in the list at the indices specified by idx1 and idx2.
int binarySearch(List list, Object value)	Returns the position of value in the list (must be in the sorted order), or -1 if value is not found.
Object max(Collection c)	Returns the largest element from the collection c as determined by natural ordering.
Object max(Collection c, Comparator comp)	Returns the largest element from the collection c as determined by Comparator comp.
Object min(Collection c)	Returns the smallest element from the collection c as determined by natural ordering.
Object min(Collection c, Comparator comp)	Returns the smallest element from the collection c as determined by Comparator comp.
void fill(List list, Object obj)	Assigns obj to each element of the list.
boolean replaceAll(List list, Object old, Object new)	Replaces all occurrences of old with new in the list.
Enumeration enumeration(Collection c)	Returns an enumeration over Collection c.
ArrayList list(Enumeration enum)	Returns an ArrayList that contains the elements of enum.

Example

```
import java.util.*;
public class CollectionAlgorithmsExample
{
```

```
        public static void main(String[] args)
        {
                ArrayList list = new ArrayList();
                PriorityQueue queue = new PriorityQueue();
                HashSet set = new HashSet();
                HashMap map = new HashMap();
                Random num = new Random();
                for(int i = 0; i < 5; i++)
                {
                        list.add(num.nextInt(100));
                        queue.add(num.nextInt(100));
                        set.add(num.nextInt(100));
                        map.put(i, num.nextInt(100));
                }
                System.out.println("List => " + list);
                System.out.println("Queue => " + queue);
                System.out.println("Set => " + set);
                System.out.println("Map => " + map);
                System.out.println("---------------------------------------");
                Collections.sort(list);
                System.out.println("List in ascending order => " + list);
                System.out.println("Largest element in set => " + Collections.max(set));
        System.out.println("Smallest element in queue => " +      Collections.min(queue));
                Collections.reverse(list);
                System.out.println("List in reverse order => " + list);
                Collections.shuffle(list);
                System.out.println("List after shuffle => " + list);
        }
}
```

Output:

```
List => [94, 47, 76, 25, 89]
Queue => [8, 26, 9, 57, 91]
Set => [17, 24, 88, 41, 76]
Map => {0=39, 1=91, 2=76, 3=97, 4=96}
---------------------------------------
List in ascending order => [25, 47, 76, 89, 94]
Largest element in set => 88
```

Smallest element in queue => 8
List in reverse order => [94, 89, 76, 47, 25]
List after shuffle => [89, 25, 94, 47, 76]

Arrays class in java

- The java collection framework has a class Arrays that provides methods for creating dynamic array and perform various operations like search, asList, campare, etc.
- The Arrays class in java is defined in the **java.util** package. All the methods defined by Arrays class are static methods.

The Arrays class in java has the following methods.

Method	Description
List<T> asList(T[] arr)	It returns a fixed-size list backed by the specified Arrays.
int binarySearch(T[] arr, element)	It searches for the specified element in the array with the help of Binary Search algorithm, and returns the position.
boolean equals(T[] arr1, T[] arr2)	It returns true if the two specified arrays of booleans are equal to one another, otherwise retruns false.
int hashCode(T[] arr)	It returns the hash code for the specified array.
String toString(T[] arr)	It Returns a string representation of the contents of the specified array.
void fill(T[] arr, T value)	It assigns the specified value to each element of the specified array.
void setAll(T[] arr, FunctionGenerator)	It sets all elements of the specified array, using the provided generator function to compute each element.
void sort(T[] arr)	It sorts the specified array into ascending order.

Example

```
import java.util.*;
public class ArraysClassExample
{
        public static void main(String[] args)
        {
                int[] arr1 = {10, 3, 50, 7, 30, 66, 28, 54, 42};
                int[] arr2 = {67, 2, 54, 67, 13, 56, 98};
```

```
                System.out.print("Array1 => ");
                for(int i:arr1)
                        System.out.print(i + ", ");
                System.out.print("\nArray2 => ");
                for(int i:arr2)
                        System.out.print(i + ", ");
                System.out.println("\n----------------------------------------");
                System.out.println("Array1 as List => " + Arrays.asList(arr1));
        System.out.println("Position of 30 in Array1 => " + Arrays.binarySearch(arr1, 30));
        System.out.println("equity of array1 and array2 => " + Arrays.equals(arr1, arr2));
                System.out.println("Hash code of Array1 => " + Arrays.hashCode(arr1));
                Arrays.fill(arr1, 15);
                System.out.print("fill Array1 with 15 => ");
                for(int i:arr1)
                        System.out.print(i + ", ");
                Arrays.sort(arr2);
                System.out.print("\nArray2 in sorted order => ");
                for(int i:arr2)
                        System.out.print(i + ", ");
        }
}
```

