The Java Collections Framework

- Java Collection Framework enables the user to perform various data manipulation operations like storing data, searching, sorting, insertion, deletion, and updating of data on the group of elements.
- It provides the implementations for the data structures like dynamic array, stack, queue, linked list, tree, and has table
- Collection frameworks have Collection classes, interfaces, and algorithms
- One more important aspect of Collections Framework is the iterator. It offers a way of accessing the elements within a collection, one by one.

The Collection Interfaces

- The collection interfaces are available in the package **java.util**

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue. This can be used to implement a stack or a queue.
List	Extends **Collection** to handle lists of objects.
NavigableSet	Extends **SortedSet** to handle the retrieval of an element based on how close its value is to another value.
Queue	Extends **Collection** to handle queues.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

- Collection is a generic interface and the syntax is

interface Collection<E>

E specifies the type of objects that the list will hold

Methods of Collection interface

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

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Otherwise, returns **false**.
boolean addAll(Collection<? extends E> *col*)	Adds the elements in *col* to the invoking collection. Returns **true** if at least one element was added. Otherwise, returns **false**.
void clear( )	Empties the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is in the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *col*)	Returns **true** if all elements of *col* are also in the invoking collection. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the invoking collection's hash code.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.

Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one item equal to *obj* from the invoking collection. Returns **true** if successful. Returns **false** if *obj* was not in the collection.
boolean removeAll(Collection<?> *col*)	Removes all elements in *col* from the invoking collection. Returns **true** if at least one item was removed. Otherwise, returns **false**.
boolean retainAll(Collection<?> *col*)	Removes all elements from the invoking collection except those in *col*. Returns **true** if at least one element was removed. Otherwise, returns **false**.
int size( )	Returns the number of elements currently stored in the invoking collection.
Object[ ] toArray( )	Returns the elements stored in the invoking collection in the form of an array. The array is independent from the invoking collection.
<T> T[ ] toArray(T[ ] *array*)	Returns the elements stored in the invoking collection in the form of an array. The array is independent from the invoking collection. If the size of *array* equals or exceeds the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned.

The List Interface

- The List interface extends Collection and declares the behaviour of a collection that stores a sequence of elements
- The syntax of List interface is

interface List<E>

E specifies the type of objects that the list will hold

Method	Description
void **add**(int *index*, E *obj*)	Adds *obj* to the invoking list at the index passed in *index*.
boolean addAll(int *index*, Collection<? extends E> *col*)	Adds the elements of *col* to the invoking list at the index passed in *index*. Returns **true** if at least one element was added and **false** otherwise.
E get(int *index*)	Returns the object stored at the index passed in *index* within the invoking collection.
int **indexOf**(Object *obj*)	Searches for the first instance of *obj* in the invoking list. If found, its index is returned. Otherwise, –1 is returned.
int **lastIndexOf**(Object *obj*)	Searches for the last instance of *obj* in the invoking list. If found, its index is returned. Otherwise, –1 is returned.
ListIterator<E> **listIterator**( )	Returns an iterator to the start of the invoking list.
ListIterator<E> **listIterator**(int *index*)	Returns an iterator to the invoking list that begins at the index passed in *index*.
E **remove**(int *index*)	Removes the element at the index passed in *index* from the invoking list and returns the deleted element.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
List<E> **subList**(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. The resulting list is a view of the invoking list.

The Set Interface

- The Set interface defines a set.
- The syntax to declare one is

interface Set<E>

E specifies the type of objects that the set will hold.

The SortedSet Interface

- It extends Set and declares the behaviour of a set sorted in ascending order
- The syntax to declare is

interface SortedSet<E>

Method	Description
Comparator<? super E> comparator( )	Returns the invoking set's comparator. If no comparator is used, **null** is returned.
E first( )	Returns the element with the smallest value. Because the set is sorted, this will be the first element in the invoking set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** that includes those elements in the invoking set that are less than *end*. The resulting set is a view of the invoking set.
E last( )	Returns the element with the largest value. Because the set is sorted, this will be the last element in the invoking set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements in the invoking set that are between *start* and *end*–1. The resulting set is a view of the invoking set.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that includes those elements in the invoking set that are greater than or equal to *start*. The resulting set is a view of the invoking set.

The NavigableSet Interface

- It extends SortedSet and declares the behaviour of a collection that supports the retrieval of an element
- The syntax to declare one is

interface NavigableSet<E>

E specifies the type of objects that the set will hold.

