UNIT-1

Data Structures in Java

Collections in java are a framework that provides architecture to store and manipulate the group of objects.

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

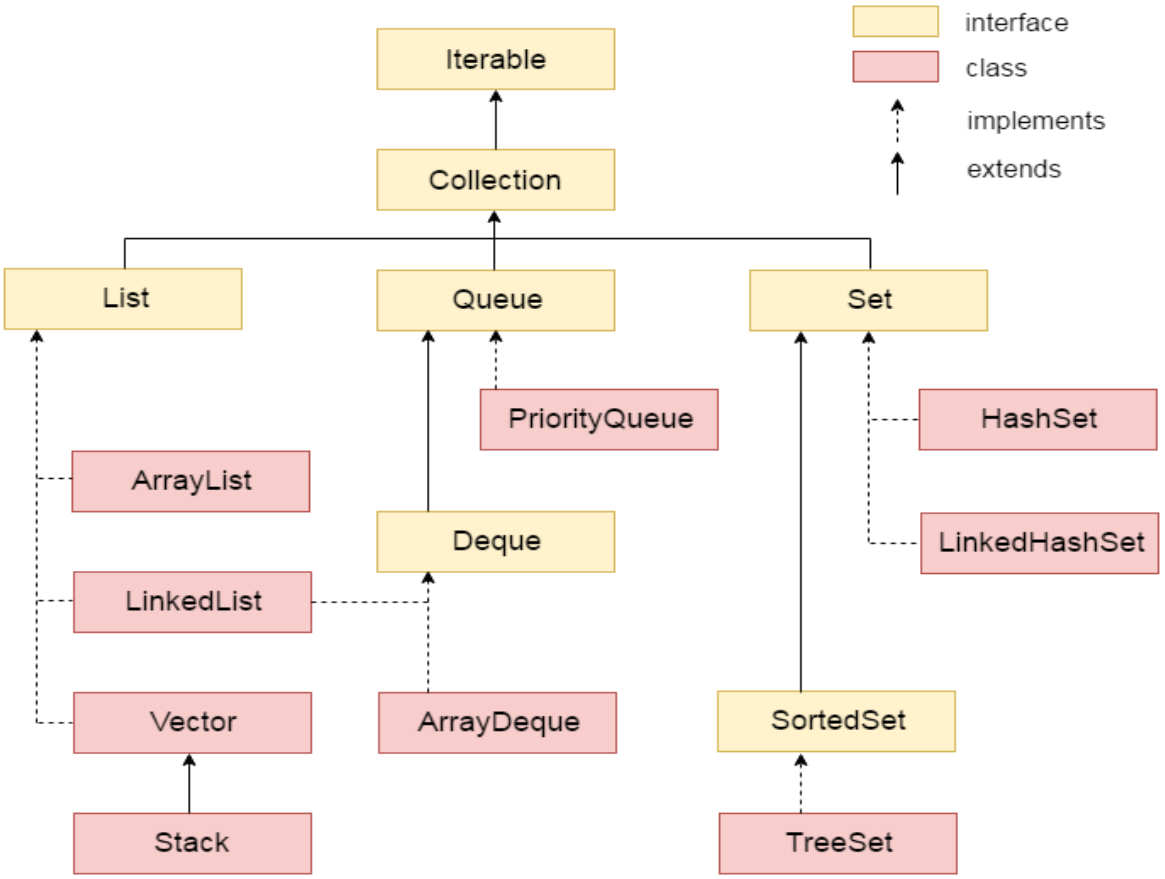
Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

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

1. Interfaces and its implementations i.e. classes
2. Algorithm

Hierarchy of Collection Framework

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



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(Object element)	is used to insert an element in this collection.
2	public boolean addAll(Collection c)	is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	is used to delete an element from this collection.
4	public boolean removeAll(Collection c)	is used to delete all the elements of specified collection from the invoking collection.
5	public boolean retainAll(Collection c)	is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	return the total number of elements in the collection.
7	public void clear()	removes the total no of element from the collection.
8	public boolean contains(Object element)	is used to search an element.
9	public boolean containsAll(Collection c)	is used to search the specified collection in this collection.
10	public Iterator iterator()	returns an iterator.
11	public Object[] toArray()	converts collection into array.
12	public boolean isEmpty()	checks if collection is empty.
13	public boolean equals(Object element)	matches two collection.

14	public int hashCode()	returns the hashcode number for collection.

Iterator interface

Iterator interface provides the facility of iterating the elements in 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 iterator has more elements.
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 rarely used.

Linked List

A linked list is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence. To understand the need for such a data structure, imagine a program that maintains a sequence of employee objects, sorted by the last names of the employees. When a new employee is hired, an object needs to be inserted into the sequence. Unless the company happened to hire employees in alphabetical order, the new object probably needs to be inserted somewhere near the middle of the sequence. If we use an array to store the objects, then all objects following the new hire must be moved toward the end. Conversely, if an employee leaves the company, the object must be removed, and the hole in the sequence needs to be closed up by moving all objects that come after it. Moving a large number of values can involve a substantial amount of processing time. We would like to structure the data in a way that minimizes this cost.

