**Collections In Java**

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that we perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (Array List, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

#### What is a framework in Java?

#### It provides readymade architecture.

#### It represents a set of classes and interfaces.

#### It is optional.

#### What is Collection framework?

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

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

### **Methods of Collection interface**

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

|  |  |  |
| --- | --- | --- |
| No. | Method | Description |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

### **Iterator Interface**

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

#### **Methods of Iterator interface**

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

1.public boolean hasNext():-It returns true if the iterator has more elements otherwise it returns false.

2.public Object next():-It returns the element and moves the cursor pointer to the next element.

3.public void remove():-It removes the last elements returned by the iterator. It is less used.

## I**terable Interface**

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

It contains only one abstract method. i.e.,Iterator<T>iterator():-it returns the iterator over the elements of T.

**Collection Interface**

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

To instantiate the List interface, we must use :

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

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

The classes that implement the List interface are given below.

## **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.

## **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.

## **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.

## **Stack**

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

## **Queue Interface**

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

Queue<String> q1 = new PriorityQueue();

Queue<String> q2 = new ArrayDeque();

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

## **PriorityQueue**

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

## **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 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.

## **Set Interface**

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

Set can be instantiated as:

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

## **HashSet**

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

## **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.

## **SortedSet Interface**

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

The SortedSet can be instantiated as: 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.

**Java 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.

The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here. The ArrayList maintains the insertion order internally.It inherits the AbstractList class and implements List Interface

The important note about Java ArrayList class are:

* 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 array works at the index basis.
* In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

**ArrayList class declaration**

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, Serializable

### **Constructors of ArrayList**

ArrayList() :-It is used to build an empty array list.

ArrayList(Collection<? extends E> c):-It is used to build an array list that is initialized with the elements of the collection c.

ArrayList(int capacity):-It is used to build an array list that has the specified initial capacity.

**Methods of ArrayList**

|  |  |
| --- | --- |
| Method | Description |
| void [add](https://www.javatpoint.com/java-arraylist-add-method)(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean add (E e) | It is used to append the specified element at the end of a list. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void clear() | It is used to remove all of the elements from this list. |
| void ensureCapacity(int requiredCapacity) | It is used to enhance the capacity of an ArrayList instance. |
| E get(int index) | It is used to fetch the element from the particular position of the list. |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |
| Iterator() |  |
| listIterator() |  |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| <T> T[] toArray(T[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| boolean contains(Object o) | It returns true if the list contains the specified element |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |
| boolean removeObject o) | It is used to remove the first occurrence of the specified element. |
| boolean removeAll(Collection<?> c) | It is used to remove all the elements from the list. |
| boolean removeIf(Predicate<? super E> filter) | It is used to remove all the elements from the list that satisfies the given predicate. |
| protected void removeRange(int fromIndex, int toIndex) | It is used to remove all the elements lies within the given range. |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |
| void retainAll(Collection<?> c) | It is used to retain all the elements in the list that are present in the specified collection. |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the specified position. |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of specified comparator. |
| Spliterator<E> spliterator() | It is used to create spliterator over the elements in a list. |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements lies within the given range. |
| int size() | It is used to return the number of elements present in the list. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

**Java 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.

The important note about Java LinkedList are:

* 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.

**LinkedList class declaration**

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>,Deque<E>,Cloneable,Serializable

**Constructors of LinkedList**

LinkedList():-It is used to construct an empty list.

LinkedList(Collection<? extends E> c):-It is used to construct a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator.