Output:

```
Array1 => 10, 3, 50, 7, 30, 66, 28, 54, 42,
Array2 => 67, 2, 54, 67, 13, 56, 98,
----------------------------------------Array1 as List => [10, 3, 50, 7, 30, 66, 28, 54, 42]
Position of 30 in Array1 => 4
equity of array1 and array2 => false
Hash code of Array1 => 1497890365
fill Array1 with 15 => 15, 15, 15, 15, 15, 15, 15, 15, 15,
Array2 in sorted order => 2, 13, 54, 56, 67, 67, 98,
```

Dictionary class in java

- In java, the package **java.util** contains a class called **Dictionary** which works like a Map. The Dictionary is an abstract class used to store and manage elements in the form of a pair of key and value.
- The Dictionary stores data as a pair of key and value. In the dictionary, each key associates with a value. We can use the key to retrieve the value back when needed.
- The Dictionary class is no longer in use, it is obsolete.

- As Dictionary is an abstract class we can not create its object. It needs a child class like Hashtable.

The Dictionary class in java has the following methods.

S. No.	Methods with Description
1	**Dictionary( )**
	It's a constructor.
2	**Object put(Object key, Object value)**
	Inserts a key and its value into the dictionary. Returns null on success; returns the previous value associated with the key if the key is already exist.
3	**Object remove(Object key)**
	It returns the value associated with given key and removes the same; Returns null if the key does not exist.
4	**Object get(Object key)**
	It returns the value associated with given key; Returns null if the key does not exist.
5	**Enumeration keys( )**
	Returns an enumeration of the keys contained in the dictionary.
6	**Enumeration elements( )**
	Returns an enumeration of the values contained in the dictionary.
7	**boolean isEmpty( )**
	It returns true if dictionary has no elements; otherwise returns false.
8	**int size( )**
	It returns the total number of elements in the dictionary.

Example

```
import java.util.*;
public class DictionaryExample
{
        public static void main(String args[])
        {
                Dictionary dict = new Hashtable();
                dict.put(1, "Rama");
                dict.put(2, "Seetha");
                dict.put(3, "Heyansh");
                dict.put(4, "Varshith");
                dict.put(5, "Manutej");
                System.out.println("Dictionary\n=> " + dict);
                // keys()
```

```
        System.out.print("\nKeys in Dictionary\n=> ");
        for (Enumeration i = dict.keys(); i.hasMoreElements();)
        {
           System.out.print(" " + i.nextElement());
        }
                    // elements()
        System.out.print("\n\nValues in Dictionary\n=> ");
        for (Enumeration i = dict.elements(); i.hasMoreElements();)
        {
           System.out.print(" " + i.nextElement());
        }
        //get()
                    System.out.println("\n\nValue associated with key 3 => " + dict.get(3));
                    System.out.println("Value associated with key 30 => " + dict.get(30));
                    //size()
                    System.out.println("\nDictionary has " + dict.size() + " elements");
                    //isEmpty()
                    System.out.println("\nIs Dictionary empty?  " + dict.isEmpty());
          }
}
```

Output:

```
Dictionary
=> {5=Manutej, 4=Varshith, 3=Heyansh, 2=Seetha, 1=Rama}

Keys in Dictionary
=>  5 4 3 2 1

Values in Dictionary
=>  Manutej Varshith Heyansh Seetha Rama
Value associated with key 3 => Heyansh
Value associated with key 30 => null
Dictionary has 5 elements
Is Dictionary empty?  false
```

Hashtable class in java

- In java, the package **java.util** contains a class called **Hashtable** which works like a HashMap but it is synchronized. The Hashtable is a concrete class of Dictionary. It is used to store and manage elements in the form of a pair of key and value.

- The Hashtable stores data as a pair of key and value. In the Hashtable, each key associates with a value. Any non-null object can be used as a key or as a value. We can use the key to retrieve the value back when needed.
- The Hashtable class is no longer in use, it is obsolete. The alternate class is HashMap.
- The Hashtable class is a concrete class of Dictionary.
- The Hashtable class is synchronized.
- The Hashtable does not allow null key or value.
- The Hashtable has the initial default capacity 11.

The Hashtable class in java has the following constructors.

