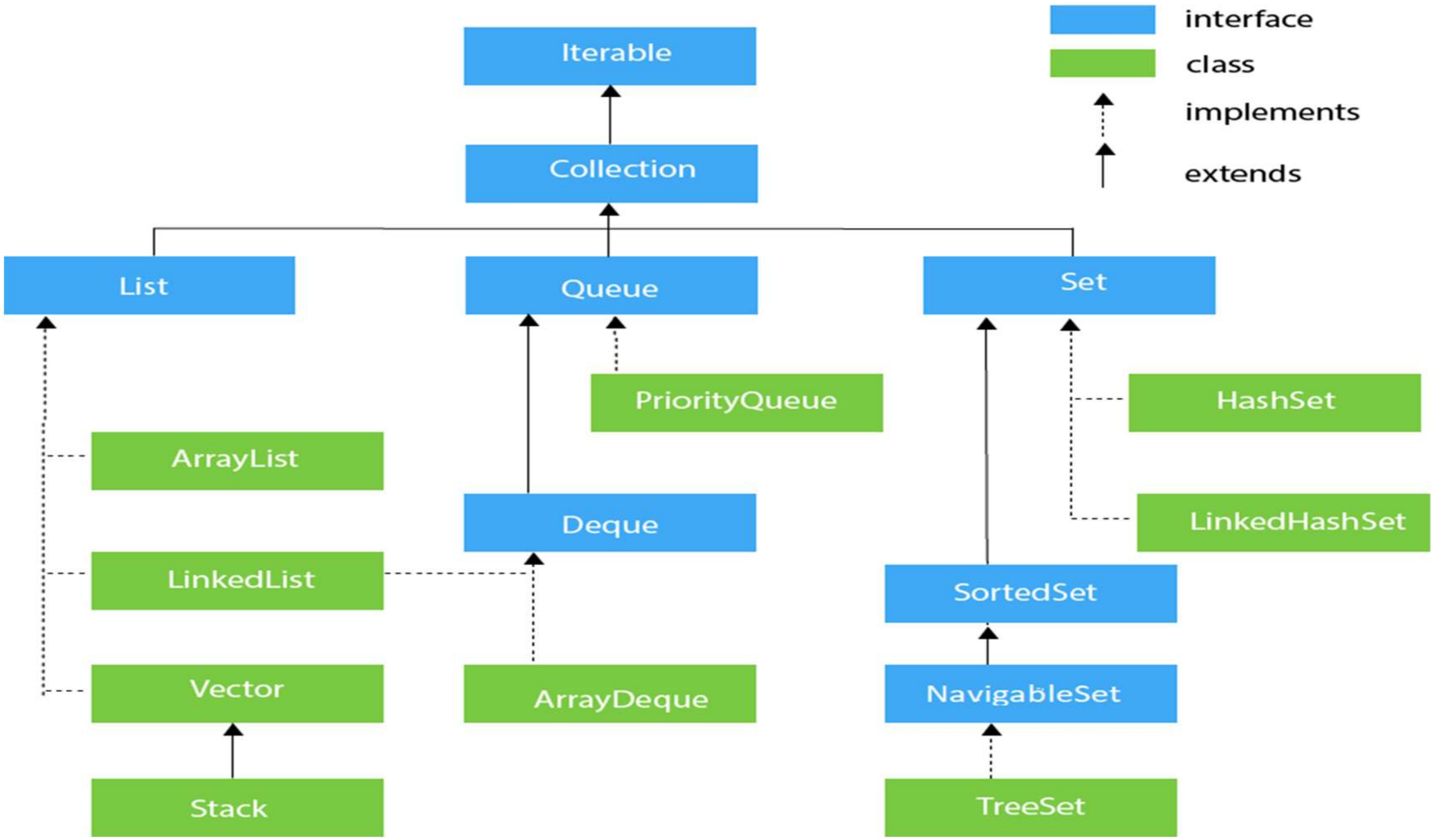
Collections in Java

AUGUST,2023

Collections - Introduction

- The Java Collections Framework **standardizes the way in which groups of objects are handled** by your programs
- Java provided ad hoc classes such as **Dictionary, Vector, Stack** and **Properties** to store and manipulate groups of objects

Hierarchy of Collection Framework



Collections Interfaces

- The Collections Framework defines several core interfaces

Interface	Description
Collection	Enable you to work with groups of objects; it is at the top of the collections hierarchy
Deque	Extends **Queue** to handle a double-ended queue
List	Extends **Collection** to handle sequences (list of objects)
NavigableSet	Extends **SortedSet** to handle retrieval of elements are removed only from the head.
Queue	Extends **Collection** to handle special types of list in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements
SortedSet	Extends **Set** to handle Sorted Sets.

