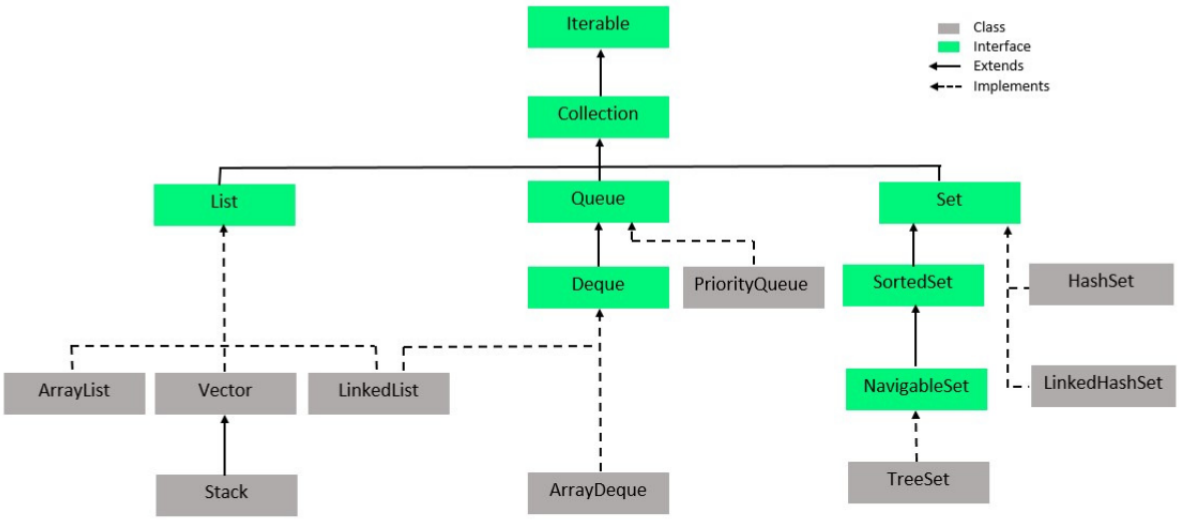
Util Package - Collection Framework

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).



List

List in Java provides the facility to maintain the ***ordered collection***. It contains the **index-based** methods to insert, update, delete and search the elements. It **can have the duplicate element**s also. We can also store the null elements in the list. The List interface is found in the java.util package and inherits the Collection interface.The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming.

ArrayList

Java ArrayList class uses a ***dynamic array*** for storing the elements. It is like an array, but there is *no size limit*. We can add or remove elements anytime. So, it is much more flexible than the traditional array. It is found in the *java.util* package.

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because the array works on an index basis.
- Default capacity 10. Load factor is 0.75

```
import java.util.*;
class ArrayListExample {
    public static void main(String[] args) {
        List<String> names=new ArrayList<String>();

        names.add("arun");
        names.add("bala");
        names.add("Charan");
        names.add("Devi");

        for(String n:names){
            System.out.println(n);
        }

        names.set(0,"Ajay");
        System.out.println(names.get(0));

        names.clear();
        System.out.println(names);
    }
}
```

LinkedList

Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue.

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

  LinkedList<String> train=new LinkedList<String>();
  train.add("carriage 1");
  train.add("carriage 2");
  train.add("carriage 3");
  train.add("carriage 4");

  Iterator<String> itr=train.iterator();
  while(itr.hasNext()){
  // System.out.println(itr.next());
  }

  train.removeFirst();
  System.out.println(train.getFirst());
 }
}
```

Vector

Vector is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

- Vector is synchronized.
- Java Vector contains many legacy methods that are not the part of a collections framework.

Stack

The stack is a linear data structure that is used to store the collection of objects. It is based on **Last-In-First-Out** (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the Stack class that provides different operations such as push, pop, search, etc.

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

  Stack<Integer> stock= new Stack<>();

  stock.push(50);
```

```
  stock.push(100);
  stock.push(70);
  stock.push(90);
  System.out.println("Stack:"+stock );
  System.out.println("Top of the Stack:"+stock.peek() );
  stock.pop();
  System.out.println("Stack:"+stock );
    System.out.println("Top of the Stack:"+stock.peek() );
   }
}
```

Queue

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list

Priority Queue

PriorityQueue is also class that is defined in the collection framework that gives us a way for processing the objects on the basis of priority. It is already described that the insertion and deletion of objects follows FIFO pattern in the Java queue. However, sometimes the elements of the queue are needed to be processed according to the priority, that's where a PriorityQueue comes into action.

```
import java.util.*;
public class QueueExample{
 public static void main(String args[]){
  PriorityQueue<Integer> stock= new PriorityQueue<>();
  stock.add(50);
  stock.add(100);
```

```
  stock.add(70);
  stock.add(90);

  System.out.println("Queue:"+stock );
  System.out.println("Top of the Queue:"+stock.peek() );
  stock.remove();
  System.out.println("Queue:"+stock );

  System.out.println("Top of the Queue:"+stock.peek() );

 }
}
```