S. No.	Constructor with Description
1	**Hashtable( )**
	It creates an empty hashtable with the default initial capacity 11.
2	**Hashtable(int capacity)**
	It creates an empty hashtable with the specified initial capacity.
3	**Hashtable(int capacity, float loadFactor)**
	It creates an empty hashtable with the specified initial capacity and loading factor.
4	**Hashtable(Map m)**
	It creates a hashtable containing elements of Map m.

The Hashtable class in java has the following methods.

S. No.	Methods with Description
1	**void put(K key, V value)**
	It inserts the specified key and value into the hash table.
2	**void putAll(Map m))**
	It inserts all the elements of Map m into the invoking Hashtable.
3	**V get(Object key)**
	It returns the value associated with the given key.
4	**Enumeration keys()**
	Returns an enumeration of the keys of the hashtable.
5	**Set keySet()**
	Returns a set view of the keys of the hashtable.
6	**Collection values()**
	It returns a collection view of the values contained in the Hashtable.
9	**Enumeration elements()**
	Returns an enumeration of the values of the hashtable.
7	**Set entrySet()**
	It returns a set view of the mappings contained in the hashtable.

8	**int hashCode()**
	It returns the hash code of the hashtable.
9	**V remove(Object key)**
	It returns the value associated with given key and removes the same.
10	**boolean remove(Object key, Object value)**
	It removes the specified values with the associated specified keys from the hashtable.
11	**boolean contains(Object value)**
	It returns true if the specified value found within the hash table, else return false.
12	**V replace(K key, V value)**
	It replaces the specified value for a specified key.
13	**boolean replace(K key, V oldValue, V newValue)**
	It replaces the old value with the new value for a specified key.
14	**void rehash()**
	It is used to increase the size of the hash table and rehashes all of its keys.
15	**boolean isEmpty( )**
	It returns true if Hashtable has no elements; otherwise returns false.
16	**int size( )**
	It returns the total number of elements in the Hashtable.
17	**void clear()**
	It is used to remove all the lements of a Hashtable.

Example

```
import java.util.*;
public class HashtableExample
{
        public static void main(String[] args)
        {
                Random num = new Random();
                Hashtable table = new Hashtable();

                //put(key, value)
                for(int i = 1; i <= 5; i++)
                        table.put(i, num.nextInt(100));
                System.out.println("Hashtable => " + table);

                //get(key)
                System.out.println("\nValue associated with key 3 => " + table.get(3));
                System.out.println("Value associated with key 30 => " + table.get(30));
```

```
                //keySet()
                System.out.println("\nKeys => " + table.keySet());

                //values()
                System.out.println("\nValues => " + table.values());

                //entrySet()
                System.out.println("\nKey, Value pairs as a set => " + table.entrySet());
                System.out.println("\nTotal number of elements => " + table.size());

                //isEmpty()
                System.out.println("\nEmpty status of Hashtable => " + table.isEmpty());
        }
}
```

Output:

```
Hashtable => {5=8, 4=3, 3=97, 2=34, 1=67}
Value associated with key 3 => 97
Value associated with key 30 => null
Keys => [5, 4, 3, 2, 1]
Values => [8, 3, 97, 34, 67]
Key, Value pairs as a set => [5=8, 4=3, 3=97, 2=34, 1=67]
Total number of elements => 5
Empty status of Hashtable => false
```

Stack class in java

- In java, the package **java.util** contains a class called **Stack** which is a child class of Vector class. It implements the standard principle Last-In-First-Out of stack data structure.
- The Stack has push method for inesrtion and pop method for deletion. It also has other utility methods.
- In Stack, the elements are added to the top of the stack and removed from the top of the stack.

The Stack class in java has the following constructor.

S. No.	Constructor with Description

S. No.	Constructor with Description
1	**Stack( )**:It creates an empty Stack.

The Stack class in java has the following methods.

S.No.	Methods with Description
1	**Object push(Object element)** It pushes the element onto the stack and returns the same.
2	**Object pop( )** It returns the element on the top of the stack and removes the same.
3	**int search(Object element)** If element found, it returns offset from the top. Otherwise, -1 is returned.
4	**Object peek( )** It returns the element on the top of the stack.
5	**boolean empty()** It returns true if the stack is empty, otherwise returns false.