Collections Interfaces

- In addition to the collection interfaces, collections also use the **Comparator**, **RandomAccess**, **Iterator**, **ListIterator**, and **Spliterator** interfaces.
- **Comparator** defines how two objects are compared.
- **Iterator**, **ListIterator** and **Spliterator** enumerate the objects within a collection.
- By implementing **RandomAccess**, a list indicates that it supports efficient, random access to its elements.

The Collection Interfaces

- The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection.
- **Collection** is a generic interface that has this declaration:

  interface Collection<E>

  Here, **E** specifies the type of objects that the collection will hold.
- **Collection** extends the **Iterable** interface.

The Collection Interfaces

Collection declares the core methods that all collections will have. These methods are

Method	Description
public boolean **add**(E e)	It is used to **insert an element** in this collection.
public boolean **addAll** (Collection<? extends E> c)	It is used to **insert the specified collection elements** in the invoking collection.
public boolean **remove** (Object element)	It is used to **delete an element** from the collection.
public boolean **removeAll** (Collection<?> c)	It is used to **delete all the elements of the specified collection** from the invoking collection.
default boolean **removeIf** (Predicate<? super E> filter)	It is used to **delete all the elements of the collection that satisfy the specified predicate**.
public boolean **retainAll** (Collection<?> c)	It is used to **delete all the elements of invoking collection except the specified collection**.
public int **size**()	It returns the **total number of elements** in the collection.
public void **clear**()	It **removes the total number of elements** from the collection.
public boolean **contains** (Object element)	It is used to **search** an element.

The Collection Interfaces

Method	Description
public boolean **containsAll** (Collection<?> c)	It is used to **search the specified collection** in the collection.
public Iterator **iterator**()	It returns an **iterator**.
public Object[] **toArray**()	It converts **collection into array**.
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.
public boolean **isEmpty**()	It checks if **collection is empty.**
default Stream<E> **parallelStream**()	It returns a **possibly parallel Stream** with the collection as its source.
default Stream<E> **stream**()	It returns a **sequential Stream** with the collection as its source.
default **Spliterator**<E> spliterator()	It generates a **Spliterator over the specified elements** in the collection.
public boolean **equals**(Object element)	It **matches** two collections.
public int **hashCode**()	It returns the **hash code number of the collection**.

The List Interface

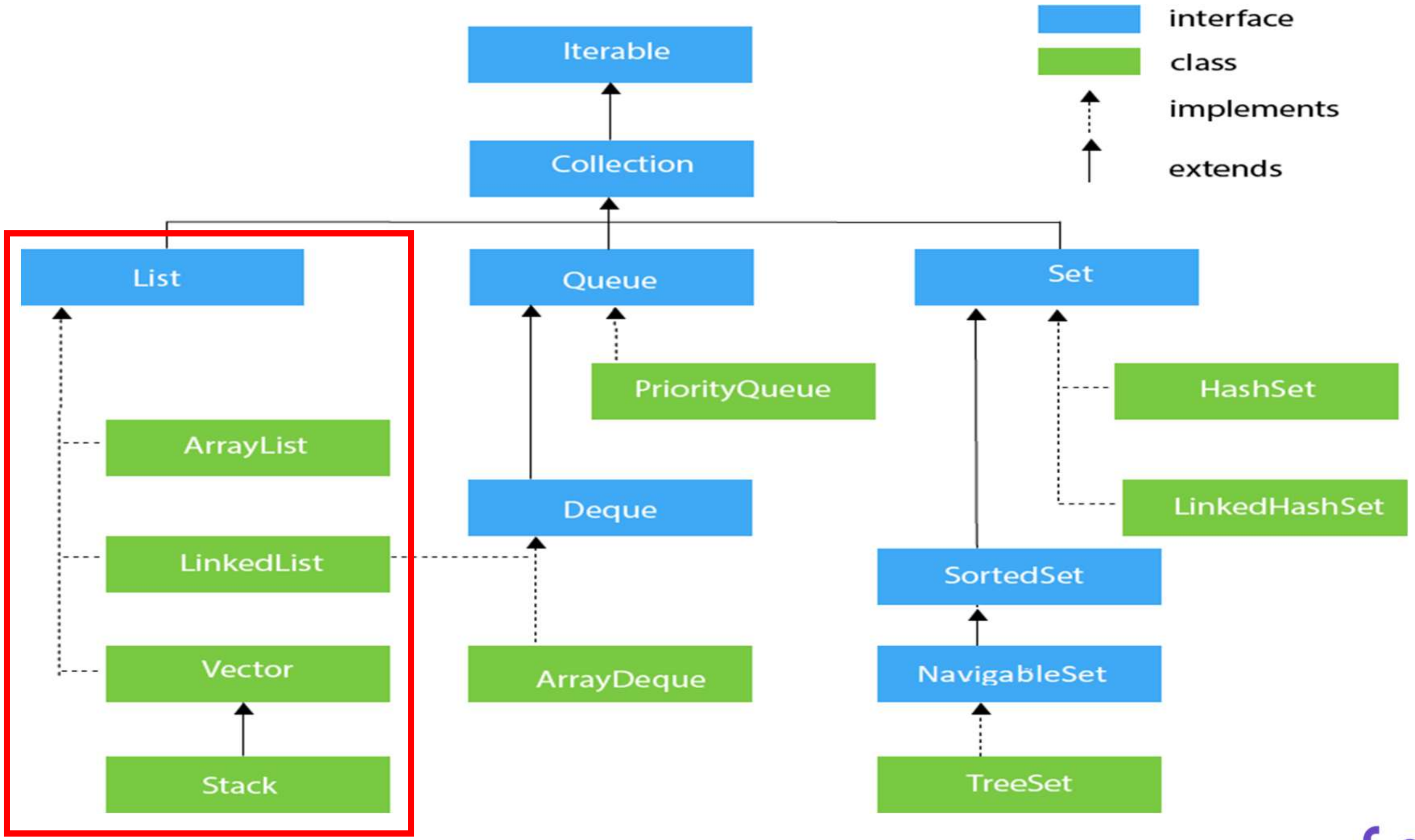
- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be **inserted or accessed** by their position in the list, using a zero-based index.
- A list may contain **duplicate** elements.
- **List** is a generic interface that has this declaration:

