Programming in Java

Topic: Collections

Contents

Introduction

- A collection is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- *java.util* package contains all the collection classes and interfaces.

Collection Framework

- A collections framework is a unified architecture for representing and manipulating collections.
- It is a collection of classes and interfaces.
- At the top of collection framework hierarchy, there is an interface, *Collection*.

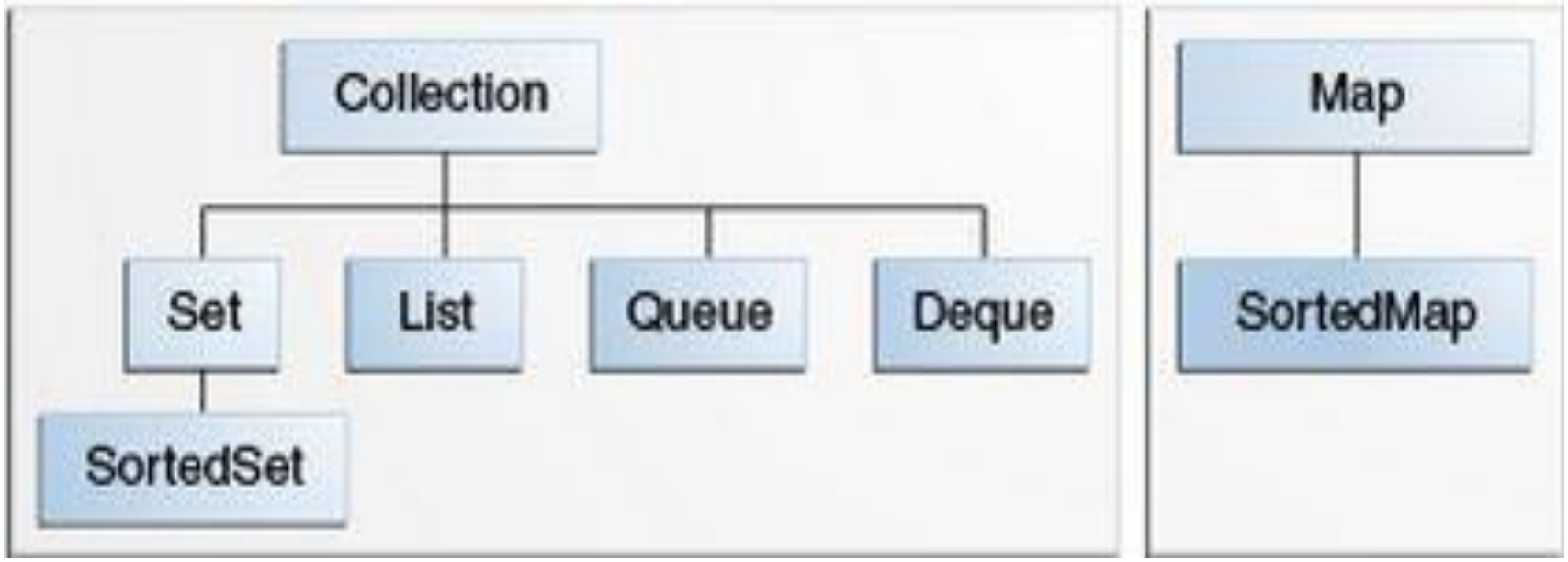
Advantages of Collections

- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs

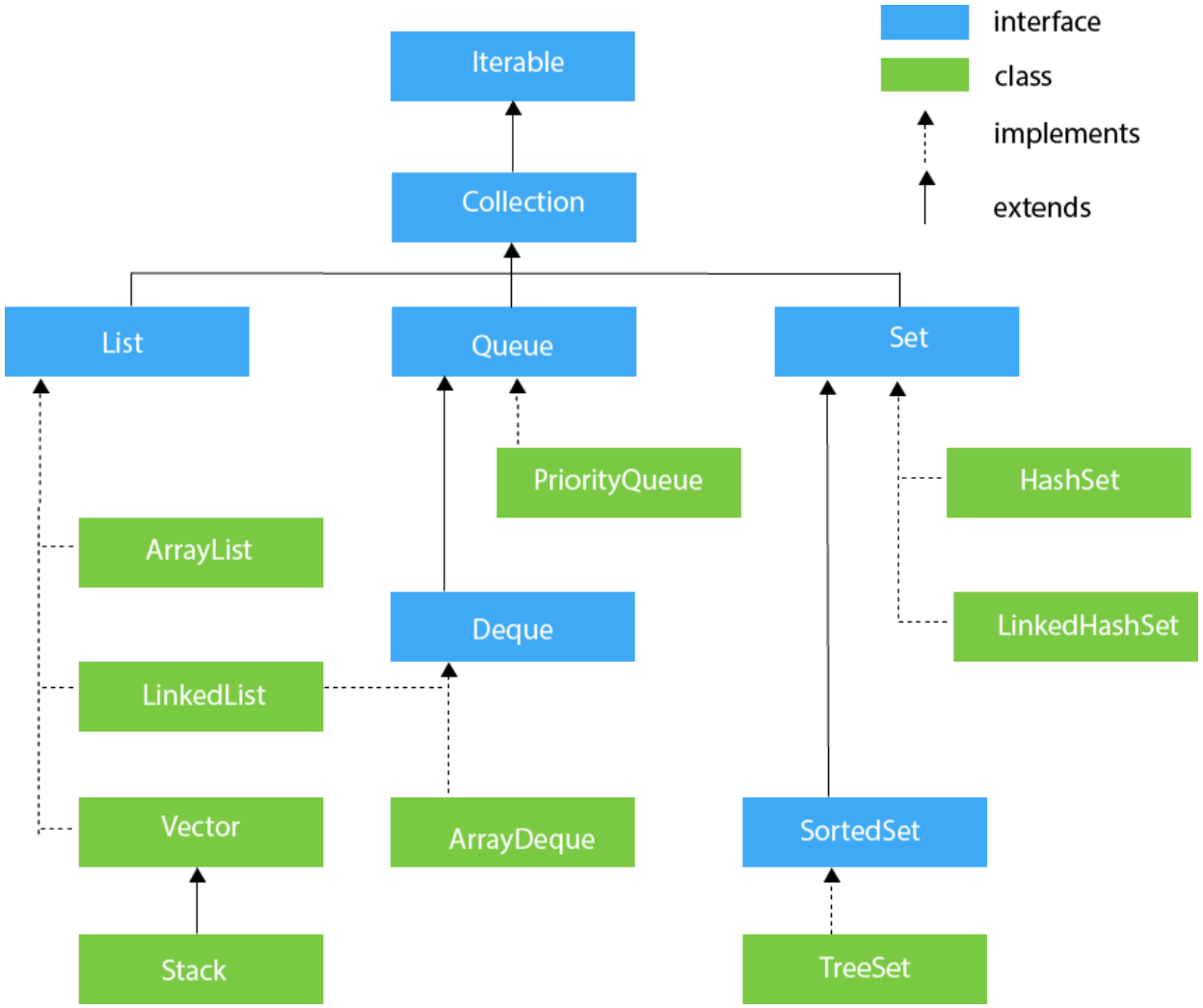
Collection Interfaces

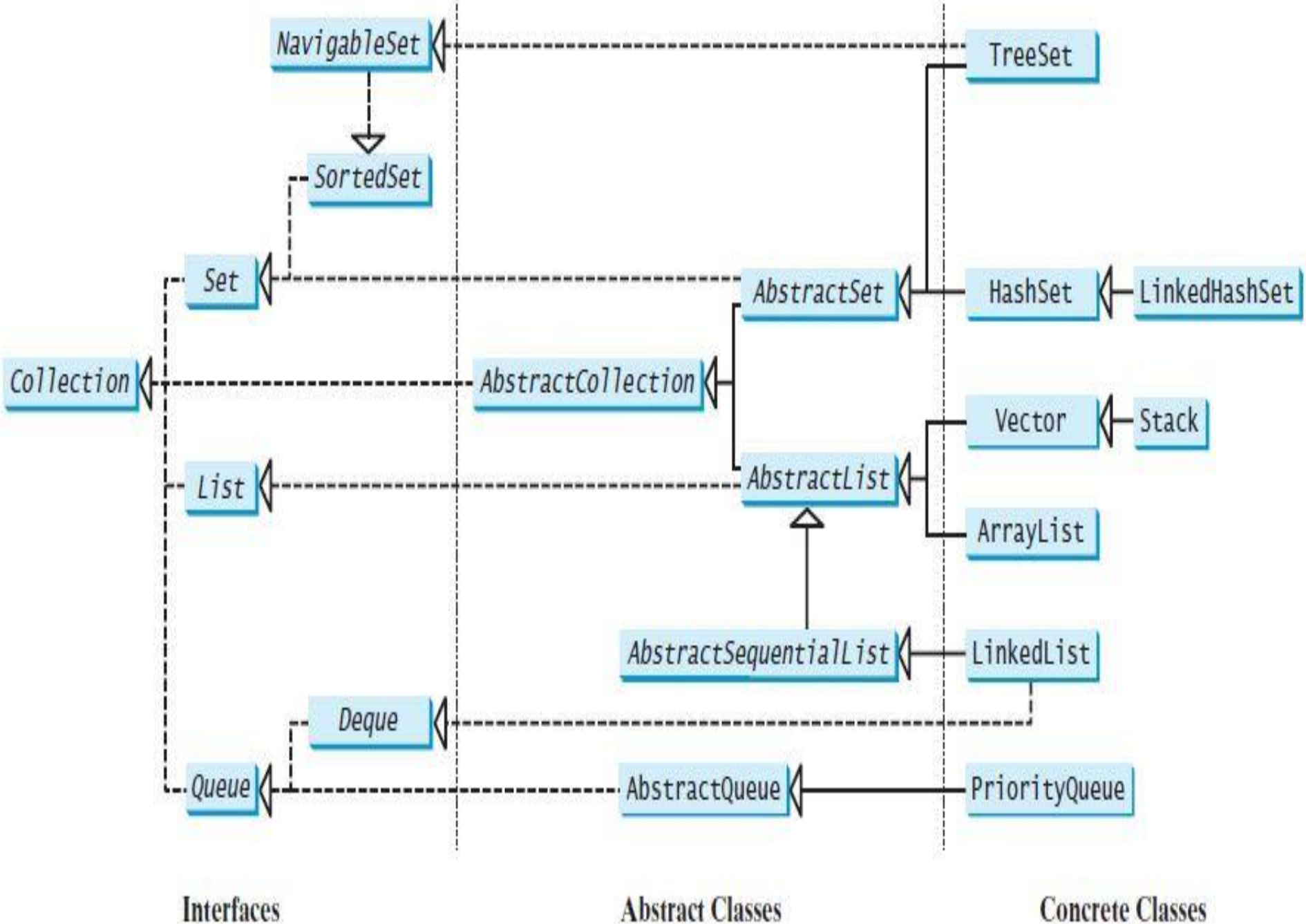
The Java Collections Framework supports two types of containers:

1. Collection: for storing a collection of elements
2. Map: for storing key/value pairs



Hierarchy of Collections Framework




NavigableSet
TreeSet
SortedSet
Set
AbstractSet
HashSet
LinkedHashSet
Collection
AbstractCollection
Vector
Stack
List
AbstractList
ArrayList
AbstractSequentialList
LinkedList
Deque
Queue
AbstractQueue
PriorityQueue
Interfaces
Abstract Classes
Concrete Classes

Collection Interfaces

- Sets store a group of non-duplicate elements.
- Lists store an ordered collection of elements.
- Queues store objects that are processed in first-in, first-out fashion.
- Maps are efficient data structures for quickly searching an element using a key.

- The *AbstractCollection* class provides partial implementation for the Collection interface.

- It implements all the methods in Collection except the *size()* and *iterator()* methods.

COLLECTION

INTERFACE

Methods of Collection Interface

boolean add(Object o)

- Appends the specified element o to the end of the collection.

boolean addAll(Collection c)

- Appends all of the elements in the specified collection to the end of the collection.

boolean contains(Object o)

- Returns true if this list contains the specified element.

boolean containsAll(Collection c)

- Returns true if this collection contains all the elements in c.

boolean remove(Object o)

- Removes the object from this collection.

boolean removeAll(Collection c)

- Removes all the elements in c from this collection.

boolean retainAll(Collection c)

- Retains the elements that are both in c and in this collection.

void clear()

- Removes all of the elements from the collection.

boolean isEmpty()

- Returns true if this collection contains no elements.

int size()

- Returns the number of elements in the collection.

object [] toArray()

- Returns an array of Object for the elements in the collection.