Example

```
import java.util.*;
public class StackClassExample
{
        public static void main(String[] args)
        {
                Stack stack = new Stack();
                Random num = new Random();
                for(int i = 0; i < 5; i++)
                        stack.push(num.nextInt(100));
                System.out.println("Stack elements => " + stack);
                System.out.println("Top element is " + stack.peek());
                System.out.println("Removed element is " + stack.pop());
                System.out.println("Element 50 availability => " + stack.search(50));
                System.out.println("Stack is empty? - " + stack.isEmpty());
        }
```

}

Output:

Stack elements => [99, 51, 48, 59, 44]
Top element is 44
Removed element is 44
Element 50 availability => -1
Stack is empty? - false

Vector class in java

- In java, the package **java.util** contains a class called **Vector** which implements the **List** interface.
- The Vector is similar to an ArrayList. Like ArrayList Vector also maintains the insertion order. But Vector is synchronized, due to this reason, it is rarly used in the non-thread application. It also lead to poor performance.
- The Vector is a class in the java.util package.
- The Vector implements List interface.
- The Vector is a legacy class.
- The Vector is synchronized.

The Vector class in java has the following constructor.

S. No.	Constructor with Description
1	**Vector( )** It creates an empty Vector with default initail capacity of 10.
2	**Vector(int initialSize)** It creates an empty Vector with specified initail capacity.
3	**Vector(int initialSize, int incr)** It creates a vector whose initial capacity is specified by size and whose increment is specified by incr.
4	**Vector(Collection c)** It creates a vector that contains the elements of collection c.

The Vector class in java has the following methods.

S.No.	Methods with Description

S.No.	Methods with Description
1	**boolean add(Object o)** It appends the specified element to the end of this Vector.
2	**void add(int index, Object element)** It inserts the specified element at the specified position in this Vector.
3	**void addElement(Object obj)** Adds the specified object to the end of the vector, increasing its size by one.
4	**boolean addAll(Collection c)** It appends all of the elements in the specified Collection to the end of the Vector.
5	**boolean addAll(int index, Collection c)** It inserts all of the elements in in the specified Collection into the Vector at the specified position.
6	**Object set(int index, Object element)** It replaces the element at the specified position in the vector with the specified element.
7	**void setElementAt(Object obj, int index)** It sets the element at the specified index of the vector to be the specified object.
8	**Object remove(int index)** It removes the element at the specified position in the vector.
9	**boolean remove(Object o)** It removes the first occurrence of the specified element in the vector.
10	**boolean removeElement(Object obj)** It removes the first occurrence of the specified element in the vector.
11	**void removeElementAt(int index)** It removes the element at specified index in the vector.
12	**void removeRange(int fromIndex, int toIndex)** It removes from the Vector all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.

S.No.	Methods with Description
13	**boolean removeAll(Collection c)** It removes from the vector all of its elements that are contained in the specified Collection.
14	**void removeAllElements()** It removes all the elements from the vector.
15	**boolean retainAll(Collection c)** It removes all the elements from the vector except elements those are in the given collection.
16	**Object elementAt(int index)** It returns the element at specified index in the Vector.
17	**Object get(int index)** It returns the element at specified index in the Vector.
18	**Enumeration elements()** It returns the Enumeration of all the elements of the Vector.
19	**Object firstElement()** It returns the first element of the Vector.
20	**Object lastElement()** It returns the last element of the Vector.
21	**int indexOf(Object element)** It returns the index value of the first occurence of the given element in the Vector.
22	**int indexOf(Object elem, int index)** It returns the index value of the first occurence of the given element, search beginning at specified index in the Vector.
23	**int lastIndexOf(Object elememnt)** It returns the index value of the last occurence of the given element, search beginning at specified index in the Vector.
24	**List subList(int fromIndex, int toIndex)** It returns a list containing elements fromIndex to toIndex in the Vector.

S.No.	Methods with Description
25	**int capacity()** It returns the current capacity of the Vector.
26	**void clear()** It removes all the elements from the Vector.
27	**boolean contains(Object element)** It returns true if element found in the Vector, otherwise returns false.
28	**boolean containsAll(Collection c)** It returns true if all the elements of geven collection found in the Vector, otherwise returns false.
29	**boolean equals(Object o)** It compares the specified Object with this vector for equality.
30	**int hashCode()** It returns the hash code of the Vector.
31	**boolean isEmpty()** It returns true if Vector has no elements, otherwise returns false.
32	**void setSize(int newSize)** It sets the size of the vector.
33	**int size()** It returns total number of elements in the vector.
34	**Object[] toArray()** It returns an array containing all the elements of the Vector.
35	**String toString()** It returns a string representation of the Vector.