**Methods in LinkedList**

|  |  |
| --- | --- |
| Method | Description |
| boolean add(E e) | It is used to append the specified element to the end of a list. |
| void add(int index, E element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void addFirst(E e) | It is used to insert the given element at the beginning of a list. |
| void addLast(E e) | It is used to append the given element to the end of a list. |
| void clear() | It is used to remove all the elements from a list. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| boolean contains(Object o) | It is used to return true if a list contains a specified element. |
| Iterator<E> descendingIterator() | It is used to return an iterator over the elements in a deque in reverse sequential order. |
| E element() | It is used to retrieve the first element of a list. |
| E get(int index) | It is used to return the element at the specified position in a list. |
| E getFirst() | It is used to return the first element in a list. |
| E getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element. |
| ListIterator<E> listIterator(int index) | It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list. |
| boolean offer(E e) | It adds the specified element as the last element of a list. |
| boolean offerFirst(E e) | It inserts the specified element at the front of a list. |
| boolean offerLast(E e) | It inserts the specified element at the end of a list. |
| E peek() | It retrieves the first element of a list |
| E peekFirst() | It retrieves the first element of a list or returns null if a list is empty. |
| E peekLast() | It retrieves the last element of a list or returns null if a list is empty. |
| E poll() | It retrieves and removes the first element of a list. |
| E pollFirst() | It retrieves and removes the first element of a list, or returns null if a list is empty. |
| E pollLast() | It retrieves and removes the last element of a list, or returns null if a list is empty. |
| E pop() | It pops an element from the stack represented by a list. |
| void push(E e) | It pushes an element onto the stack represented by a list. |
| E remove() | It is used to retrieve and removes the first element of a list. |
| E remove(int index) | It is used to remove the element at the specified position in a list. |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element in a list. |
| E removeFirst() | It removes and returns the first element from a list. |
| boolean removeFirstOccurrence(Object o) | It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail). |
| E removeLast() | It removes and returns the last element from a list. |
| boolean removeLastOccurrence(Object o) | It removes the last occurrence of the specified element in a list (when traversing the list from head to tail). |
| E set(int index, E element) | It replaces the element at the specified position in a list with the specified element. |
| Object[] toArray() | It is used to return an array containing all the elements in a list in proper sequence (from first to the last element). |
| <T> T[] toArray(T[] a) | It returns an array containing all the elements in the proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array. |
| int size() | It is used to return the number of elements in a list. |

# **Java 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 elements 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. It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions. The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector. The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

**List Interface declaration**

public interface List<E> extends Collection<E>

**Methods in List**

|  |  |  |
| --- | --- | --- |
| Method | Description |  |
| void add(int index, E element) | It is used to insert the specified element at the specified position in a list. |  |
| boolean add(E e) | It is used to append the specified element at the end of a list. |  |
| boolean addAll(Collection<? extends E> c) | It is used to append all of the elements in the specified collection to the end of a list. |  |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |  |
| void clear() | It is used to remove all of the elements from this list. |  |
| boolean equals(Object o) | It is used to compare the specified object with the elements of a list. |  |
| int hashcode() | It is used to return the hash code value for a list. |  |
| E get(int index) | It is used to fetch the element from the particular position of the list. |  |
| boolean isEmpty() | It returns true if the list is empty, otherwise false. |  |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |  |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |  |
| <T> T[] toArray(T[] a) | It is used to return an array containing all of the elements in this list in the correct order. |  |
| boolean contains(Object o) | It returns true if the list contains the specified element |  |
| boolean containsAll(Collection<?> c) | It returns true if the list contains all the specified element |  |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |  |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |  |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element. |  |
| boolean removeAll(Collection<?> c) | It is used to remove all the elements from the list. |  |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |  |
| void retainAll(Collection<?> c) | It is used to retain all the elements in the list that are present in the specified collection. |  |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the specified position. |  |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of specified comparator. |  |
| Spliterator<E> spliterator() | It is used to create spliterator over the elements in a list. |  |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements lies within the given range. |  |
| int size() | It is used to return the number of elements present in the list. |  |

# **Java 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 note 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, and the load factor is 0.75.

**HashSet Class declaration**

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

**Constructors of HashSet class**

HashSet():-It is used to construct a default HashSet.

HashSet(int capacity):-It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.

HashSet(int capacity, float loadFactor):-It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.

HashSet(Collection<? extends E> c):-It is used to initialize the hash set by using the elements of the collection c.

**Methods in HashSet**

|  |  |  |  |
| --- | --- | --- | --- |
| SN | Modifier & Type | Method | Description |
| 1) | boolean | [add(E e)](https://www.javatpoint.com/java-hashset-add-method) | It is used to add the specified element to this set if it is not already present. |
| 2) | void | [clear()](https://www.javatpoint.com/java-hashset-clear-method) | It is used to remove all of the elements from the set. |
| 3) | object | [clone()](https://www.javatpoint.com/java-hashset-clone-method) | It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| 4) | boolean | [contains(Object o)](https://www.javatpoint.com/java-hashset-contains-method) | It is used to return true if this set contains the specified element. |
| 5) | boolean | [isEmpty()](https://www.javatpoint.com/java-hashset-isempty-method) | It is used to return true if this set contains no elements. |
| 6) | Iterator<E> | i[terator()](https://www.javatpoint.com/java-hashset-iterator-method) | It is used to return an iterator over the elements in this set. |
| 7) | boolean | [remove(Object o)](https://www.javatpoint.com/java-hashset-remove-method) | It is used to remove the specified element from this set if it is present. |
| 8) | int | [size()](https://www.javatpoint.com/java-hashset-size-method) | It is used to return the number of elements in the set. |
| 9) | Spliterator<E> | [spliterator()](https://www.javatpoint.com/java-hashset-spliterator-method) | It is used to create a late-binding and fail-fast Spliterator over the elements in the set. |

**Java LinkedHashSet class**

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

The important notes about Java LinkedHashSet class are:

* Java LinkedHashSet class contains unique elements only like HashSet.
* Java LinkedHashSet class provides all optional set operation and permits null elements.
* Java LinkedHashSet class is non synchronized.
* Java LinkedHashSet class maintains insertion order.