Iterator iterator()

- Returns the iterator object for the invoking collection.
  - I

ITERATORS

Iterator

- There is an Iterable interface at the top of Collection hierarchy.
- The Collection interface extends the java.lang.Iterable interface.
- It defines a method which returns an Iterator object for the elements of Collection.

Iterator iterator()

- *Iterator* object is used to traverse the elements in a collection.

Methods of Iterator Interface

boolean hasNext()

Returns true if this iterator has more elements to traverse.

Object next()

Returns the next element from this iterator.

void remove()

Removes the last element obtained using the next method.

ListIterator

ListIterator

- ListIterator is derived from Iterator interface and comes with more methods than Iterator.
- ListIterator can be used to traverse for List type Objects.
- Allows to traverse in both the directions.

ListIterator

- A ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

- The remove() and set(Object) methods are not defined in terms of the cursor position; they are defined to operate on the last element returned by a call to next() or previous().

ListIterator Methods

- public abstract boolean hasPrevious()
- public abstract Object previous();

- public abstract int nextIndex();
- public abstract int previousIndex();

- public abstract void set(Object);
- public abstract void add(Object);

LIST INTERFACE

List

- The List interface *extends the Collection* interface.
- It defines a collection for storing elements in a sequential order.
- It define an *ordered collection* with *duplicates allowed*.
- *ArrayList* , *Vector* and *LinkedList* are the sub-classes of List interface.

Methods of List Interface

- boolean add(int index, Object o)
- boolean addAll(int index, Collection c)
- Object remove(int index)
- Object get(int index)
- Object set(int index, Object o)
- int indexOf(Object o)
- int lastIndexOf(Object o)
- ListIterator listIterator()
- ListIterator listIterator(int start_index)
- List subList(int start_index, int end_index): returns a sublist from start_index to end_index-1

Sub-classes of List

- ArrayList
- LinkedList
- Vector
- Stack

ARRAYLIST

ArrayList

- The ArrayList class extends *AbstractList* and implements the *List* interface.
- Defined in *java.util* package.
- ArrayList supports dynamic arrays that can grow as needed.
- Array lists are created with an initial size.
- When this size is exceeded, the collection is automatically enlarged.
- When objects are removed, the array may be shrunk.

ArrayList Constructors

- T he ArrayList class supports three constructors.

- ArrayList() : creates an empty array list with an initial capacity sufficient to hold 10 elements.

- ArrayList(int capacity) : creates an array list that has the specified initial capacity.

- ArrayList(Collection c) : creates an array list that is initialized with the elements of the collection c.

Methods of ArrayList

boolean add(Object o)

- Appends the specified element to the end of this list.

void add(int index, Object element)

- Inserts the specified element at the specified position index in this list.
- Throws IndexOutOfBoundsException if the specified index is out of range.

boolean addAll(Collection c)

- Appends all of the elements in the specified collection to the end of this list.
- Throws NullPointerException if the specified collection is null.

boolean addAll(int index, Collection c)

- Inserts all of the elements of the specified collection into this list, starting at the specified position.
- Throws NullPointerException if the specified collection is null.

Object remove(int index)

- Removes the element at the specified position in this list.

Object remove(Object o)

- Removes the element o from this list.

Object clone()

- Returns a shallow copy of this ArrayList.

int size()

- Returns the number of elements in the list.

void clear()

- Removes all of the elements from this list.

boolean contains(Object o)

- Returns true if this list contains the specified element.

Object get(int index)

- Returns the element at the specified position in this list.

int indexOf(Object o)

- Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain the element.

int lastIndexOf(Object o)

- Returns the index in this list of the last occurrence of the specified element, or -1.

void ensureCapacity(int minCapacity)

- Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

Object set(int index, Object element)

- Replaces the element at the specified position in this list with the specified element.
- Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).

void trimToSize()

- Trims the capacity of this ArrayList instance to be the list's current size.

SET INTERFACE

Set Interface

- A set is an efficient data structure for storing and processing *non-duplicate elements*.
- Set does not introduce new methods or constants.
- The sub-classes of Set (HashSet, TreeSet & LinkedHashSet) ensure that no duplicate elements can be added to the set.

Set Interface

- The *AbstractSet* class provides implementations for the *equals()* method and the *hashCode()*.
- The hash code of a set is the sum of the hash codes of all the elements in the set.
- The *size()* and *iterator()* are not implemented in the AbstractSet class.

TREESET

TreeSet

- TreeSet implements the NavigableSet interface.
- We can add objects into a tree set as long as they can be compared with each other.
- Access and retrieval times are quite fast.
- TreeSet implements the SortedSet interface. So, duplicate values are not allowed.
- Objects in a TreeSet are stored in a sorted and ascending order.
- TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
- Null values are not accepted by TreeSet

TreeSet

- Constructor:

  TreeSet()

  TreeSet(Collection c)

  TreeSet(Comparator c)

Implementing TreeSet[Example 1]

```
import java.util.*;
public class Main {
public static void main(String args[]){
Set<Integer> tree = new TreeSet<Integer>();
tree.add(3);
tree.add(1);
tree.add(2);
System.out.println("treeSet : "+tree);
}
}
```

Output:

treeSet: [1,2,3]

Example 2

CODERINDEED

```
import java.util.*;
public class Main{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Example 3

CODERINDEED

```
import java.util.*;
public class Main{
 public static void main(String args[]){
 TreeSet<String> set=new TreeSet<String>();
        set.add("Ravi");
        set.add("Vijay");
        set.add("Ajay");
        System.out.println("Traversing element through Iterator in descending order");
        Iterator i=set.descendingIterator();
        while(i.hasNext())
        {
            System.out.println(i.next());
        }
 }
}
```

Example 4

```
import java.util.*;
public class Main{
 public static void main(String args[]){
 TreeSet<Integer> set=new TreeSet<Integer>();
        set.add(24);
        set.add(66);
        set.add(12);
        set.add(15);
        System.out.println("Lowest Value: "+set.pollFirst());
        System.out.println("Highest Value: "+set.pollLast());
        System.out.println(set);
 }
}
```

Example 5

CODERINDEED