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that contains the entries in the invoking set in reverse order. The resulting set is a view of the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is a view of the invoking set.
E higher(E *obj*)	Searches the set for the smallest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E pollFirst( )	Returns the first element in the invoking set, removing the element in the process. Because the set is sorted, this is the element with the smallest value. Returns **null** if called on an empty set.
E pollLast( )	Returns the last element in the invoking set, removing the element in the process. Because the set is sorted, this is the element with the largest value. Returns **null** if called on an empty set.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is a view of the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is a view of the invoking set.

The Queue Interface

- It extends Collection and declares the behaviour of a queue. Its syntax is

interface Queue<E>

E specifies the type of objects that the queue will hold.

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. If called on an empty queue, a **NoSuchElementException** is thrown.
boolean offer (E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise. This method will fail if an attempt is made to add an element to a full capacity-restricted queue.
E peek( )	Returns the element at the head of the queue. The element is not removed. Returns **null** if called on an empty queue.
E poll( )	Returns the element at the head of the queue, removing the element in the process. Returns **null** if called on an empty queue.
E remove( )	Removes the element at the head of the queue, returning the element in the process. If called on an empty queue, a **NoSuchElementException** is thrown.

The Deque Interface

- It extends Queue and declares the behaviour of a double-ended queue.
- Its syntax is

interface Deque<E>

Method	Description
void addFirst(E *obj*)	Attempts to add *obj* to the head of the deque. It throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Attempts to add *obj* to the tail of the deque. It throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the element at the head of the deque. The object is not removed. If called on an empty deque, a **NoSuchElementException** is thrown.
E getLast( )	Returns the element at the tail of the deque. The object is not removed. If called on an empty deque, a **NoSuchElementException** is thrown.
boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
E peekFirst( )	Returns the element at the head of the deque. The object is not removed. Returns **null** if called on an empty deque.
E peekLast( )	Returns the element at the tail of the deque. The object is not removed. Returns **null** if called on an empty deque.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. Returns **null** if called on an empty deque.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if called on an emtpy deque.
E pop( )	Returns the element at the head of the deque, removing it in the process. If called on an empty deque, a **NoSuchElementException** is thrown.
void push(E *obj*)	Adds *obj* to the head of the deque. It throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. If called on an empty deque, a **NoSuchElementException** is thrown.
boolean removeFirstOccurrence (Object *obj*)	Searches for the first instance of *obj*. If found, the element is removed. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. If called on an empty deque, a **NoSuchElementException** is thrown.
boolean emoveLastOccurrence (Object *obj*)	Searches for the last instance of *obj*. If found, the element is removed. Returns **true** if successful and **false** if the deque did not contain *obj*.

The Collection Classes

- The collection interfaces are implemented by different classes

Class	Description
AbstractCollection	Implements parts of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements parts of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection designed for sequential rather than random access of its elements.
AbstractSet	Extends **AbstractCollection** and implements parts of the **Set** interface.

ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection**. It also implements the **Deque** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Implements a set stored in a hash table by extending **AbstractSet**.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
LinkedList	Implements a linked list by extending **AbstractSequentialList**. Also implements **Deque**.
PriorityQueue	Implements a priority-based queue by extending **AbstractQueue**.
TreeSet	Implements a set stored in a tree by extending **AbstractSet**. Also implements **SortedSet**.

The ArrayList Class

- It extends AbstractList and implements the List interface
- It has the following declaration

class ArrayList<E>

- ArrayList supports dynamic arrays that can grow as needed whereas standard arrays are of fixed length
- ArrayList has the following constructors
  1. ArrayList( )
     - It creates an empty array list
  2. ArrayList(Collection<? extends E> col)
     - It builds an array list that is initialized with the elements of the collection col
  3. ArrayList(int initCapacity)
     - It builds an array list that has the initial capacity passed to initCapacity

The LinkedList Class

- It extends AbstractSequentialList and implements the List and Deque interfaces
- It provides a doubly linked list data structure
- The syntax goes like this

  class LinkedList<E>
- It has two constructors
  1. LinkedList( )
     - It creates an empty list
  2. LinkedList(Collection<? extends E> col)
     - It builds a linked list that is initialized with the elements of the collection col
- Since it implements Deque interface, all the methods defined by Deque can be accessed by linked list class

The HashSet Class

- It extends AbstractSet and implements the Set interface
- It creates a collection that uses a hash table for storage
- Its syntax is
- It has the following constructors
  1. HashSet( )
     - Creates an empty hash set
  2. HashSet(Collection<? extends E> col)
     - Creates a hash set that is initialized with the elements of the collection col
  3. HashSet(int initCapacity)
     - It builds a has set that has the initial capacity passed to initCapacity
  4. HashSet(int initCapacity, float fillRatio)
     - It initializes both initial capacity and fill ratio of the hash set from its arguments