Rather than storing the values in an array, a linked list uses a sequence of nodes. Each node stores a value and a reference to the next node in the sequence (see Figure 1). When you insert a new node into a linked list, only the neighbouring node references need to be updated. The same is true when you remove a node

Instead, the Java library supplies a List Iterator type. A list iterator describes a position anywhere inside the linked list

program:

```
import java.util.*;
public class LinkedListDemo {
public static void main(String args[]) {
// create a linked list
LinkedList ll = new LinkedList();
// add elements to the linked list
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
// remove elements from the linked list
ll.remove("F");
ll.remove(2);

System.out.println("Contents of ll after deletion: " + ll);
// remove first and last elements
ll.removeFirst();
ll.removeLast();

System.out.println("ll after deleting first and last: "+ ll);
// get and set a value
Object val = ll.get(2);
ll.set(2, (String) val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

Output:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

b) Stacks

A stack lets you insert and remove elements at only one end, traditionally called the top of the stack. To visualize a stack, think of a stack of books. New items can be added to the top of the stack. Items are removed at the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, called last in, first out or LIFO order. For example, if you add items A, B, and C and then remove them, you obtain C, B, and A. Traditionally, the addition and removal operations are called push and pop

Program:

```
import java.util.*;
public class StackDemo {
  static void showpush(Stack st, int a) {
    st.push(new Integer(a));
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }
  static void showpop(Stack st) {
    System.out.print("pop -> ");
    Integer a = (Integer) st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }
  public static void main(String args[]) {
    Stack st = new Stack();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
```

```
      showpop(st);
      showpop(st);
      showpop(st);
      try {
        showpop(st);
      }catch (EmptyStackException e) {
        System.out.println("empty stack");
      }
    }
}
```

output:

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

c) Queues

A queue is similar to a stack, except that you add items to one end of the queue (the tail) and remove them from the other end of the queue (the head). To visualize a queue, simply think of

people lining up (see Figure 13). People join the tail of the queue and wait until they have reached the head of the queue. Queues store items in a first in, first out or FIFO fashion. Items are removed in the same order in which they have been added

```
// Java program to demonstrate working of Queue interface in Java
import java.util.LinkedList;
import java.util.Queue;
public class QueueExample
{
  public static void main(String[] args)
  {
    Queue<Integer> q = new LinkedList<>();
    // Adds elements {0, 1, 2, 3, 4} to queue
    for (int i=0; i<5; i++)
     q.add(i);
    // Display contents of the queue.
    System.out.println("Elements of queue-"+q);
     // To remove the head of queue.
    int removedele = q.remove();
    System.out.println("removed element-" + removedele);
    System.out.println(q);
    // To view the head of queue
    int head = q.peek();
System.out.println("head of queue-" + head);
    // Rest all methods of collection interface,
    // Like size and contains can be used with this
    // implementation.
    int size = q.size();
    System.out.println("Size of queue-" + size);
```

```
  }
}
```

Output:

```
Elements of queue-[0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
Size of queue-4
```

d) Set

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ.

```
import java.util.*;
public class SetDemo {
public static void main(String args[]) {
int count[] = {34, 22,10,60,30,22};
Set<Integer> set = new HashSet<Integer>();
try{
for(int i = 0; i<5; i++){
set.add(count[i]);
}
System.out.println(set);
TreeSet sortedSet = new TreeSet<Integer>(set);
System.out.println("The sorted list is:");
System.out.println(sortedSet);
System.out.println("The First element of the set is: "+(Integer)sortedSet.first());
System.out.println("The last element of the set is: "+(Integer)sortedSet.last());
}
```

```
catch(Exception e){}
}
}
```

Output:

```
[34, 22, 10, 60, 30]
The sorted list is:
[10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

e) Map

A map is a data type that keeps associations between keys and values. The figure 1 gives a 18 typical example: a map that associates names with colors. This map might describe the favorite colors of various people.

Mathematically speaking, a map is a function from one set, the key set, to another set, the value set. Every key in the map has a unique value, but a value may be associated with several keys. Just as there are two kinds of set implementations, the Java library has two implementations for maps: HashMap and TreeMap. Both of them implement the Map interface. As with sets, you need to decide which of the two to use. As a rule of thumb, use a hash map unless you want to visit the keys in sorted order.

After constructing a HashMap or TreeMap, you should store the reference to the map object in a Map reference:

```
Map<String, Color> favoriteColors = new HashMap<String, Color>(); or
Map<String, Color> favoriteColors = new TreeMap<String, Color>();
```

Program:

```
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
public class MapDemo
```

```
{

public static void main(String[] args)
{
Map<String, Color> favoriteColors = new HashMap<String, Color>();
favoriteColors.put("sai", Color.BLUE); favoriteColors.put("Ram", Color.GREEN);
favoriteColors.put("krishna", Color.RED);
favoriteColors.put("narayana", Color.BLUE); // Print all keys and values in the
map
Set<String> keySet = favoriteColors.keySet(); for (String key : keySet)
{
Color value = favoriteColors.get(key);
System.out.println(key + " : " + value);
}
}
}
```

Output:

```
narayana : java.awt.Color[r=0,g=0,b=255]
sai : java.awt.Color[r=0,g=0,b=255]
krishna : java.awt.Color[r=255,g=0,b=0]
Ram : java.awt.Color[r=0,g=255,b=0]
```

Generic class

A class that can refer to any type is known as generic class. Here, we are using T type parameter to create the generic class of specific type.