```
// Java code to demonstrate
// the working of TreeSet
import java.util.*;
public class Main {
        public static void main(String[] args)
        {
        TreeSet<String> ts= new TreeSet<String>();
        // Elements are added using add() method
        ts.add("AA");
        ts.add("BB");
        ts.add("CC");
        System.out.println("Tree Set is " + ts);
        String check = "BB";
// Check if the above string exists in the treeset or not
System.out.println("Contains " + check + " " + ts.contains(check));
// Print the first element in the TreeSet
System.out.println("First Value " + ts.first());
// Print the last element in the TreeSet
System.out.println("Last Value " + ts.last());
String val = "BB";
// Find the values just greater and smaller than the above string
System.out.println("Higher " + ts.higher(val));
System.out.println("Lower " + ts.lower(val));
}
}
```

Example 6[Iterating through TreeSet]


```
// Java code to demonstrate the working of TreeSet
import java.util.*;
class TreeSetDemo {
        public static void main(String[] args)
        {
    TreeSet<String> ts= new TreeSet<String>();
                // Elements are added using add() method
                ts.add("AA");
                ts.add("BB");
                ts.add("CC");
                // Iterating though the TreeSet
                for (String value : ts)
              System.out.print(value+ ", ");
                System.out.println();
        }
}
```

Comparator Interface

- java.util.Comparator interface provides two abstract methods:

```
public interface Comparator<T>
        {
          public abstract int compare(T ob1, T ob2);
          public abstract boolean equals(java.lang.Object);
          …
        }
```

Comparable Interface

- java.lang.Comparable interface provides only one abstract method:

```
public interface Comparable<T>
        {
          public abstract int compareTo(T ob);
        }
```

Implementing TreeSet with Comparator sorting in descending order


```
import java.util.*;
public class Main {
public static void main(String args[]){
Set<Integer> tree = new TreeSet<Integer>(new Comparator<Integer>() {
public int compare(Integer o1, Integer o2) {
return o2.compareTo(o1);
}
});
tree.add(3);
tree.add(1);
tree.add(2);
System.out.println("treeSet : "+tree);
}
}
```

Implementing TreeSet with Custom Objects using Comparator


```
import java.util.*;
class Employee{
String name;
String id;
public Employee(String name, String id) {
this.name = name;
this.id = id;
}
public String toString() {
return "Employee{" + "name=" + name + ", id=" + id + '}';
}
}
```

Implementing TreeSet with Custom Objects using Comparator


```
public class Main {
public static void main(String...a){
Employee emp1=new Employee("Smith","4");
Employee emp2=new Employee("Paul","2");
Employee emp3=new Employee("Charles","1");
Set<Employee> treeSet = new TreeSet<Employee>(new
   Comparator<Employee>()
{
public int compare(Employee o1, Employee o2)
{ return o1.name.compareTo(o2.name); }
});
treeSet.add(emp1);treeSet.add(emp2);treeSet.add(emp3);
System.out.println("treeSet : "+treeSet);
}}
```

DEQUE

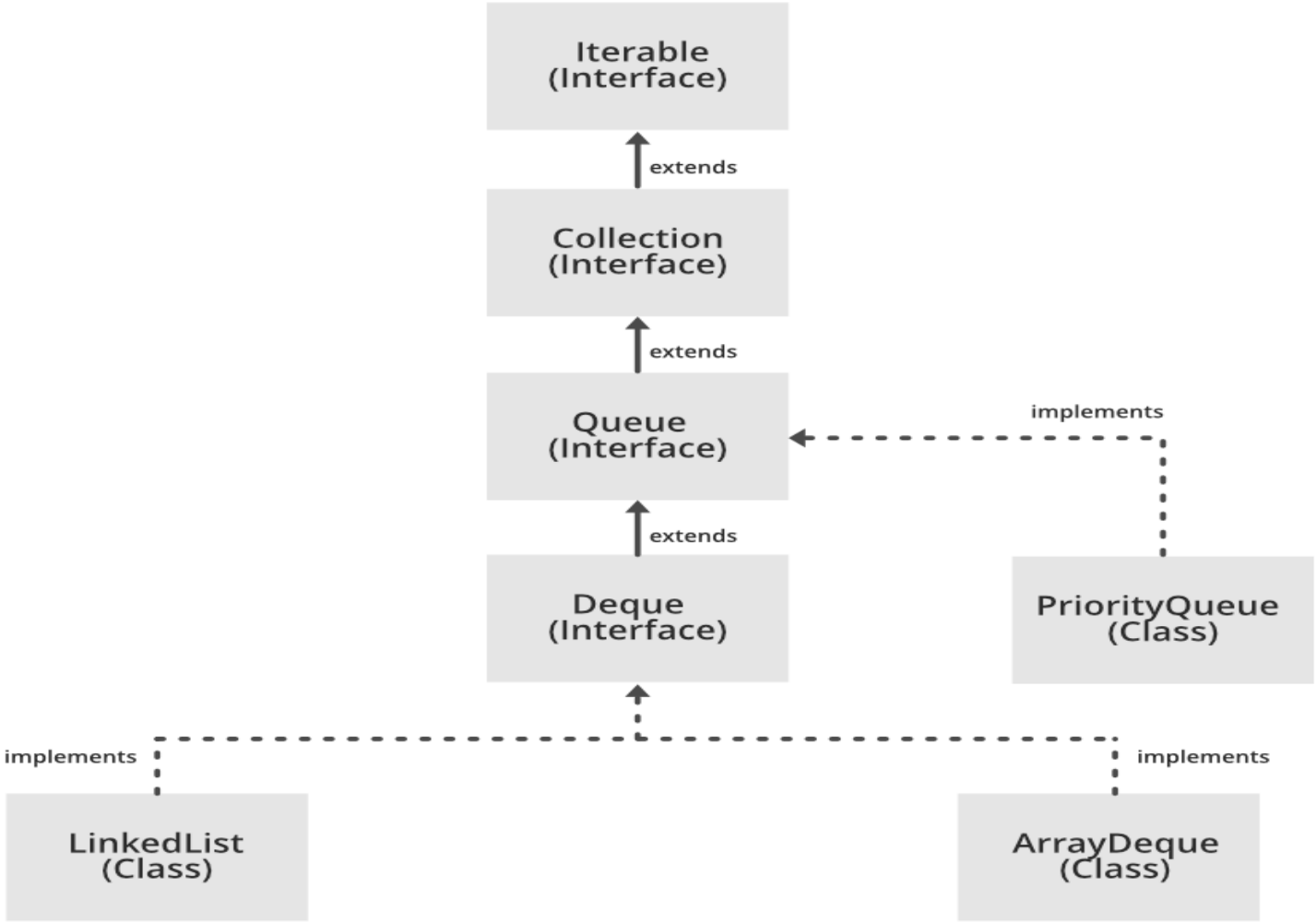
Deque Interface

- Deque supports element insertion and removal at both ends.
- The name deque is short for “double-ended queue” and is usually pronounced “deck.”

- The Deque interface extends Queue with additional methods for inserting and removing elements from both ends of the queue.

- The methods *addFirst(e)*, *removeFirst()*, *addLast(e)*, *removeLast()*, *getFirst()*, and *getLast()* are defined in the Deque interface.

Hierarchy of classes



Deque Methods

Type of Operation	First Element (Beginning of the Deque instance)	Last Element (End of the Deque instance)
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

Methods Description

*addFirst():*Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted deque, it is generally preferable to use method offerFirst(E).

*offerFirst():*Inserts the specified element at the front of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the addFirst(E) method, which can fail to insert an element only by throwing an exception.

*removeFirst():*Retrieves and removes the first element of this deque. This method differs from pollFirst only in that it throws an exception if this deque is empty.

Throws:

NoSuchElementException - if this deque is empty

*pollFirst():*Retrieves and removes the first element of this deque, or returns null if this deque is empty.

*addLast():*Inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted deque, it is generally preferable to use method offerLast(E).

*offerLast():*Inserts the specified element at the end of this deque unless it would violate capacity restrictions. When using a capacity-restricted deque, this method is generally preferable to the addLast(E) method, which can fail to insert an element only by throwing an exception.

removeLast()

Retrieves and removes the last element of this deque. This method differs from pollLast only in that it throws an exception if this deque is empty.

pollLast()

Retrieves and removes the last element of this deque, or returns null if this deque is empty.

Insert Methods

- The addfirst and offerFirst methods insert elements at the beginning of the Deque instance.
- The methods addLast and offerLast insert elements at the end of the Deque instance.
- When the capacity of the Deque instance is restricted, the preferred methods are offerFirst and offerLast because addFirst might fail to throw an exception if it is full.

Remove Methods

- removeFirst and pollFirst methods remove elements from the beginning of the Deque.
- The removeLast and pollLast methods remove elements from the end.
- The methods pollFirst and pollLast return null if the Deque is empty whereas the methods removeFirst and removeLast throw an exception if the Deque instance is empty.

Retrieve Methods

- getFirst and peekFirst retrieve the first element but don't remove the value from the Deque.
- Similarly, getLast and peekLast retrieve the last element.
- The methods getFirst and getLast throw an exception if the deque instance is empty whereas the methods peekFirst and peekLast return NULL.

Deque Implementation

The ArrayDeque class is the resizable array implementation of the Deque interface, whereas the LinkedList class is the list implementation.

- The LinkedList is more flexible than the ArrayDeque as null elements are allowed in the LinkedList implementation but not in ArrayDeque.
- ArrayDeque is more efficient than the LinkedList for add and remove operation at both ends.
- The best operation in a LinkedList implementation is removing the current element during the iteration.

ArrayDeque Constructors

ArrayDeque()

Constructs an empty array deque with an initial capacity sufficient to hold 16 elements.

ArrayDeque(Collection<? extends E> c)

Constructs a deque containing the elements of the specified collection, in the order they are returned by the collection's iterator.

ArrayDeque(int numElements)

Constructs an empty array deque with an initial capacity sufficient to hold the specified number of elements.

LinkedList Constructors

LinkedList()

Constructs an empty LinkedList.

LinkedList(Collection<? extends E> c)

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Example 1

CODERINDEED