**LinkedHashSet class declaration**

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable

**Constructors of LinkedHashSet class**

HashSet():-It is used to construct a default HashSet.

HashSet(Collection c):-It is used to initialize the hash set by using the elements of the collection c.

LinkedHashSet(int capacity):-It is used initialize the capacity of the linked hash set to the given integer value capacity.

LinkedHashSet(int capacity, float fillRatio):-It is used to initialize both the capacity and the fill ratio (also called load capacity) of the hash set from its argument.

**Java TreeSet Class**

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 note about 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.

### **TreeSet class declaration**

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable

### **Constructors of Java TreeSet class**

TreeSet():-It is used to construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.

TreeSet(Collection<? extends E> c):-It is used to build a new tree set that contains the elements of the collection c.

TreeSet(Comparator<? super E> comparator):-It is used to construct an empty tree set that will be sorted according to given comparator.

TreeSet(SortedSet<E> s):-It is used to build a TreeSet that contains the elements of the given SortedSet.

**Methods in TreeSet class**

|  |  |
| --- | --- |
| Method | Description |
| boolean add(E e) | It is used to add the specified element to this set if it is not already present. |
| boolean addAll(Collection<? extends E> c) | It is used to add all of the elements in the specified collection to this set. |
| E ceiling(E e) | It returns the equal or closest greatest element of the specified element from the set, or null there is no such element. |
| Comparator<? super E> comparator() | It returns comparator that arranged elements in order. |
| Iterator | It is used iterate the elements in descending order. |
| NavigableSet | It returns the elements in reverse order. |
| E floor(E e) | It returns the equal or closest least element of the specified element from the set, or null there is no such element. |
| SortedSet | It returns the group of elements that are less than the specified element. |
| NavigableSet | It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element. |
| E higher(E e) | It returns the closest greatest element of the specified element from the set, or null there is no such element. |
| Iterator | It is used to iterate the elements in ascending order. |
| E lower(E e) | It returns the closest least element of the specified element from the set, or null there is no such element. |
| E pollFirst() | It is used to retrieve and remove the lowest(first) element. |
| E pollLast() | It is used to retrieve and remove the highest(last) element. |
| Spliterator | It is used to create a late-binding and fail-fast spliterator over the elements. |
| NavigableSet | It returns a set of elements that lie between the given range. |
| SortedSet | It returns a set of elements that lie between the given range which includes fromElement and excludes toElement. |
| SortedSet | It returns a set of elements that are greater than or equal to the specified element. |
| NavigableSet | It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element. |
| boolean contains(Object o) | It returns true if this set contains the specified element. |
| boolean isEmpty() | It returns true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It returns a shallow copy of this TreeSet instance. |
| E first() | It returns the first (lowest) element currently in this sorted set. |
| E last() | It returns the last (highest) element currently in this sorted set. |
| int size() | It returns the number of elements in this set. |

**Java Queue Interface**

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

### **Queue Interface declaration**

public interface Queue<E> extends Collection<E>

**Methods in Queue Interface**

|  |  |
| --- | --- |
| Method | Description |
| boolean add(object) | It is used to insert the specified element into this queue and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and removes the head of this queue. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

# **Java Deque Interface**

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

## **Deque Interface declaration**

public interface Deque<E> extends Queue<E>

**Methods in DeQue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean add(object) | It is used to insert the specified element into this deque and return true upon success. |
| boolean offer(object) | It is used to insert the specified element into this deque. |
| Object remove() | It is used to retrieves and removes the head of this deque. |
| Object poll() | It is used to retrieves and removes the head of this deque, or returns null if this deque is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this deque. |
| Object peek() | It is used to retrieves, but does not remove, the head of this deque, or returns null if this deque is empty. |

## **ArrayDeque class**

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.

### ArrayDeque class declaration

public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable

# **Java Map Interface**

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

A Map is useful for search, update or delete elements on the basis of a key.