  interface List<E>

  Here, **E** specifies the type of objects that the list will hold.
- In addition to the methods defined by **Collection**, **List** defines some of its own

Collections in Java

The List Interface



The List Interface

- In addition to the methods defined by **Collection**, **List** defines some of its own

Method	Description
void **add**(int index, E element)	It is used to **insert the specified element** at the specified position in 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.
E **get**(int index)	It is used to **fetch the element from the particular position** of the list.
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.
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.
void **replaceAll** (UnaryOperator<E> operator)	It is used to **replace all the elements** from the list with the specified element.

The List Interface

Method	Description
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**.
List<E> **subList**(int fromIndex, int toIndex)	It is used to **fetch all the elements lies within the given range.**
Static <E> List <E> **copyOf** (Collection<? extends E> from)	It returns a **list that contains the same elements as that specified by *from***

The ArrayList Class

- The **ArrayList** class extends **AbstractList** and implements the **List** interface.
- **ArrayList** is a generic class that has this declaration:

  class ArrayList<E>

  Here, **E** specifies the type of objects that the list will hold.
- **ArrayList** supports dynamic arrays that can grow as needed.
- In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.
- The Collections Framework defines **ArrayList** which is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size

The ArrayList Class – Example#1

```
/*This example demonstrate the ArrayList Classes*/
import java.util.*;
public class ArrayListDemo {
        public static void main(String args[]) {
                //Create an Arraylist
                ArrayList <String> Arr = new ArrayList<String>();
                System.out.println("Initial Size of Array List is "+Arr.size());
                Arr.add("C");
                Arr.add("A");
                Arr.add("E");
                Arr.add("B");
                Arr.add("D");
                Arr.add("F");
                Arr.add(1, "G");
                System.out.println("After Insert the Size of Array List is "+Arr.size());
```

Collections in Java

The ArrayList Class – Example#1

```
                System.out.println("Contents of ArrayList "+Arr);
                //Remove Element from array List
                Arr.remove("F");
                Arr.remove(2);
                System.out.println("Contents of ArrayList "+Arr);
        }
}
```

Output:

```
Initial Size of Array List is 0
After Insert the Size of Array List is 7
Contents of ArrayList [C, G, A, E, B, D, F]
Contents of ArrayList [C, G, E, B, D]
```

The ArrayList Class – Example#2

```
/*This example demonstrate the ArrayList Classes*/
import java.util.*;

public class ArrayListDemo {
        public static void main(String args[]) {
                //Create an Arraylist
                ArrayList <Integer> Arr = new ArrayList<Integer>();
                System.out.println("Initial Size of Array List is "+Arr.size());
                Arr.add(1);
                Arr.add(2);
                Arr.add(3);
                Arr.add(4);
                System.out.println("After Insert the Size of Array List is "+Arr.size());
                System.out.println("Contents of ArrayList "+Arr);
```