```
// Java program to demonstrate the working of a Deque in Java
import java.util.*;
public class Main {
        public static void main(String[] args)
        {
                Deque<String> deque= new LinkedList<String>();
                // We can add elements to the queue in various ways
                // Add at the last
                deque.add("Element 1 (Tail)");
                // Add at the first
                deque.addFirst("Element 2 (Head)");
                // Add at the last
                deque.addLast("Element 3 (Tail)");
                // Add at the first
                deque.push("Element 4 (Head)");
                // Add at the last
                deque.offer("Element 5 (Tail)");
                // Add at the first
                deque.offerFirst("Element 6 (Head)");
                System.out.println(deque + "\n");
                // We can remove the first element or the last element.
                deque.removeFirst();
                deque.removeLast();
                System.out.println("Deque after removing "+ "first and last: "+ deque);
        }
```

Output:

[Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail), Element 5 (Tail)]

Deque after removing first and last:

[Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail)]

Example 2

```
// Java program to demonstrate the
// addition of elements in deque
import java.util.*;
public class Main {
        public static void main(String[] args)
        {
                // Initializing an deque
        Deque<String> dq= new ArrayDeque<String>();
                // add() method to insert
                dq.add("AA");
                dq.addFirst("BB");
                dq.addLast("CC");
                System.out.println(dq);
        }
}
```

Example 3

CODERINDEED