Example

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

```
        {
                Vector list = new Vector();
                list.add(10);
                list.add(30);
                list.add(0, 100);
                list.addElement(50);
                System.out.println("Vector => " + list);
                System.out.println("get(2) => " + list.get(2));
                System.out.println("firstElement() => " + list.firstElement());
                System.out.println("indexOf(50) => " + list.indexOf(50));
                System.out.println("contains(30) => " + list.contains(30));
                System.out.println("capacity() => " + list.capacity());
                System.out.println("size() => " + list.size());
                System.out.println("isEmpty() => " + list.isEmpty());
        }
}
```

Output:

```
Vector => [100, 10, 30, 50]
get(2) => 30
firstElement() => 100
indexOf(50) => 3
contains(30) => true
capacity() => 10
size() => 4
isEmpty() => false
```

Date class in java

- The **Date** is a built-in class in java used to work with date and time in java. The Date class is available inside the **java.util** package. The Date class represents the date and time with millisecond precision.
- The Date class implements **Serializable**, **Cloneable** and **Comparable** interface.
- Most of the constructors and methods of Date class has been deprecated after Calendar class introduced.

The Date class in java has the following constructor.

S. No.	Constructor with Description

S. No.	Constructor with Description
1	**Date( )** It creates a Date object that represents current date and time.
2	**Date(long milliseconds)** It creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
3	**Date(int year, int month, int date) -** Depricated It creates a date object with the specified year, month, and date.
4	**Date(int year, int month, int date, int hrs, int min) -** Depricated It creates a date object with the specified year, month, date, hours, and minuts.
5	**Date(int year, int month, int date, int hrs, int min, int sec) -** Depricated It creates a date object with the specified year, month, date, hours, minuts and seconds.
5	**Date(String s) -** Depricated It creates a Date object and initializes it so that it represents the date and time indicated by the string s, which is interpreted as if by the parse(java.lang.String) method.

The Date class in java has the following methods.

S.No.	Methods with Description
1	**long getTime()** It returns the time represented by this date object.
2	**boolean after(Date date)** It returns true, if the invoking date is after the argumented date.
3	**boolean before(Date date)** It returns true, if the invoking date is before the argumented date.
4	**Date from(Instant instant)** It returns an instance of Date object from Instant date.
5	**void setTime(long time)**

S.No.	Methods with Description
	It changes the current date and time to given time.
6	**Object clone( )** It duplicates the invoking Date object.
7	**int compareTo(Date date)** It compares current date with given date.
8	**boolean equals(Date date)** It compares current date with given date for equality.
9	**int hashCode()** It returns the hash code value of the invoking date object.
10	**Instant toInstant()** It converts current date into Instant object.
11	**String toString()** It converts this date into Instant object.

Example

```
import java.time.Instant;
import java.util.Date;
public class DateClassExample
{
        public static void main(String[] args)
        {
                Date time = new Date();
                System.out.println("Current date => " + time);
                System.out.println("Date => " + time.getTime() + " milliseconds");
                System.out.println("after() => " + time.after(time) + " milliseconds");
                System.out.println("before() => " + time.before(time) + " milliseconds");
                System.out.println("hashCode() => " + time.hashCode());
        }
}
```

Output:

```
Current date => Thu Jun 17 04:13:16 GMT 2021
Date => 1623903196155 milliseconds
```

after() => false milliseconds
before() => false milliseconds
hashCode() => 405557889

Java Calendar Class

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

Method	Description
abstract void add(int field, int amount)	It is used to add or subtract the specified amount of time to the given calendar field, based on the calendar's rules.
int get(int field)	It is used to return the value of the given calendar field.
static Calendar getInstance()	It is used to get a calendar using the default time zone and locale.
abstract intgetMaximum(int field)	It is used to return the maximum value for the given calendar field of this Calendar instance.
abstract intgetMinimum(int field)	It is used to return the minimum value for the given calendar field of this Calendar instance.
void set(int field, int value)	It is used to set the given calendar field to the given value.
void setTime(Date date)	It is used to set this Calendar's time with the given Date.
Date getTime()	It is used to return a Date object representing this Calendar's time value.

Example

```
import java.util.Calendar;
public class CalendarExample
{
```

```
public static void main(String args[])
{
        Calendar cal= Calendar.getInstance();
        System.out.println("The current date is : " + cal.getTime());
        cal.add(Calendar.DATE, -15);
        System.out.println("15 days ago: " + cal.getTime());
        cal.add(Calendar.MONTH, 4);
        System.out.println("4 months later: " + cal.getTime());
        cal.add(Calendar.YEAR, 2);
        System.out.println("2 years later: " + cal.getTime());
        }
}
```

Output:

The current date is : Fri Jun 18 07:40:06 GMT 2021
15 days ago: Thu Jun 03 07:40:06 GMT 2021
4 months later: Sun Oct 03 07:40:06 GMT 2021
2 years later: Tue Oct 03 07:40:06 GMT 2023

Random class in java

- The **Random** is a built-in class in java used to generate a stream of pseudo-random numbers in java programming. The Random class is available inside the **java.util** package.
- The Random class implements **Serializable**, **Cloneable** and **Comparable** interface.
- The **Random** class is a part of java.util package.
- The **Random** class provides several methods to generate random numbers of type integer, double, long, float etc.
- The **Random** class is thread-safe.
- Random number generation algorithm works on the seed value. If not provided, seed value is created from system nano time.