The ArrayList Class – Example#2

```
        Integer ia[] = new Integer[Arr.size()];
        ia = Arr.toArray(ia);
        int sum = 0;
        for(int i:ia) {
                sum+=i;
        }
        System.out.println("Sum value is "+sum);
    }
}
```

Output:

```
Initial Size of Array List is 0
After Insert the Size of Array List is 4
Contents of ArrayList [1, 2, 3, 4]
Sum value is 10
```

The LinkedList Class

- The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces.
- It provides a linked-list data structure.
- **LinkedList** is a generic class that has this declaration:

class LinkedList<E>

Here, **E** specifies the type of objects that the list will hold.

LinkedList has the two constructors shown here:

- **LinkedList()** - builds an empty linked list
- **LinkedList(Collection<? extends E> c)** – builds a linked list that is initialized with the elements of the collection *c*
- **LinkedList** also implements the **Deque** interface

The LinkedList Class – Example#1

```
/*This example demonstrate the LinkedList Classes*/
import java.util.*;
public class LinkedListDemo01 {
        public static void main(String args[]) {
                //Create Linkedlist
                LinkedList <String> Arr = new LinkedList<String>();
                System.out.println("Initial Size of LinkedList is "+Arr.size());
                Arr.add("C");
                Arr.add("A");
                Arr.add("E");
                Arr.add("B");
                Arr.add("D");
                Arr.addFirst("F");
                Arr.addLast("G");
                System.out.println("After Insert the Size of LinkedList is "+Arr.size());
```

The LinkedList Class – Example#1

```
                System.out.println("Contents of LinkedList "+Arr);
        //Remove Element from array List
        Arr.remove(“E");
        Arr.removeFirst(2);
        System.out.println("Contents of LinkedList "+Arr);
    }
}
```

Output:

```
Initial Size of LinkedList is 0
After Insert the Size of LinkedList is 7
Contents of LinkedList [F, C, A, E, B, D, G]
Contents of LinkedList [C, A, B, D, G]
```

The Stack Class

- 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 **pop(),** boolean **peek(),** boolean **push(object o)** which defines its properties.

The Stack Class – Example#1

```
/*This example demonstrate the Stack Class*/
import java.util.*;

public class StackDemo {
	public static void main(String args[]){
		Stack<String> stk=new Stack<String>();
		System.out.println("Size of the stack is "+stk.size());
		stk.push("A");
		stk.push("B");
		stk.push("C");
		stk.push("D");
		System.out.println("Elements in the stack "+stk);
		System.out.println("Size of the stack is "+stk.size());
		stk.pop();
```

The Stack Class – Example#1

```
        System.out.println("Elements in the stack after remove "+stk);
        System.out.println("Size of the stack after the removal is "+stk.size());
    }
}
```

Output:

```
Size of the stack is 0
Elements in the stack [A, B, C, D]
Size of the stack is 4
Elements in the stack after remove [A, B, C]
Size of the stack after the removal is 3
```