**Methods in Map interface**

|  |  |
| --- | --- |
| Method | Description |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| Set keySet() | It returns the Set view containing all the keys. |
| Set<Map.Entry<K,V>> entrySet() | It returns the Set view containing all the keys and values. |
| void clear() | It is used to reset the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| int hashCode() | It returns the hash code value for the Map |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

## **Map.Entry Interface**

Entry is the subinterface of Map. So we will be accessed it by Map.Entry name. It returns a collection-view of the map, whose elements are of this class. It provides methods to get key and value.

### **Methods in Map.Entry interface**

|  |  |
| --- | --- |
| Method | Description |
| K getKey() | It is used to obtain a key. |
| V getValue() | It is used to obtain value. |
| int hashCode() | It is used to obtain hashCode. |
| V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with the other existing objects. |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey() | It returns a comparator that compare the objects in natural order on key. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp) | It returns a comparator that compare the objects by key using the given Comparator. |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp) | It returns a comparator that compare the objects by value using the given Comparator. |

**Java 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.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

### **Note**

* 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 with a load factor of 0.75.

### **HashMap class declaration**

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable

### **Constructors of Java HashMap class**

HashMap():-It is used to construct a default HashMap.

HashMap(Map<? extends K,? extends V> m):-It is used to initialize the hash map by using the elements of the given Map object m.

HashMap(int capacity):-It is used to initializes the capacity of the hash map to the given integer value, capacity.

HashMap(int capacity, float loadFactor):-It is used to initialize both the capacity and load factor of the hash map by using its arguments.

**Methods in HashMap class**

|  |  |
| --- | --- |
| Method | Description |
| void clear() | It is used to remove all of the mappings from this map. |
| boolean isEmpty() | It is used to return true if this map contains no key-value mappings. |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| V remove(Object key) | It is used to delete an entry for the specified key. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| boolean equals(Object o) | It is used to compare the specified Object with the Map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| boolean isEmpty() | This method returns true if the map is empty; returns false if it contains at least one key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection<V> values() | It returns a collection view of the values contained in the map. |
| int size() | This method returns the number of entries in the map. |

**Java LinkedHashMap class**

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

### Note

* 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.

### LinkedHashMap class declaration

public class LinkedHashMap<K,V> extends HashMap<K,V> implement Map<K,V>

### Constructors of Java LinkedHashMap class

LinkedHashMap():-It is used to construct a default LinkedHashMap.

LinkedHashMap(int capacity):-It is used to initialize a LinkedHashMap with the given capacity.

LinkedHashMap(int capacity, float loadFactor):-It is used to initialize both the capacity and the load factor.

LinkedHashMap(int capacity, float loadFactor, boolean accessOrder):-It is used to initialize both the capacity and the load factor with specified ordering mode.

LinkedHashMap(Map<? extends K,? extends V> m):-It is used to initialize the LinkedHashMap with the elements from the given Map class m.

**Methods in LinkedHashMap**

|  |  |
| --- | --- |
| Method | Description |
| V get(Object key) | It returns the value to which the specified key is mapped. |
| void clear() | It removes all the key-value pairs from a map. |
| boolean containsValue(Object value) | It returns true if the map maps one or more keys to the specified value. |
| Set<Map.Entry<K,V>> entrySet() | It returns a Set view of the mappings contained in the map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key. |
| Set<K> keySet() | It returns a Set view of the keys contained in the map |
| protected boolean removeEldestEntry(Map.Entry<K,V> eldest) | It returns true on removing its eldest entry. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection<V> values() | It returns a Collection view of the values contained in this map. |

**Java TreeMap class**

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

The important note about Java TreeMap class are:

* 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.

### TreeMap class declaration

public class TreeMap<K,V> extends AbstractMap<K,V> implements NavigableMap<K,V>, Cloneable, Serializable

### Constructors of TreeMap class

TreeMap():-It is used to construct an empty tree map that will be sorted using the natural order of its key.

TreeMap(Comparator<? super K> comparator):-It is used to construct an empty tree-based map that will be sorted using the comparator comp.

TreeMap(Map<? extends K,? extends V> m):-It is used to initialize a treemap with the entries from m, which will be sorted using the natural order of the keys.

TreeMap(SortedMap<K,? extends V> m):-It is used to initialize a treemap with the entries from the SortedMap sm, which will be sorted in the same order as sm.