Let’s see the simple example to create and use the generic class.

Creating generic class:

1. class MyGen<T>{
2. T obj;
3. void add(T obj){this.obj=obj;}
4. T get(){return obj;}
5. }

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

Using generic class:

Let’s see the code to use the generic class.

```
1. class TestGenerics3{
2. public static void main(String args[]){
3. MyGen<Integer> m=new MyGen<Integer>();
4. m.add(2);
5. //m.add("vivek");//Compile time error
6. System.out.println(m.get());
7. }}
```

Output:2

Type Parameters

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

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

Generic Method

Like generic class, we can create generic method that can accept any type of argument.

Let’s see a simple example of java generic method to print array elements. We are using here E to denote the element.

```
public class TestGenerics4{
    public static < E > void printArray(E[] elements) {
      for ( E element : elements){
       System.out.println(element );
       }
```

```
            System.out.println();
        }
        public static void main( String args[] ) {
            Integer[] intArray = { 10, 20, 30, 40, 50 };
            Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };
             System.out.println( "Printing Integer Array" );
            printArray( intArray  );
             System.out.println( "Printing Character Array" );
            printArray( charArray );
        }
    }
```

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

Wildcard in Java Generics

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, double etc. Now we can call the method of Number class through any child class object.

Let's understand it by the example given below:

```
import java.util.*;
abstract class Shape{
abstract void draw();
}
```

```
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle");}
}
class GenericTest{
//creating a method that accepts only child class of Shape
public static void drawShapes(List<? extends Shape> lists){
for(Shape s:lists){
s.draw();//calling method of Shape class by child class instance
}
}
public static void main(String args[]){
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());
List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());

drawShapes(list1);
drawShapes(list2);
}}
```

OutPut

drawing rectangle

drawing circle

drawing circle

Wrapper class in Java

Wrapper class in java provides the mechanism *to convert primitive into object and object into primitive.*

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.

The eight classes of *java.lang* package are known as wrapper classes in java. The list of eight wrapper classes are given below:

Primitive Type	Wrapper class
Boolean	Boolean
Char	Character
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	Float
Double	Double

Wrapper class Example: Primitive to Wrapper

```
1. public class WrapperExample1{
2. public static void main(String args[]){
3. //Converting int into Integer
4. int a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
7.
8. System.out.println(a+" "+i+" "+j);
9. }}
```

Output:

20 20 20

Wrapper class Example: Wrapper to Primitive

```
1. public class WrapperExample2{
2. public static void main(String args[]){
3. //Converting Integer to int
4. Integer a=new Integer(3);
5. int i=a.intValue();//converting Integer to int
6. int j=a;//unboxing, now compiler will write a.intValue() internally
7.
8. System.out.println(a+" "+i+" "+j);
9. }}
```

Output:

3 3 3

Serialization in Java

Serialization in java is a mechanism of *writing the state of an object into a byte stream*.

It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called *deserialization*.

Advantage of Java Serialization

It is mainly used to travel object's state on the network (known as marshaling).

java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implements *java.io.Serializable* interface by default.

Let's see the example given below:

```
1. import java.io.Serializable;
2. public class Student implements Serializable{
3.  int id;
4.  String name;
5.  public Student(int id, String name) {
6.   this.id = id;
7.   this.name = name;
8.  }
9. }
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {}creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

Example of Java Serialization

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

```
1. import java.io.*;
2. class Persist{
3.  public static void main(String args[])throws Exception{
4.   Student s1 =new Student(211,"ravi");
5.
6.   FileOutputStream fout=new FileOutputStream("f.txt");
7.   ObjectOutputStream out=new ObjectOutputStream(fout);
8.     out.writeObject(s1);
9.   out.flush();
10.  System.out.println("success");
11.  }
12. }
```

success

Deserialization in java

Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor

1) public ObjectInputStream(InputStream in) throws IOException {}	creates an ObjectInputStream that reads from the specified InputStream.

Important Methods

Method	**Description**
1) public final Object readObject() throws IOException, ClassNotFoundException{}	reads an object from the input stream.

2) public void close() throws IOException {}	closes ObjectInputStream.

Example of Java Deserialization

```
1. import java.io.*;
2. class Depersist{
3.  public static void main(String args[])throws Exception{
4.   ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
5.   Student s=(Student)in.readObject();
6.   System.out.println(s.id+" "+s.name);
7.
8.   in.close();
9.  }
10. }
```

OutPut:

211 ravi