```
// Java program to demonstrate the removal of elements in deque
import java.util.*;
public class Main {
        public static void main(String[] args)
        {
                // Initializing an deque
                Deque<String> dq= new ArrayDeque<String>();
                // add() method to insert
                dq.add("ABC");
                dq.addFirst("PQR");
                dq.addLast("STU");
                System.out.println(dq);
                System.out.println(dq.pop());
                System.out.println(dq.pollFirst());
                System.out.println(dq.pollLast());
                System.out.println(dq);
        }
}
```

Example 4

CODERINDEED

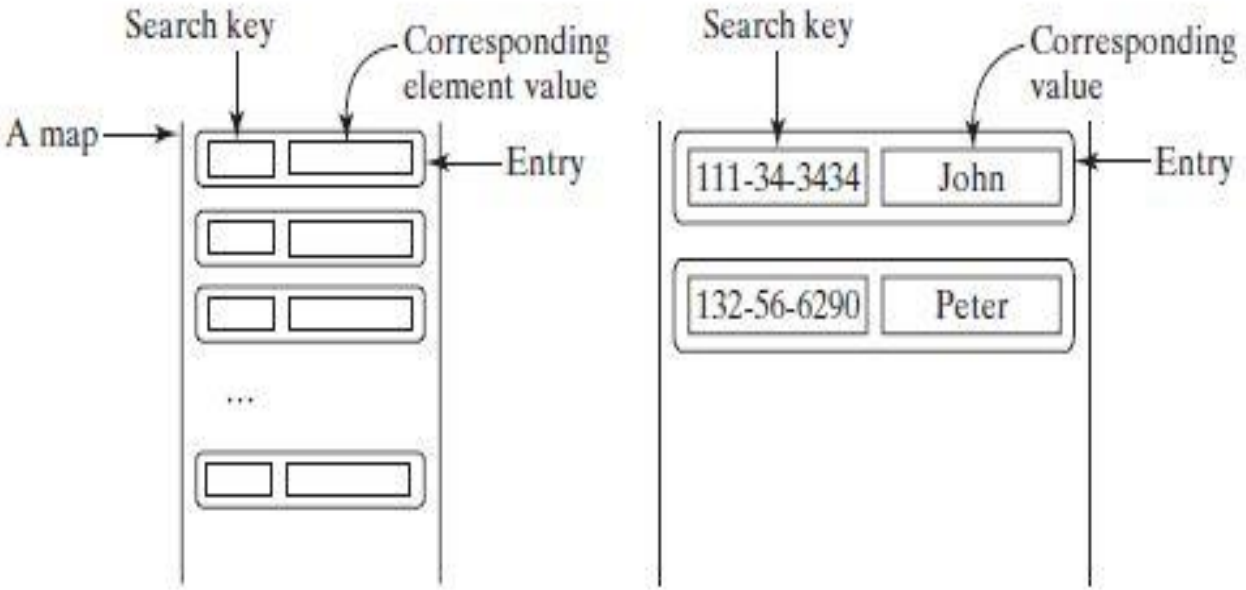
```
// Java program to demonstrate the iteration of elements in deque
import java.util.*;
public class Main {
        public static void main(String[] args)
        {
                // Initializing an deque
                Deque<String> dq= new ArrayDeque<String>();
                // add() method to insert
                dq.add("ABC");
                dq.addFirst("PQR");
                dq.addLast("STU");
                dq.add("BB");
                for (Iterator itr = dq.iterator();
                        itr.hasNext();) {
                        System.out.print(itr.next() + " ");
                }
                System.out.println();
                for (Iterator itr = dq.descendingIterator();
                        itr.hasNext();) {
                        System.out.print(itr.next() + " ");
                }
        }
}
```

MAP INTERFACE

MAP Interface

- A map is a container object that stores a collection of key/value pairs.
- Keys are like indexes in List but in Map, the keys can be any objects.
- A map cannot contain duplicate keys.
- Each key maps to one value.
- A key and its corresponding value form an entry stored in a map.

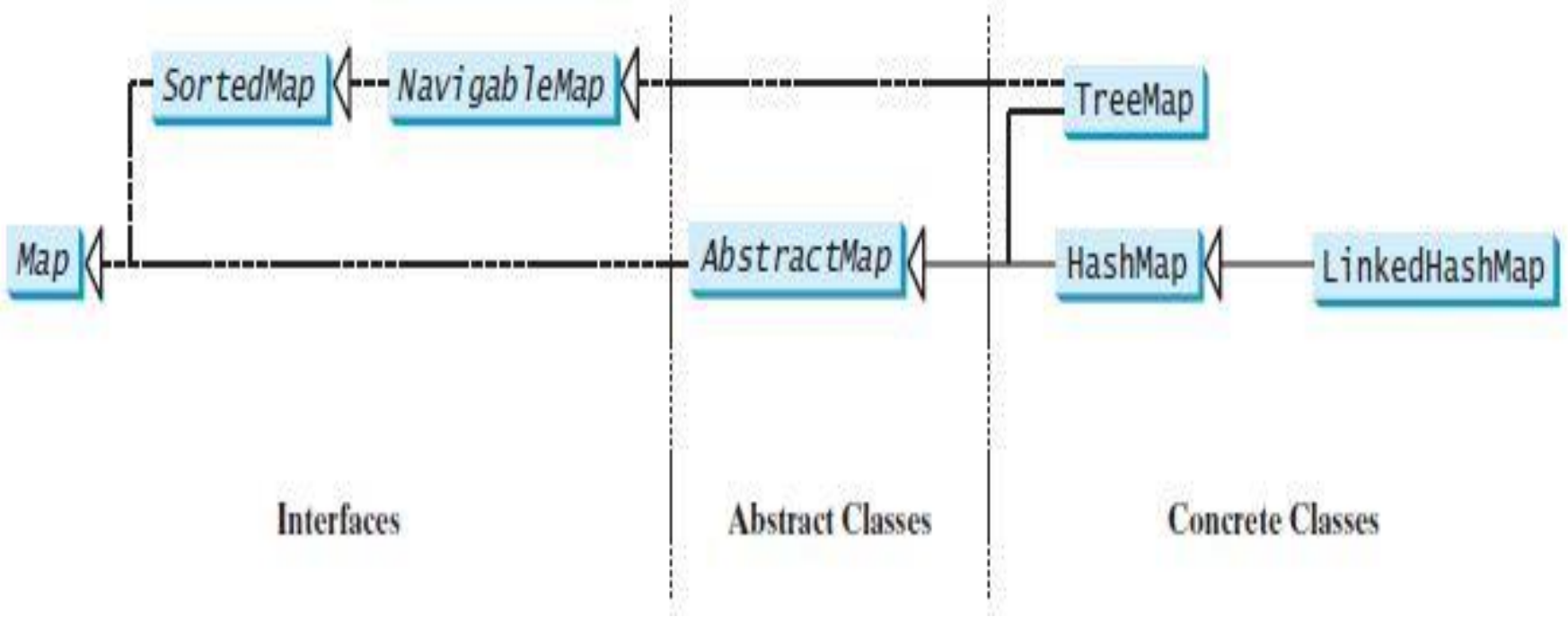
MAP

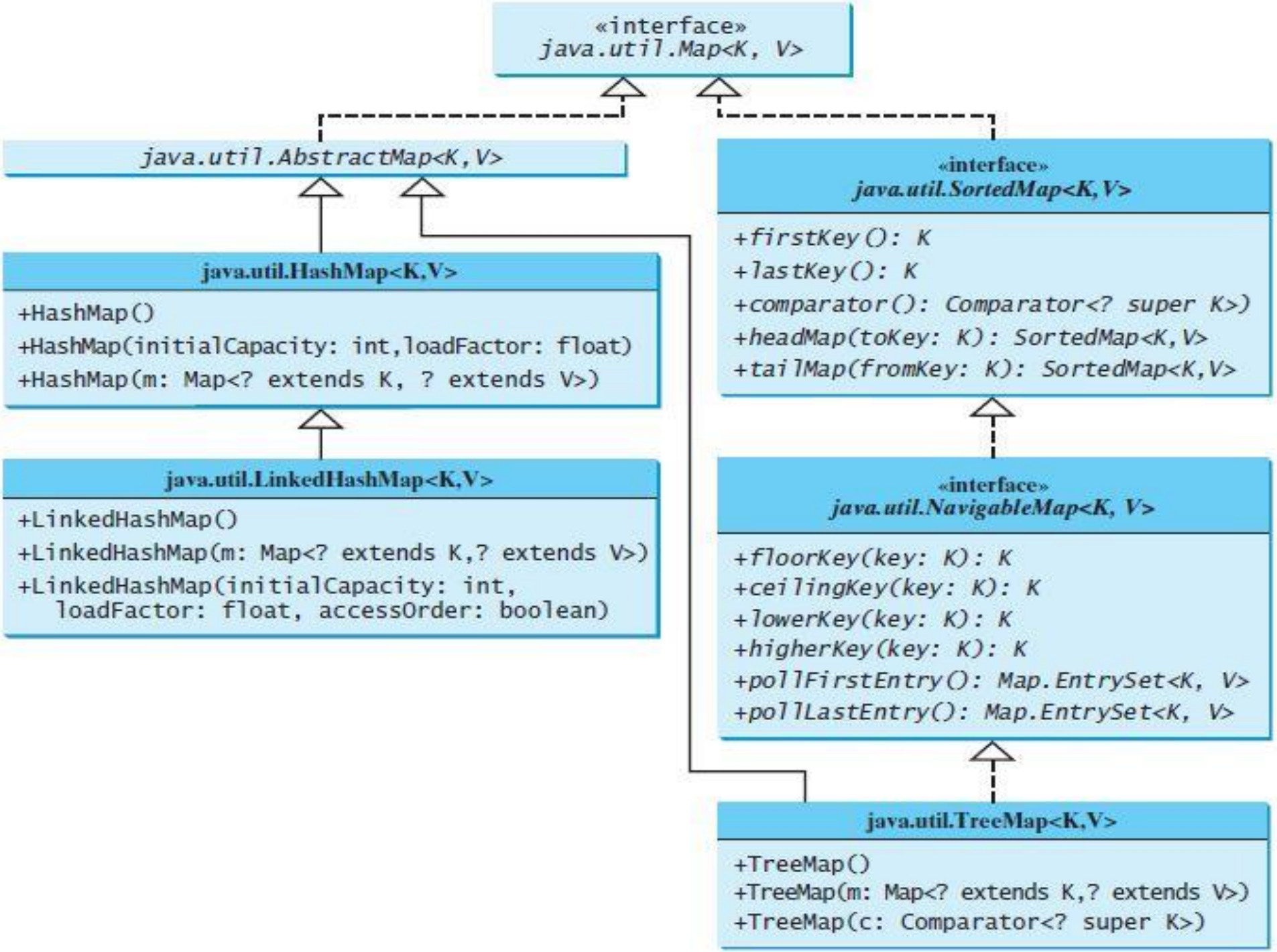


MAP Methods

- int size()
- void clear()
- boolean isEmpty()
- V put(K key, V value)