**Methods in TreeMap class**

|  |  |
| --- | --- |
| Method | Description |
| Map.Entry<K,V> ceilingEntry(K key) | It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key. |
| K ceilingKey(K key) | It returns the least key, greater than the specified key or null if there is no such key. |
| void clear() | It removes all the key-value pairs from a map. |
| Object clone() | It returns a shallow copy of TreeMap instance. |
| Comparator<? super K> comparator() | It returns the comparator that arranges the key in order, or null if the map uses the natural ordering. |
| NavigableSet<K> descendingKeySet() | It returns a reverse order NavigableSet view of the keys contained in the map. |
| NavigableMap<K,V> descendingMap() | It returns the specified key-value pairs in descending order. |
| Map.Entry | It returns the key-value pair having the least key. |
| Map.Entry<K,V> floorEntry(K key) | It returns the greatest key, less than or equal to the specified key, or null if there is no such key. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| SortedMap<K,V> headMap(K toKey) | It returns the key-value pairs whose keys are strictly less than toKey. |
| NavigableMap<K,V> headMap(K toKey, boolean inclusive) | It returns the key-value pairs whose keys are less than (or equal to if inclusive is true) toKey. |
| Map.Entry<K,V> higherEntry(K key) | It returns the least key strictly greater than the given key, or null if there is no such key. |
| K higherKey(K key) | It is used to return true if this map contains a mapping for the specified key. |
| Set | It returns the collection of keys exist in the map. |
| Map.Entry<K,V> lastEntry() | It returns the key-value pair having the greatest key, or null if there is no such key. |
| Map.Entry<K,V> lowerEntry(K key) | It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key. |
| K lowerKey(K key) | It returns the greatest key strictly less than the given key, or null if there is no such key. |
| NavigableSet<K> navigableKeySet() | It returns a NavigableSet view of the keys contained in this map. |
| Map.Entry<K,V> pollFirstEntry() | It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty. |
| Map.Entry<K,V> pollLastEntry() | It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty. |
| V put(K key, V value) | It inserts the specified value with the specified key in the map. |
| void putAll(Map<? extends K,? extends V> map) | It is used to copy all the key-value pair from one map to another map. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) | It returns key-value pairs whose keys range from fromKey to toKey. |
| SortedMap<K,V> subMap(K fromKey, K toKey) | It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive. |
| SortedMap<K,V> tailMap(K fromKey) | It returns key-value pairs whose keys are greater than or equal to fromKey. |
| NavigableMap<K,V> tailMap(K fromKey, boolean inclusive) | It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey. |
| boolean containsKey(Object key) | It returns true if the map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It returns true if the map maps one or more keys to the specified value. |
| K firstKey() | It is used to return the first (lowest) key currently in this sorted map. |
| V get(Object key) | It is used to return the value to which the map maps the specified key. |
| K lastKey() | It is used to return the last (highest) key currently in the sorted map. |
| V remove(Object key) | It removes the key-value pair of the specified key from the map. |
| Set<Map.Entry<K,V>> entrySet() | It returns a set view of the mappings contained in the map. |
| int size() | It returns the number of key-value pairs exists in the hashtable. |
| Collection | It returns a collection view of the values contained in the map. |

**Java Lambda Expression**

Lambda expression is a new and important feature of Java which was included in Java SE 8. It 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.

The Lambda expression 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 anotation @FunctionalInterface, which is used to declare an interface as functional interface.

## Why use Lambda Expression

1.To provide the implementation of functional interface

2.less coding.

## **Java Lambda Expression Syntax**

(argument-list)->{body}

Java lambda expression consists of three components.

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

No parameter syntax

()->{

//body of the no parameter lambda

}

One parameter syntax

(p1)->{

//body of one parameter lambda

}

Two parameter syntax

(p1,p2)->{

//body of multiple parameter lambda

}

**Stream API**

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements.

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example filter, map and sorted are intermediate operations whereas forEach is a terminal operation.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both non-interfering and stateless.

A function is non-interfering when it does not modify the underlying data source of the stream.

A function is stateless when the execution of the operation is deterministic.

**Diffrent kind of Streams**

Streams can be created from various data sources, especially collections. Lists and Sets support new methods stream() and parallelStream() to either create a sequential or a parallel stream. Parallel streams are capable of operating on multiple threads.