Deque

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

ArrayDeque

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

```
import java.util.*;
public class DequeueExample{
 public static void main(String args[]){
  Deque<String> deque=new ArrayDeque<String>();
    deque.offer("arvind");
    deque.offer("vimal");
    deque.add("mukundhan");
    deque.offerFirst("jai");
    System.out.println("After offerFirst Traversal...");
    for(String s:deque){
        System.out.println(s);
    }

    deque.pollLast();
    System.out.println("After pollLast() Traversal...");
    for(String s:deque){
        System.out.println(s);
    }
 }
}
```

Set

The set is an interface available in the java.util package. The set interface extends the Collection interface. An **unordered collection** or list in which **duplicates are not allowed** is referred to as a collection interface. The set interface is used to create the mathematical set. The set interface use collection interface's methods to **avoid the insertion of the same elements**. SortedSet and NavigableSet are two interfaces that extend the set implementation.

HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.

- HashSet doesn't maintain the insertion order. Here, elements are **inserted on the basis of their hashcode.**
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16

```
import java.util.*;
public class SetOperations
{
   public static void main(String args[])
   {
      Integer[] A = {22, 45,33, 66, 55, 34, 77};
      Integer[] B = {33, 2, 83, 45, 3, 12, 55};
      Set<Integer> set1 = new HashSet<Integer>();
      set1.addAll(Arrays.asList(A));
      Set<Integer> set2 = new HashSet<Integer>();
      set2.addAll(Arrays.asList(B));

      Set<Integer> union_data = new HashSet<Integer>(set1);
      union_data.addAll(set2);
      System.out.print("Union of set1 and set2 is:");
      System.out.println(union_data);

      Set<Integer> intersection_data = new HashSet<Integer>(set1);
      intersection_data.retainAll(set2);
      System.out.print("Intersection of set1 and set2 is:");
      System.out.println(intersection_data);

      Set<Integer> difference_data = new HashSet<Integer>(set1);
      difference_data.removeAll(set2);
      System.out.print("Difference of set1 and set2 is:");
      System.out.println(difference_data);
   }
}
```

LinkedHashSet

Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.

The important points about the Java LinkedHashSet class are:

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operations and permits null elements.

- Java LinkedHashSet class is non-synchronized.
- Java LinkedHashSet class maintains insertion order.

```
import java.util.*;
class LinkedHashSetExample{
public static void main(String args[]){
 LinkedHashSet<String> al=new LinkedHashSet<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());
 }
 }
}
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about the Java TreeSet class are:

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and **retrieval times are quiet fast.**
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains **ascending** order.

Map

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.

HashMap

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 contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16

```
import java.util.*;
public class HashMapExample{
public static void main(String args[]){
 HashMap<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"Mango");
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());
```

```
  }
}
}
```

LinkedHashMap

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 contains values based on the key.
- Java LinkedHashMap contains unique elements.
- Java LinkedHashMap may have one null key and multiple null values.
- Java LinkedHashMap is non synchronized.
- Java LinkedHashMap maintains insertion order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

TreeMap

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 contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
- Java TreeMap contains only unique elements.
- Java TreeMap cannot have a null key but can have multiple null values.
- Java TreeMap is non synchronized.
- Java TreeMap maintains ascending order.

Generics

Generics means **parameterized types**. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

Types of Java Generics

Generic Classes:

A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.

```
/*class Employee<T, U>
{
	T empId;
	U salary;

	public Employee(T empId, U salary) {
		this.empId = empId;
		this.salary = salary;
	}

	public void display()
	{
		System.out.println("EmpID :"+this.empId+"\nSalary :"+this.salary);
	}
}*/

class Student<T>
{
	T rollno;

	public Student(T rollno) {

		this.rollno = rollno;
	}
```

```
	public void display()
	{
		System.out.println("Rollno :"+this.rollno);
	}

}


public class GenericClassExample {

	public static void main(String[] args) {

		Student<Integer> s1 = new Student<Integer>(1001);
		s1.display();
	}

}
```

Generic Method: Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

```
class A<T>
{

	 <T> void display(T num)
	 {
		System.out.println("the given value is :"+ num);

	 }
}

public class GenericMethodExample {

	public static void main(String[] args) {

		A obj = new A();
		obj.display(10);

	}
```

```
}
```

Advantages

Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Type casting is not required: There is no need to typecast the object.

Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.