The Set Interface

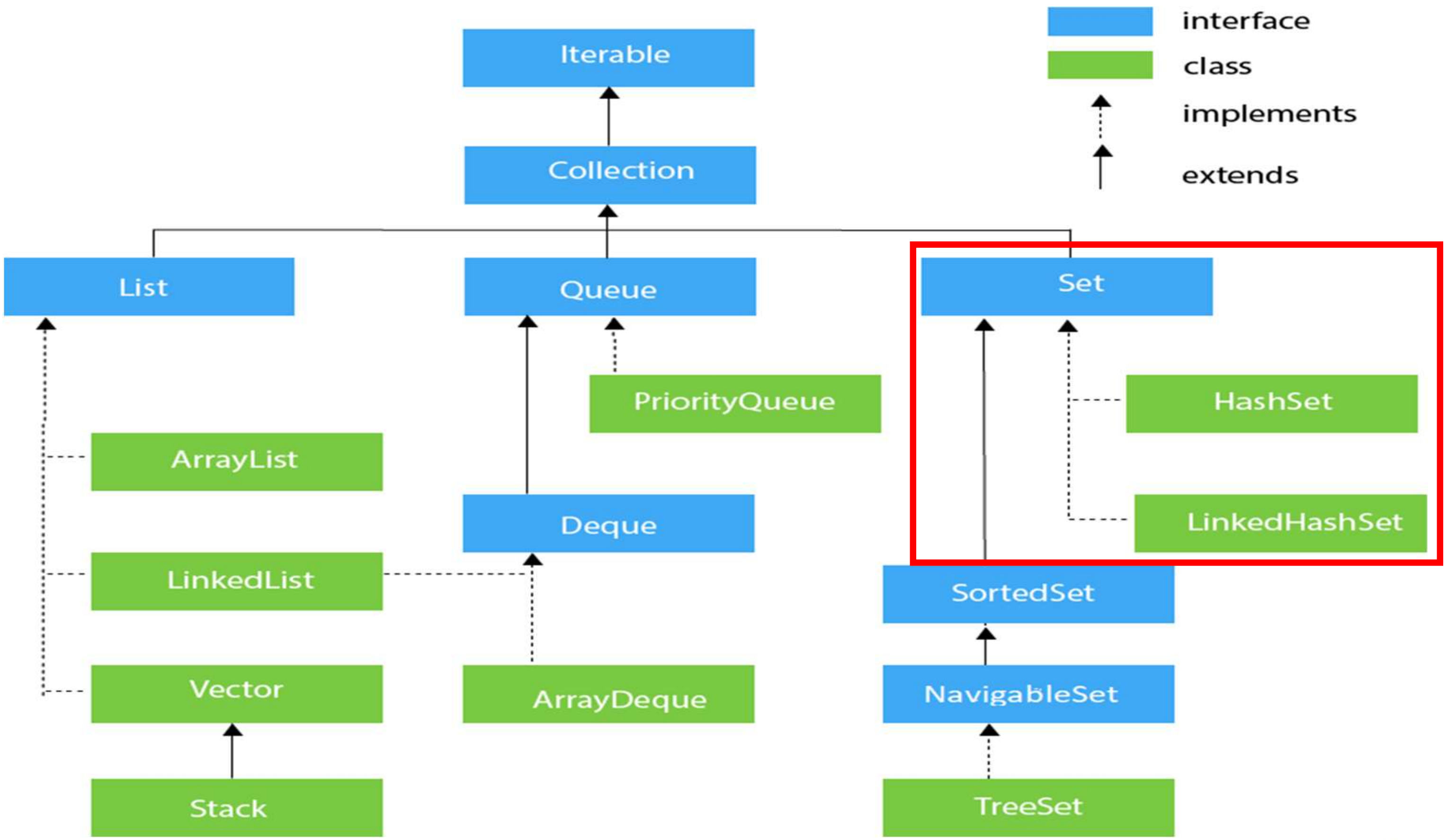
- The **Set** interface defines a set.
- It extends **Collection** and specifies the behavior of a collection that does not allow duplicate elements.
- The **add( )** method returns **false** if an attempt is made to add duplicate elements to a set.
- With two exceptions, it does not specify any additional methods of its own.
- **Set** is a generic interface that has this declaration:

  interface Set<E>

  Here, **E** specifies the type of objects that the set will hold.

Collections in Java

The Set Interface



The HashSet Class

- **HashSet** extends **AbstractSet** and implements the **Set** interface.
- It creates a collection that uses a hash table for storage.
- **HashSet** is a generic class that has this declaration:

  class HashSet<E>

  Here, **E** specifies the type of objects that the set will hold.
- In **hashing**, the informational content of a key is used to determine a unique value, called its **hash code**.
- **HashSet** does not define any additional methods beyond those provided by its superclasses and interfaces.

The HashSet Class – Example#1

```
/*This example demonstrate the Stack Class*/
import java.util.*;
public class HashSetDemo {
        public static void main(String args[]){
                HashSet<String> hs=new HashSet<String>();
                System.out.println("Size of the HashSet is "+hs.size());
                //adding elements
                hs.add("Alpha");
                hs.add("Beta");
                hs.add("Gamma");
                hs.add("Epsilon");
                hs.add("Eta");
                hs.add("Omega");
                System.out.println("Elements in the HashSet "+hs);
                System.out.println("Size of the HashSet is "+hs.size());
```

The HashSet Class – Example#1

```
        hs.remove("Eta");
        System.out.println("Elements in the HashSet after remove "+hs);
        System.out.println("Size of the HashSet after the removal is "+hs.size());
    }
}
```

Output:

```
Size of the HashSet is 0
Elements in the HashSet [Gamma, Eta, Alpha, Epsilon, Omega, Beta]
Size of the HashSet is 6
Elements in the HashSet after remove [Gamma, Alpha, Epsilon, Omega, Beta]
Size of the HashSet after the removal is 5
```

The LinkedHashSet Class

- The **LinkedHashSet** class extends **HashSet** and adds no members of its own.
- It is a generic class that has this declaration:

  class LinkedHashSet<E>

  Here, **E** specifies the type of objects that the set will hold.
- Its constructors parallel those in **HashSet**.
- **LinkedHashSet** maintains a linked list of the entries in the set, in the order in which they were inserted.