**Methods in Stream API**

|  |  |
| --- | --- |
| Methods | Description |
| boolean allMatch(Predicate<? super T> predicate) | It returns all elements of this stream which match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| boolean anyMatch(Predicate<? super T> predicate) | It returns any element of this stream that matches the provided predicate. If the stream is empty then false is returned and the predicate is not evaluated. |
| static <T> Stream.Builder<T> builder() | It returns a builder for a Stream. |
| <R,A> R collect(Collector<? super T,A,R> collector) | It performs a mutable reduction operation on the elements of this stream using a Collector. A Collector encapsulates the functions used as arguments to collect(Supplier, BiConsumer, BiConsumer), allowing for reuse of collection strategies and composition of collect operations such as multiple-level grouping or partitioning. |
| <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner) | It performs a mutable reduction operation on the elements of this stream. A mutable reduction is one in which the reduced value is a mutable result container, such as an ArrayList, and elements are incorporated by updating the state of the result rather than by replacing the result. |
| static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b) | It creates a lazily concatenated stream whose elements are all the elements of the first stream followed by all the elements of the second stream. The resulting stream is ordered if both of the input streams are ordered, and parallel if either of the input streams is parallel. When the resulting stream is closed, the close handlers for both input streams are invoked. |
| long count() | It returns the count of elements in this stream. This is a special case of a reduction. |
| Stream<T> distinct() | It returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream. |
| static <T> Stream<T> empty() | It returns an empty sequential Stream. |
| Stream<T> filter(Predicate<? super T> predicate) | It returns a stream consisting of the elements of this stream that match the given predicate. |
| Optional<T> findAny() | It returns an Optional describing some element of the stream, or an empty Optional if the stream is empty. |
| Optional<T> findFirst() | It returns an Optional describing the first element of this stream, or an empty Optional if the stream is empty. If the stream has no encounter order, then any element may be returned. |
| <R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper) | It returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| DoubleStream flatMapToDouble(Function<? super T,? extends DoubleStream> mapper) | It returns a DoubleStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have placed been into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| IntStream flatMapToInt(Function<? super T,? extends IntStream> mapper) | It returns an IntStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| LongStream flatMapToLong(Function<? super T,? extends LongStream> mapper) | It returns a LongStream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) |
| void forEach(Consumer<? super T> action) | It performs an action for each element of this stream. |
| void forEachOrdered(Consumer<? super T> action) | It performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order. |
| static <T> Stream<T> generate(Supplier<T> s) | It returns an infinite sequential unordered stream where each element is generated by the provided Supplier. This is suitable for generating constant streams, streams of random elements, etc. |
| static <T> Stream<T> iterate(T seed,UnaryOperator<T> f) | It returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc. |
| Stream<T> limit(long maxSize) | It returns a stream consisting of the elements of this stream, truncated to be no longer than maxSize in length. |
| <R> Stream<R> map(Function<? super T,? extends R> mapper) | It returns a stream consisting of the results of applying the given function to the elements of this stream. |
| DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper) | It returns a DoubleStream consisting of the results of applying the given function to the elements of this stream. |
| IntStream mapToInt(ToIntFunction<? super T> mapper) | It returns an IntStream consisting of the results of applying the given function to the elements of this stream. |
| LongStream mapToLong(ToLongFunction<? super T> mapper) | It returns a LongStream consisting of the results of applying the given function to the elements of this stream. |
| Optional<T> max(Comparator<? super T> comparator) | It returns the maximum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| Optional<T> min(Comparator<? super T> comparator) | It returns the minimum element of this stream according to the provided Comparator. This is a special case of a reduction. |
| boolean noneMatch(Predicate<? super T> predicate) | It returns elements of this stream match the provided predicate. If the stream is empty then true is returned and the predicate is not evaluated. |
| @SafeVarargs static <T> Stream<T> of(T... values) | It returns a sequential ordered stream whose elements are the specified values. |
| static <T> Stream<T> of(T t) | It returns a sequential Stream containing a single element. |
| Stream<T> peek(Consumer<? super T> action) | It returns a stream consisting of the elements of this stream, additionally performing the provided action on each element as elements are consumed from the resulting stream. |
| Optional<T> reduce(BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using an associative accumulation function, and returns an Optional describing the reduced value, if any. |
| T reduce(T identity, BinaryOperator<T> accumulator) | It performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value. |
| <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) | It performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. |
| Stream<T> skip(long n) | It returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream. If this stream contains fewer than n elements then an empty stream will be returned. |
| Stream<T> sorted() | It returns a stream consisting of the elements of this stream, sorted according to natural order. If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed. |
| Stream<T> sorted(Comparator<? super T> comparator) | It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator. |
| Object[] toArray() | It returns an array containing the elements of this stream. |
| <A> A[] toArray(IntFunction<A[]> generator) | It returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing. |