The Random class in java has the following constructors.

S.No.	Constructor with Description
1	**Random()** It creates a new random number generator.

S.No.	Constructor with Description
2	**Random(long seedValue)** It creates a new random number generator using a single long seedValue.

The Random class in java has the following methods.

S.No.	Methods with Description
1	**int next(int bits)** It generates the next pseudo-random number.
2	**Boolean nextBoolean()** It generates the next uniformly distributed pseudo-random boolean value.
3	**double nextDouble()** It generates the next pseudo-random double number between 0.0 and 1.0.
4	**void nextBytes(byte[] bytes)** It places the generated random bytes into an user-supplied byte array.
5	**float nextFloat()** It generates the next pseudo-random float number between 0.0 and 1.0..
6	**int nextInt()** It generates the next pseudo-random int number.
7	**int nextInt(int n)** It generates the next pseudo-random integer value between zero and n.
8	**long nextLong()** It generates the next pseudo-random, uniformly distributed long value.
9	**void setSeed(long seedValue)** It sets the seed of the random number generator using a single long seedValue.

Example

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

```
        {
                Random rand = new Random();
                System.out.println("Integer random number - " + rand.nextInt());
                System.out.println("Integer random number from 0 to 100 - " +
rand.nextInt(100));
                System.out.println("Boolean random value - " + rand.nextBoolean());
                System.out.println("Double random number - " + rand.nextDouble());
                System.out.println("Float random number - " + rand.nextFloat());
                System.out.println("Long random number - " + rand.nextLong());

        }
}
```

Output:

```
Integer random number - -449808304
Integer random number from 0 to 100 - 50
Boolean random value - true
Double random number - 0.40273523750011975
Float random number - 0.5990135
Long random number - -265962001106673708
```

StringTokenizer in Java

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class.

Constructor of StringTokenizer class

Constructor	Description
StringTokenizer(String str)	Creates StringTokenizer with specified string.

Methods

The 6 useful methods of StringTokenizer class are as follows:

Public method	Description

Boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
Boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
Int countTokens()	returns the total number of tokens.

Example

Let's see the simple example of StringTokenizer class that tokenizes a string "my name is khan" on the basis of whitespace.

```
import java.util.StringTokenizer;
public class Simple
{
        public static void main(String args[])
        {
                StringTokenizer st = new StringTokenizer("my name is khan"," ");
                while (st.hasMoreTokens())
                {
                    System.out.println(st.nextToken());
                }
        }
}
```

Output:

```
my
name
is
khan
```

Scanner class in java

- The **Scanner** is a built-in class in java used for read the input from the user in java programming. The Scanner class is defined inside the **java.util** package.
- The Scanner class implements **Iterator** interface.

- The Scanner class provides the easiest way to read input in a Java program.
- The Scanner object breaks its input into tokens using a delimiter pattern, the default delimiter is whitespace.

The Scanner class in java has the following constructors.

S.No.	Constructor with Description
1	**Scanner(InputStream source)** It creates a new Scanner that produces values read from the specified input stream.
2	**Scanner(InputStream source, String charsetName)** It creates a new Scanner that produces values read from the specified input stream.
3	**Scanner(File source)** It creates a new Scanner that produces values scanned from the specified file.
4	**Scanner(File source, String charsetName)** It creates a new Scanner that produces values scanned from the specified file.
5	**Scanner(String source)** It creates a new Scanner that produces values scanned from the specified string.
6	**Scanner(Readable source)** It creates a new Scanner that produces values scanned from the specified source.

The Scanner class in java has the following methods.

S.No.	Methods with Description
1	**String next()** It reads the next complete token from the invoking scanner.
2	**String next(Pattern pattern)** It reads the next token if it matches the specified pattern.
3	**String next(String pattern)** It reads the next token if it matches the pattern constructed from the specified string.
4	**boolean nextBoolean()** It reads a boolean value from the user.