- void putAll(m: Map<? extends K,? extendsV>)
- boolean containsKey (Object key)
- boolean containsValue(Object value)
- V get(Object key)
- Set<K> keySet()
- V remove(Object key)
- Collection <V> values()
- Set<Map.Entry<K,V>> entrySet()

MAP Class Hierarchy




«interface»
java.util.Map<K, V>
java.util.AbstractMap<K,V>
«interface»
java.util.SortedMap<K,V>
+firstKey(): K
+lastKey(): K
+comparator(): Comparator<? super K>)
+headMap(toKey: K): SortedMap<K,V>
+tailMap(fromKey: K): SortedMap<K,V>
java.util.HashMap<K,V>
+HashMap()
+HashMap(initialCapacity: int,loadFactor: float)
+HashMap(m: Map<? extends K, ? extends V>)
«interface»
java.util.NavigableMap<K, V>
+floorKey(key: K): K
+ceilingKey(key: K): K
+lowerKey(key: K): K
+higherKey(key: K): K
+pollFirstEntry(): Map.EntrySet<K, V>
+pollLastEntry(): Map.EntrySet<K, V>
java.util.LinkedHashMap<K,V>
+LinkedHashMap()
+LinkedHashMap(m: Map<? extends K,? extends V>)
+LinkedHashMap(initialCapacity: int,
loadFactor: float, accessOrder: boolean)
java.util.TreeMap<K,V>
+TreeMap()
+TreeMap(m: Map<? extends K,? extends V>)
+TreeMap(c: Comparator<? super K>)

HashMap

- The HashMap class is efficient for locating a value, inserting an entry, and deleting an entry.

- The entries in a HashMap are not ordered.

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

More features of HashMap

- HashMap is a part of java.util package.
- HashMap extends an abstract class AbstractMap which also provides an incomplete implementation of Map interface.
- HashMap doesn't allow duplicate keys but allows duplicate values. That means A single key can't contain more than 1 value but more than 1 key can contain a single value.
- HashMap allows null key also but only once and multiple null values.

HashMap

Constructors:

- HashMap()

  Constructs an empty HashMap with the default initial capacity (16)

- HashMap(int initialCapacity)

  Constructs an empty HashMap with the specified initial capacity

- HashMap(Map<? extends K,? extends V> m)

  Constructs a new HashMap with the same mappings as the specified Map.

Example 1

CODERINDEED