The TreeSet Class

- It extends AbstractSet and implements the NavigableSet interface
- Creates a collection that uses form of balanced, sorted binary tree for storage
- Objects are stored in sorted, ascending order
- Its syntax is

  class TreeSet<E>
- It has the following constructors
  1. TreeSet( )

- Creates an empty tree set that will be sorted in ascending order

2. TreeSet(Collection<? extends E> col)
   - Creates a tree set that is initialized with the elements of the collection col
3. TreeSet(Comparator<? super E> comp)
   - Creates an empty tree set that will be sorted according to the comparator specified by comp
4. TreeSet(SortedSet<E> ss)
   - Builds a tree set that contains the elements of ss

The LinkedHashSet Class

- It extends HashSet and does not add any members on its own
- The syntax is

  class LinkedHashSet<E>
- It has the same constructors as HashSet
- It also uses hash table to store elements

The ArrayDeque Class

- It extends AbstractCollection and implements the Deque interface
- Its syntax is

  class ArrayDeque<E>
- Its constructors are
  1. ArrayDeque( )
     - Creates an empty deque
  2. ArrayDeque(Collection<? extends E> col)
     - It builds a deque that is initialized with the elements of the collection col
  3. ArrayDeque(int initCapacity)
     - It builds a deque that has the initial capacity passed to initCapacity

The PriorityQueue Class

- It extends AbstractQueue and implements the Queue interface
- It creates a sorted queue, with the sort order indicating the priority
- Its syntax is

  class PriorityQueue<E>
- It has the following constructors
  1. PriorityQueue( )
     - Creates an empty queue

2. PriorityDeque(int initCapacity)
   - It builds a queue that has the initial capacity passed to initCapacity
3. PriorityQueue(int initCapacity, Comparator<? super E> comp)
   - It creates a queue with the specified initial capacity and comparator

Accessing A Collection via an Iterator

- Iterator enables to cycle through a collection, obtaining or removing elements
- Its syntax is

interface Iterator<E>

Table 25.7: *The methods declared by Iterator*

Method	Description
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
E next( )	Returns the next element. If there are no more elements, **NoSuchElementException** is thrown.
void remove( )	Removes the element returned by **next( )** from the collection being iterated.

- ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements

interface ListIterator<E>

Table 25.8: *The methods declared by ListIterator*

Method	Description
void add(E *obj*)	Adds *obj* to the collection in front of the element that will be returned by the next call to **next( )**.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**. This method is used when moving forward.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**. This method is used when moving backwards.
E next( )	Returns the next element in the forward direction. If there are no more elements in that direction, **NoSuchElementException** is thrown.
int nextIndex( )	Returns the index of the next element in the forward direction. If there is not a next element, the size of the collection being iterated is returned.
E previous( )	Returns the previous element, which is the next element in the backwards direction. If there are no more elements in that direction, **NoSuchElementException** is thrown.
int previousIndex( )	Returns the index of the next element in the backward direction. If there is not a previous element, –1 is returned.
void remove( )	Removes the element returned by **next( )** or **previous( )** from the collection being iterated.
void set(E *obj*)	Assigns *obj* to the currently iterated element. This is the element last returned by a call to either **next( )** or **previous( )**. This change affects the collection being iterated.

Using and Iterator

- Each one of the collection classes provides an iterator( ) method that returns an iterator to the start of the collection
- Steps to use an iterator to cycle through the contents of a collection,
  1. Obtain an itearator to the start of the collection by calling the collection's iterator() method.
  2. Set up a loop that makes a call to hasNext( ). Iterate as long as hasNext( ) returns true
  3. Within the loop, obtain each element by calling next( )