S.No.	Methods with Description
5	**byte nextByte()** It reads a byte value from the user.
6	**double nextDouble()** It reads a double value from the user.
7	**float nextFloat()** It reads a floating-point value from the user.
8	**int nextInt()** It reads an integer value from the user.
9	**long nextLong()** It reads a long value from the user.
10	**short nextShort()** It reads a short value from the user.
11	**String nextLine()** It reads a string value from the user.
12	**boolean hasNext()** It returns true if the invoking scanner has another token in its input.
13	**void remove()** It is used when remove operation is not supported by this implementation of Iterator.
14	**void close()** It closes the invoking scanner.

Example

```
import java.util.Scanner;
public class ScannerClassExample
{
        public static void main(String[] args)
        {
                Scanner read = new Scanner(System.in); // Input stream is used
```

```
                System.out.print("Enter any name: ");
                String name = read.next();
                System.out.print("Enter your age in years: ");
                int age = read.nextInt();
                System.out.print("Enter your salary: ");
                double salary = read.nextDouble();
                System.out.print("Enter any message: ");
                read = new Scanner(System.in);
                String msg = read.nextLine();
                System.out.println("\n-----------------------------------------");
                System.out.println("Hello, " + name);
                System.out.println("You are " + age + " years old.");
                System.out.println("You are earning Rs." + salary + " per month.");
                System.out.println("Words from " + name + " - " + msg);
        }
}
```

BitSet class in java

- The BitSet is a built-in class in java used to create a dynamic array of bits represented by boolean values. The BitSet class is available inside the java.util package.
- The BitSet array can increase in size as needed. This feature makes the BitSet similar to a Vector of bits.
- The bit values can be accessed by non-negative integers as an index.
- The size of the array is flexible and can grow to accommodate additional bit as needed.
- The default value of the BitSet is boolean false with a representation as 0 (off).
- BitSet uses 1 bit of memory per each boolean value.

The BitSet class in java has the following constructor.

S. No.	Constructor with Description
1	**BitSet( )** It creates a default BitSet object.
2	**BitSet(int noOfBits)** It creates a BitSet object with number of bits that it can hold. All bits are initialized to zero.

The BitSet class in java has the following methods.

S.No.	Methods with Description
1	**void and(BitSet bitSet)** It performs AND operation on the contents of the invoking BitSet object with those specified by bitSet.
2	**void andNot(BitSet bitSet)** For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3	**void flip(int index)** It reverses the bit at specified index.
4	**void flip(int startIndex, int endIndex)** It reverses all the bits from specified startIndex to endIndex.
5	**void or(BitSet bitSet)** It performs OR operation on the contents of the invoking BitSet object with those specified by bitSet.
6	**void xor(BitSet bitSet)** It performs XOR operation on the contents of the invoking BitSet object with those specified by bitSet.
7	**int cardinality( )** It returns the number of bits set to true in the invoking BitSet.
8	**void clear( )** It sets all the bits to zeros of the invoking BitSet.
9	**void clear(int index)** It set the bit specified by given index to zero.
10	**void clear(int startIndex, int endIndex)** It sets all the bits from specified startIndex to endIndex to zero.
11	**boolean equals(Object bitSet)** It retruns true if both invoking and argumented BitSets are equal, otherwise returns false.
12	**boolean get(int index)**

S.No.	Methods with Description
	It retruns the present state of the bit at given index in the invoking BitSet.
13	**BitSet get(int startIndex, int endIndex)** It returns a BitSet object that consists all the bits from startIndex to endIndex.
14	**int hashCode( )** It returns the hash code of the invoking BitSet.
15	**boolean intersects(BitSet bitSet)** It returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
16	**boolean isEmpty( )** It returns true if all bits in the invoking object are zero, otherwise returns false.
17	**int length( )** It returns the total number of bits in the invoking BitSet.
18	**void set(int index)** It sets the bit specified by index.
19	**void set(int index, boolean value)** It sets the bit specified by index to the value passed in value.
20	**void set(int startIndex, int endIndex)** It sets all the bits from startIndex to endIndex.
21	**void set(int startIndex, int endIndex, boolean value)** It sets all the bits to specified value from startIndex to endIndex.
22	**int size( )** It returns the total number of bits in the invoking BitSet.
23	**String toString( )** It returns the string equivalent of the invoking BitSet object.