```
// Java program to demonstrate the HashMap() constructor
import java.util.*;
public class Main {
        public static void main(String args[])
        {
                // No need to mention the Generic type twice
                HashMap<Integer, String> hm1 = new HashMap<>();
                // Initialization of a HashMap using Generics
                HashMap<Integer, String> hm2= new HashMap<Integer, String>();
                // Add Elements using put method
                hm1.put(1, "one");
                hm1.put(2, "two");
                hm1.put(3, "three");
                hm2.put(4, "four");
                hm2.put(5, "five");
                hm2.put(6, "six");
                System.out.println("Mappings of HashMap hm1 are : "+ hm1);
                System.out.println("Mapping of HashMap hm2 are : "+ hm2);
        }
}
```

Example 2

CODERINDEED

```
// Java program to demonstrate the HashMap(int initialCapacity) constructor
import java.util.*;
public class Main {
        public static void main(String args[])
        {
                // No need to mention the Generic type twice
                HashMap<Integer, String> hm1 = new HashMap<>(10);
                // Initialization of a HashMap using Generics
                HashMap<Integer, String> hm2= new HashMap<Integer, String>(2);
                // Add Elements using put method
                hm1.put(1, "one");
                hm1.put(2, "two");
                hm1.put(3, "three");
                hm2.put(4, "four");
                hm2.put(5, "five");
                hm2.put(6, "six");
                System.out.println("Mappings of HashMap hm1 are : "+ hm1);
                System.out.println("Mapping of HashMap hm2 are : "+ hm2);
        }
}
```

Example 3

CODERINDEED

```
// Java program to demonstrate the HashMap(Map map) Constructor
import java.util.*;
public class Main {
        public static void main(String args[])
        {
                // No need to mention the Generic type twice
                Map<Integer, String> hm1 = new HashMap<>();
                // Add Elements using put method
                hm1.put(1, "one");
                hm1.put(2, "two");
                hm1.put(3, "three");
                // Initialization of a HashMap using Generics
                HashMap<Integer, String> hm2= new HashMap<Integer, String>(hm1);
                System.out.println("Mappings of HashMap hm1 are : "+ hm1);
                System.out.println("Mapping of HashMap hm2 are : "+ hm2);
        }
}
```

Example 4

CODERINDEED

```
// Java program to add elements to the HashMap
import java.util.*;
public class Main {
        public static void main(String args[])
        {
                // No need to mention the Generic type twice
                HashMap<Integer, String> hm1 = new HashMap<>();
                // Initialization of a HashMap using Generics
                HashMap<Integer, String> hm2= new HashMap<Integer, String>();
                // Add Elements using put method
                hm1.put(1, "One");
                hm1.put(2, "two");
                hm1.put(3, "three");
                hm2.put(1, "AA");
                hm2.put(2, "BB");
                hm2.put(3, "CC");
                System.out.println("Mappings of HashMap hm1 are : "+ hm1);
                System.out.println("Mapping of HashMap hm2 are : "+ hm2);
        }
}
```

Example 5


```
// Java program to illustrate Java.util.HashMap
import java.util.HashMap;
public class Main {
        public static void main(String[] args)
        {
                // Create an empty hash map
                HashMap<String, Integer> map = new HashMap<>();
                // Add elements to the map
                map.put("ABC", 10);
                map.put("PQR", 30);
                map.put("STU", 20);
                // Print size and content
                System.out.println("Size of map is:- "+ map.size());
                System.out.println(map);
                // Check if a key is present and if present, print value
                if (map.containsKey("ABC")) {
                        Integer a = map.get("ABC");
                        System.out.println("Value for key"+ " \"ABC\" is:- " + a);
                }
        }
}
```

Example 6

CODERINDEED

```
// Java program to remove elements from HashMap
import java.util.*;
public class Main{
        public static void main(String args[])
        {
                // Initialization of a HashMap
                Map<Integer, String> hm= new HashMap<Integer, String>();
                // Add elements using put method
                hm.put(1, "AA");
                hm.put(2, "BB");
                hm.put(3, "CC");
                hm.put(4, "DD");
                // Initial HashMap
                System.out.println("Mappings of HashMap are : "+ hm);
                // remove element with a key using remove method
                hm.remove(4);
                // Final HashMap
                System.out.println("Mappings after removal are : "+ hm);
        }
}
```

Example 7

CODERINDEED

```
// Java program to traversal a Java.util.HashMap
import java.util.HashMap;
import java.util.Map;
public class Main {
        public static void main(String[] args)
        {
                // initialize a HashMap
                HashMap<String, Integer> map = new HashMap<>();
                // Add elements using put method
                map.put("AA", 10);
                map.put("BB", 30);
                map.put("CC", 20);
                // Iterate the map using for-each loop
                for (Map.Entry<String, Integer> e : map.entrySet())
                        System.out.println("Key: " + e.getKey() + "Value: " + e.getValue());
        }
}
```

Example 8(More methods)

CODERINDEED

```
// Java program to traversal a Java.util.HashMap
import java.util.*;
public class Main {
        public static void main(String[] args)
        {
                // initialize a HashMap
                HashMap<String, Integer> map = new HashMap<>();
                // Add elements using put method
                map.put("AA", 10);
                map.put("BB", 30);
                map.put("CC", 20);
                HashMap<String, Integer> map1 = new HashMap<>();
                map1.putAll(map);
                System.out.println(map1);
                System.out.println(map.get("AA"));
                Collection c=map.values();
                for(Object x:c)
                System.out.println(x);
                Set s=map.keySet();
                for(Object x:s)
                System.out.println(x);

        }
}
```

Q1

Which of these packages contain all the collection classes?

a) java.lang
b) java.util
c) java.net
d) java.awt

Q2

Which of these methods deletes all the elements from invoking collection?

a) clear()
b) reset()
c) delete()
d) refresh()

Q3

What is Collection in Java?

a) A group of objects
b) A group of classes
c) A group of interfaces
d) None of the mentioned

Q4

The collection interface

A.extends Collections class

B.extends Iterable interface

C.implements Serializable interface

D.implements Traversable interface

Q5

The collection is a _____

A.framework and interface

B.framework and class

C.only interface

D.only class

Q6

List, Set and Queue _____ Collection.

A.extends

B.implements

C.both of the above

D.none of the above

Q7

Arraylist, Linkedlist and vector are all _____

A.interfaces

B.enums

C.classes

D.Depends on implementation

Q8

Which of the following is true?

1. Iterator can traverse in both forward and backward directions.

2. ListIterator can traverse in both forward and backward directions.

A. Only 1

B. Only 2

C. Both 1 & 2

D. None of the above

Q9

Which of the following interface doesn't extend collection interface?

A. List

B. Set

C. Map

D. Queue

Q10

The default capacity of a ArrayList is:

a.12

b.16

c.1

d.10

Q11

Deque and Queue are derived from:

a.AbstractList

b.Collection

c.AbstractCollection

d.List

Q12

nextIndex() and previousIndex() are methods of which interface?

a.IndexIterator

b.Iterator

c.ListIterator

d.NextPreviousIterator

Q13

Which of these allows duplicate elements?

a.Set

b.List

c.All

d.None