Example of Iterator and ListIterator

```
import java.util.*;

public class IteratorDemo {
    public static void main(String[] args) {
        ArrayList<String> al = new ArrayList<>();
        al.add("Alpha");
        al.add("Beta");
        al.add("Gamma");
        al.add("Delta");
        al.add("Epsilon");
        al.add("Zeta");
        al.add("Eta");

        System.out.println("Original Contents");
        Iterator<String> i = al.iterator();
        while(i.hasNext())
            System.out.print(i.next() + " ");
        System.out.println("\n");
        //to remove Gamma from the list
        i = al.iterator();
        while(i.hasNext()){
            if(i.next().equals("Gamma"))
                i.remove();
        }
        System.out.println("Contents after deletion");
        i=al.iterator();
        while(i.hasNext())
            System.out.print(i.next() + " ");
        System.out.println("\n");

        //to add Gamma back to the list
        ListIterator<String> li = al.listIterator();
        while(li.hasNext()){
            if(li.next().equals("Beta"))
            li.add("Gamma");
        }
        System.out.println("Contents after addition ");
        li = al.listIterator();
        while(li.hasNext())
            System.out.print(li.next() + " ");
        System.out.println("\n");
        //to modify the objects
        String str;
        li = al.listIterator();
```

```
while(li.hasNext()){
    str = li.next();
    switch (str) {
        case "Eta":
            li.set("Omega");
            break;
        case "Zeta":
            li.set("Psi");
            break;
        case "Epsilon":
            li.set("Chi");
            break;
        case "Delta":
            li.set("Dlt");
            break;
        default:
            break;
    }
}
System.out.println("Contents after changes ");
li= al.listIterator();
while(li.hasNext())
    System.out.print(li.next() + " ");
System.out.println("\n");

//ListIterator to display the backwards
System.out.println("Modified list backwards ");
while(li.hasPrevious()){
    System.out.print(li.previous() + " ");
}
System.out.println("\n");
System.out.println("Array elements before sorting ");
for(String s:al)
    System.out.print(s+" ");
System.out.println("\n");
System.out.println("Array elements after sorting in ascending order ");
Collections.sort(al);
```

```
for(String s:al)
    System.out.print(s+" ");
System.out.println("\n");
System.out.println("Array elements after sorting in descending order ");
Collections.sort(al,Collections.reverseOrder());
for(String s:al)
    System.out.print(s+" ");
System.out.println("\n");

Scanner sc = new Scanner(System.in);
System.out.println("Enter the element to be searched in the list: ");
String key = sc.nextLine();
for(String s:al){
    if(s.equals(key)){
        System.out.println("Element found");
        return;
    }
}
```

```
            System.out.println("Element not found");
        }

}
```

OUTPUT:

```
Original Contents
Alpha Beta Gamma Delta Epsilon Zeta Eta

Contents after deletion
Alpha Beta Delta Epsilon Zeta Eta

Contents after addition
Alpha Beta Gamma Delta Epsilon Zeta Eta

Contents after changes
Alpha Beta Gamma Dlt Chi Psi Omega

Modified list backwards
Omega Psi Chi Dlt Gamma Beta Alpha

Array elements before sorting
Alpha Beta Gamma Dlt Chi Psi Omega

Array elements after sorting in ascending order
Alpha Beta Chi Dlt Gamma Omega Psi

Array elements after sorting in descending order
Psi Omega Gamma Dlt Chi Beta Alpha

Enter the element to be searched in the list:
Psi
Element found
```

Java Lambda Expressions

- Lambda expression provides a clear and concise way to represent one method interface using an expression.
- It is very useful in collection library. It helps to iterate, filter and extract data from collection.
- It is used to provide the implementation of an interface which has functional interface

Functional Interface

- Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface.
- Java provides an annotation @FunctionalInterface, which is used to declare an interface as functional interface.

Java Lambda Expression Syntax

(argument-list) -> {body}

Types of Lambda Expression:

- There are 3 types of lambda expressions

1. **Zero or No Parameter**

```
() -> {

//Body of no parameter lambda

}
```

```
@FunctionalInterface
interface Drawable{
    void draw();
}

public class Lambda {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

OUTPUT:

```
Drawing 10
```

2. **One Parameter**

(p1) -> {

//Body of single parameter lambda

}

```
@FunctionalInterface
interface Circle{
    void area(int r);
}

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

        //with lambda
        Circle c1=(radius)->{
            double area =3.142*radius*radius;
            System.out.println("The area is "+area);
        };
        c1.area(4);
    }
}
```

OUTPUT:

```
The area is 50.272
```

3. **Multiple Parameters**

(p1, p2) -> {

//Body of multiple parameter lambda

}

Example:

```
@FunctionalInterface
interface Addition{
     int add(int a,int b);
}

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

        //with lambda
        Addition a1=(a,b)->(a+b);
        System.out.println("The result from a1 is "+a1.add(2,3));

        Addition a2=(a,b)->{
            return (a+b);
        };
        System.out.println("The result from a2 is "+a2.add(3,5));
    }
}
```

OUTPUT:

```
The result from a1 is 5
The result from a2 is 8
```