The LinkedHashSet Class – Example#1

```
/*This example demonstrate the Stack Class*/
import java.util.*;

public class LinkedHashSetDemo01 {
	public static void main(String args[]){
		LinkedHashSet<String> hs=new LinkedHashSet<String>();
		System.out.println("Size of the LinkedHashSet is "+hs.size());
		//adding elements
		hs.add("Alpha");
		hs.add("Beta");
		hs.add("Gamma");
		hs.add("Epsilon");
		hs.add("Eta");
		hs.add("Omega");
		System.out.println("Elements in the LinkedHashSet "+hs);
```

The LinkedHashSet Class – Example#1

```
        System.out.println("Size of the LinkedHashSet is "+hs.size());
        hs.remove("Eta");
        System.out.println("Elements in the LinkedHashSet after remove "+hs);
        System.out.println("Size of the LinkedHashSet after the removal is "+hs.size());
    }
}
```

Output:

```
Size of the LinkedHashSet is 0
Elements in the LinkedHashSet [Alpha, Beta, Gamma, Epsilon, Eta, Omega]
Size of the LinkedHashSet is 6
Elements in the LinkedHashSet after remove [Alpha, Beta, Gamma, Epsilon, Omega]
Size of the LinkedHashSet after the removal is 5
```

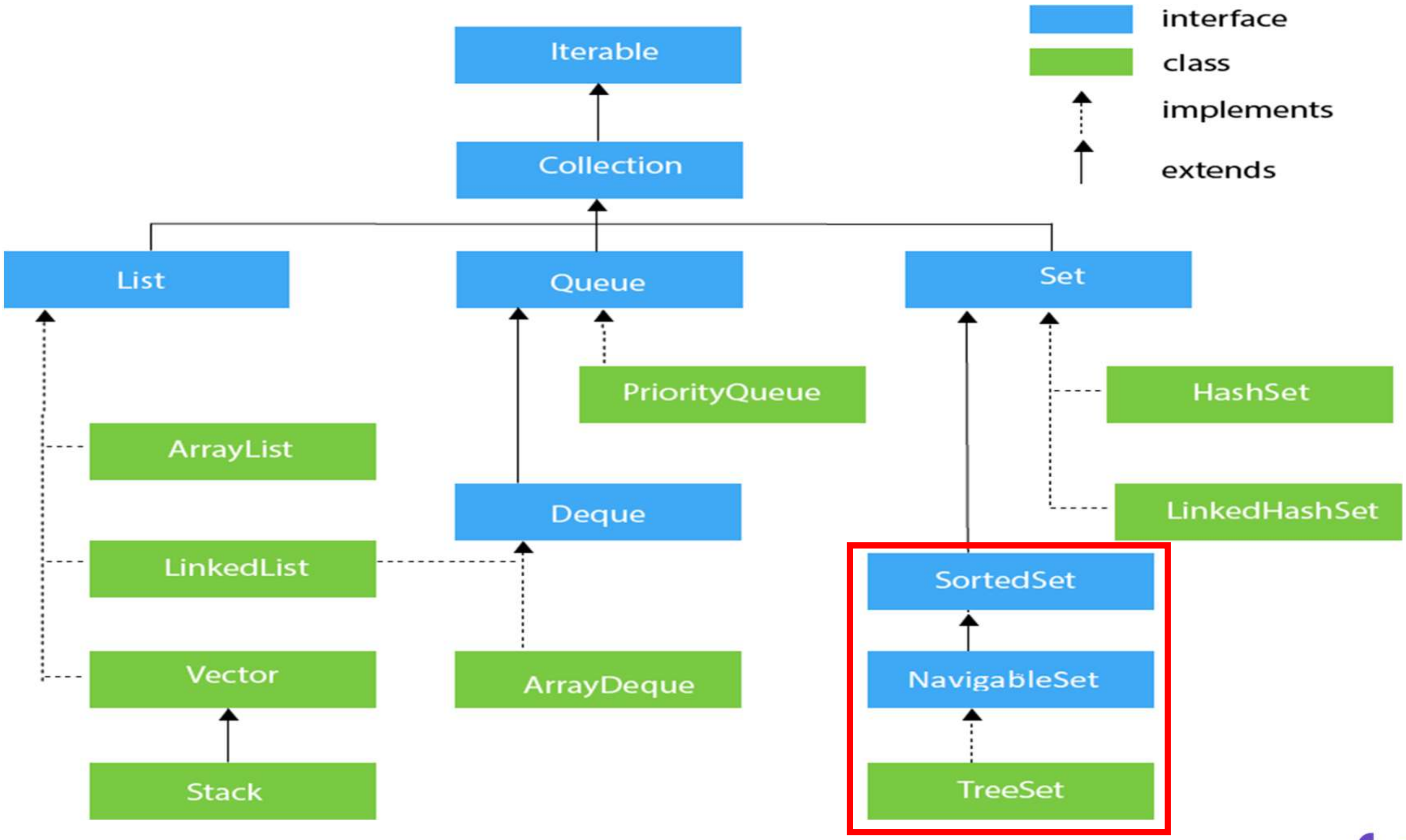
The SortedSet Interface

- The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order.
- **SortedSet** is a generic interface that has this declaration:

  interface SortedSet<E>

  Here, **E** specifies the type of objects that the set will hold.