Example

```
import java.util.*;
public class BitSetClassExample
```

```
{
	public static void main(String[] args)
	{
		BitSet bSet_1 = new BitSet();
		BitSet bSet_2 = new BitSet(16);
		bSet_1.set(10);
		bSet_1.set(5);
		bSet_1.set(0);
		bSet_1.set(7);
		bSet_1.set(20);
		bSet_2.set(1);
		bSet_2.set(15);
		bSet_2.set(20);
		bSet_2.set(77);
		bSet_2.set(50);
		System.out.println("BitSet_1 => " + bSet_1);
		System.out.println("BitSet_2 => " + bSet_2);
		bSet_1.and(bSet_2);
		System.out.println("BitSet_1 after and with bSet_2 => " + bSet_1);
		bSet_1.andNot(bSet_2);
		System.out.println("BitSet_1 after andNot with bSet_2 => " + bSet_1);
		System.out.println("Length of the bSet_2 => " + bSet_2.length());
		System.out.println("Size of the bSet_2 => " + bSet_2.size());
		System.out.println("Bit at index 2 in bSet_2 => " + bSet_2.get(2));
		bSet_2.set(2);
		System.out.println("Bit at index 2 after set in bSet_2 => " + bSet_2.get(2));
	}
}
```

Output:

```
BitSet_1 => {0, 5, 7, 10, 20}
BitSet_2 => {1, 15, 20, 50, 77}
BitSet_1 after and with bSet_2 => {20}
BitSet_1 after andNot with bSet_2 => {}
Length of the bSet_2 => 78
Size of the bSet_2 => 128
Bit at index 2 in bSet_2 => false
Bit at index 2 after set in bSet_2 => true
```

Formatter class in java

- The **Formatter** is a built-in class in java used for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output in java programming.
- The Formatter class is defined as final class inside the **java.util** package.
- The Formatter class implements **Cloneable** and **Flushable** interface.

The Formatter class in java has the following constructors.

S.No.	Constructor with Description
1	**Formatter()** It creates a new formatter.

The Formatter class in java has the following methods.

S.No.	Methods with Description
1	**Formatter format(Locale l, String format, Object... args)** It writes a formatted string to the invoking object's destination using the specified locale, format string, and arguments.
2	**Formatter format(String format, Object... args)** It writes a formatted string to the invoking object's destination using the specified format string and arguments.
3	**void flush()** It flushes the invoking formatter.
4	**Appendable out()** It returns the destination for the output.
5	**Locale locale()** It returns the locale set by the construction of the invoking formatter.
6	**String toString()** It converts the invoking object to string.
7	**IOException ioException()** It returns the IOException last thrown by the invoking formatter's Appendable.

S.No.	Methods with Description
8	**void close()** It closes the invoking formatter.

Example

```
import java.util.*;
public class FormatterClassExample
{
        public static void main(String[] args)
        {
                Formatter formatter=new Formatter();
                formatter.format("%2$5s %1$5s %3$5s", "Smart", "BTech", "Class");
                System.out.println(formatter);
                formatter = new Formatter();
                formatter.format(Locale.FRANCE,"%.5f", -1325.789);
                System.out.println(formatter);
            String name = "Java";
            formatter = new Formatter();
            formatter.format(Locale.US,"Hello %s !", name);
            System.out.println("" + formatter + " " + formatter.locale());
            formatter = new Formatter();
            formatter.format("%.4f", 123.1234567);
            System.out.println("Decimal floating-point notation to 4 places: " + formatter);
            formatter = new Formatter();
            formatter.format("%010d", 88);
            System.out.println("value in 10 digits: " + formatter);
        }
}
```

Output:

```
BTech Smart Class-
1325,78900
Hello Java ! en_US
Decimal floating-point notation to 4 places: 123.1235
value in 10 digits: 0000000088
```

Properties class in java

- In java, the package **java.util** contains a class called **Properties** which is a child class of Hashtable class. It implements interfaces like Map and Serializable.
- Java has this built-in class Properties which allow us to save and load multiple values from a file. This makes the class extremely useful for accessing data related to configuration.
- The Properties class used to store configuration values managed as key, value pairs. In each pair, both key and value are String values. We can use the key to retrieve the value back when needed.
- The Properties class provides methods to get data from the properties file and store data into the properties file. It can also be used to get the properties of a system.
- The Properties class is child class of Hashtable class.
- The Properties class implements Map, Cloneable, and Serializable interfaces.
- The Properties class used to store configuration values.
- The Properties class stores the data as key, value pairs.
- In Properties class both key and value are String data type.
- Using Properties class, we can load key, value pairs into a Properties object from a stream.
- Using Properties class, we can save the Properties object to a stream.

The Properties class in java has the following constructors.

S. No.	Constructor with Description
1	**Properties( )**
	It creates an empty property list with no default values.
2	**Properties(Properties defaults)**
	It creates an empty property list with the specified defaults.

The Properties class in java has the following methods.

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 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.

Example

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

```
user=system
password=oracle
```

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

```
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
```