- In addition to those methods provided by **Set**, the **SortedSet** interface declares the methods summarized

The SortedSet Interface



Collections in Java

The SortedSet Interface

- In addition to the methods defined by **Collection**, **SortedSet** defines some of its own

Method	Description
comparator()	Returns the **comparator** which is used to order the elements in the given set. Also returns null if the given set uses the natural ordering of the element.
first()	Returns the **first element** from the current set.
headSet(E toElement)	Returns a **view of the portion of the given set** whose elements are strictly less than the toElement.
last()	Returns the **reverse order** view of the mapping which present in the map.
spliterator()	Returns a **key-value mapping which is associated with the least key** in the given map. Also, returns null if the map is empty.
subSet(E fromElement, E toElement)	Returns a key-value mapping which is associated with the greatest key which is less than or equal to the given key. Also, returns null if the map is empty.
tailSet(E fromElement)	Returns a view of the map whose keys are strictly less than the toKey.

The NavigableSet Interface

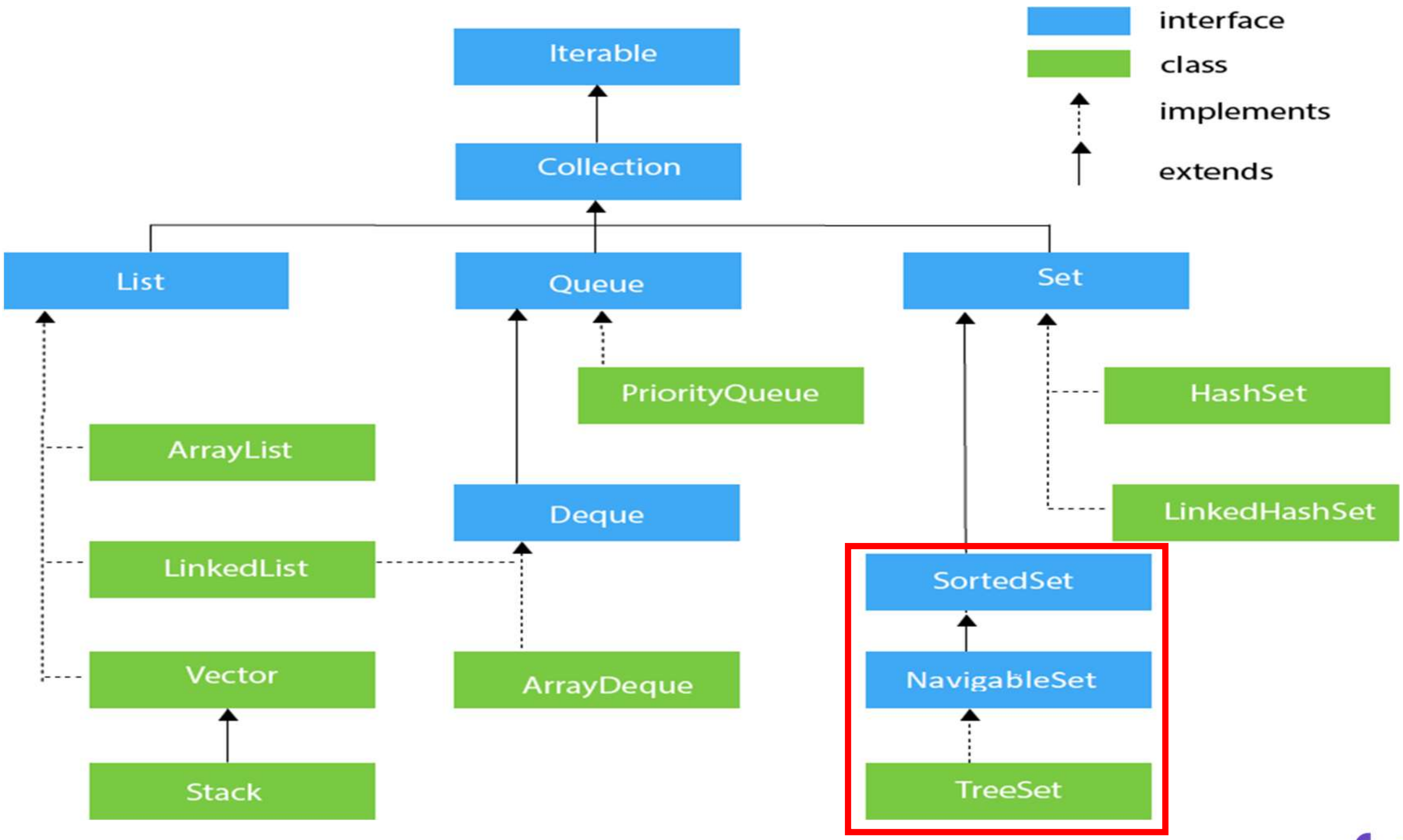
- The **NavigableSet** interface extends **SortedSet** and declares the behavior of a
- collection that supports the retrieval of elements based on the closest match to
- a given value or values.
- **NavigableSet** is a generic interface that has this declaration:

interface NavigableSet<E>

Here, **E** specifies the type of objects that the set will hold. In addition to the

Collections in Java

The NavigableSet Interface



The NavigableSet Interface

- Methods that it inherits from **SortedSet**, **NavigableSet** adds those summarized

Method	Description
E **ceiling**(E obj)	Searches the set for the smallest element e such that e >= obj. If such an element is found, it is returned. Otherwise, null is returned.
Iterator<E> **descendingIterator**()	It returns a reverse iterator.
NavigableSet<E> **descendingSet**()	Returns a NavigableSet that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E **floor**(E obj)	Searches the set for the largest element e such that e <= obj. If such an element is found, it is returned. Otherwise, null is returned.
NavigableSet<E> **headSet** (E upperBound, boolean incl)	Returns a NavigableSet that includes all elements from the invoking set that are less than upperBound. If incl is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set.
E **higher**(E obj)	Searches the set for the largest element e such that e > obj. If such an element is found, it is returned. Otherwise, null is returned.

The NavigableSet Interface

Method	Description
E **lower**(E obj)	Searches the set for the largest element e such that e < obj. If such an element is found, it is returned. Otherwise, null is returned.
E **pollFirst**()	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. null is returned if the set is empty.
E **pollLast**()	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. null is returned if the set is empty.
NavigableSet<E> **subset** (E lowerBound, boolean low_Include, E upperBound, boolean high_Include)	Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound and less than upperBound. If low_Include is true, then an element equal to lowerBound is included. If high_Include is true, then an element equal to upperBound is included. The resulting set is backed by the invoking set.
NavigableSet<E> **tailSet**(E lowerBound, boolean include)	Returns a NavigableSet that includes all elements from the invoking set that are greater than lowerBound. If incl is true, then an element equal to lowerBound is included. The resulting set is backed by the invoking set.

The TreeSet Class

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface.
- It creates a collection that uses a tree for storage.
- Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.
- **TreeSet** is a generic class that has this declaration:

  class TreeSet<E>

  Here, **E** specifies the type of objects that the set will hold.

The TreeSet Class – Example#1

```
/*This example demonstrate the TreeSet Class*/
import java.util.*;

public class TreeSetDemo {
    public static void main(String args[]){
        TreeSet<String> Ts=new TreeSet<String>();
        System.out.println("Size of the TreeSet is "+Ts.size());
        //adding elements
        Ts.add("C");
        Ts.add("B");
        Ts.add("A");
        Ts.add("E");
        Ts.add("F");
        Ts.add("D");
        System.out.println("Elements in the TreeSet "+Ts);
```

The TreeSet Class – Example#1

```
        System.out.println("Size of the TreeSet is "+Ts.size());
        Ts.remove("E");
        System.out.println("Elements in the TreeSet after remove "+Ts);
        System.out.println("Size of the TreeSet after the removal is "+Ts.size());
    }
}
```

Output:

```
Size of the TreeSet is 0
Elements in the TreeSet [A, B, C, D, E, F]
Size of the TreeSet is 6
Elements in the TreeSet after remove [A, B, C, D, F]
Size of the TreeSet after the removal is 5
```

The Queue Interface

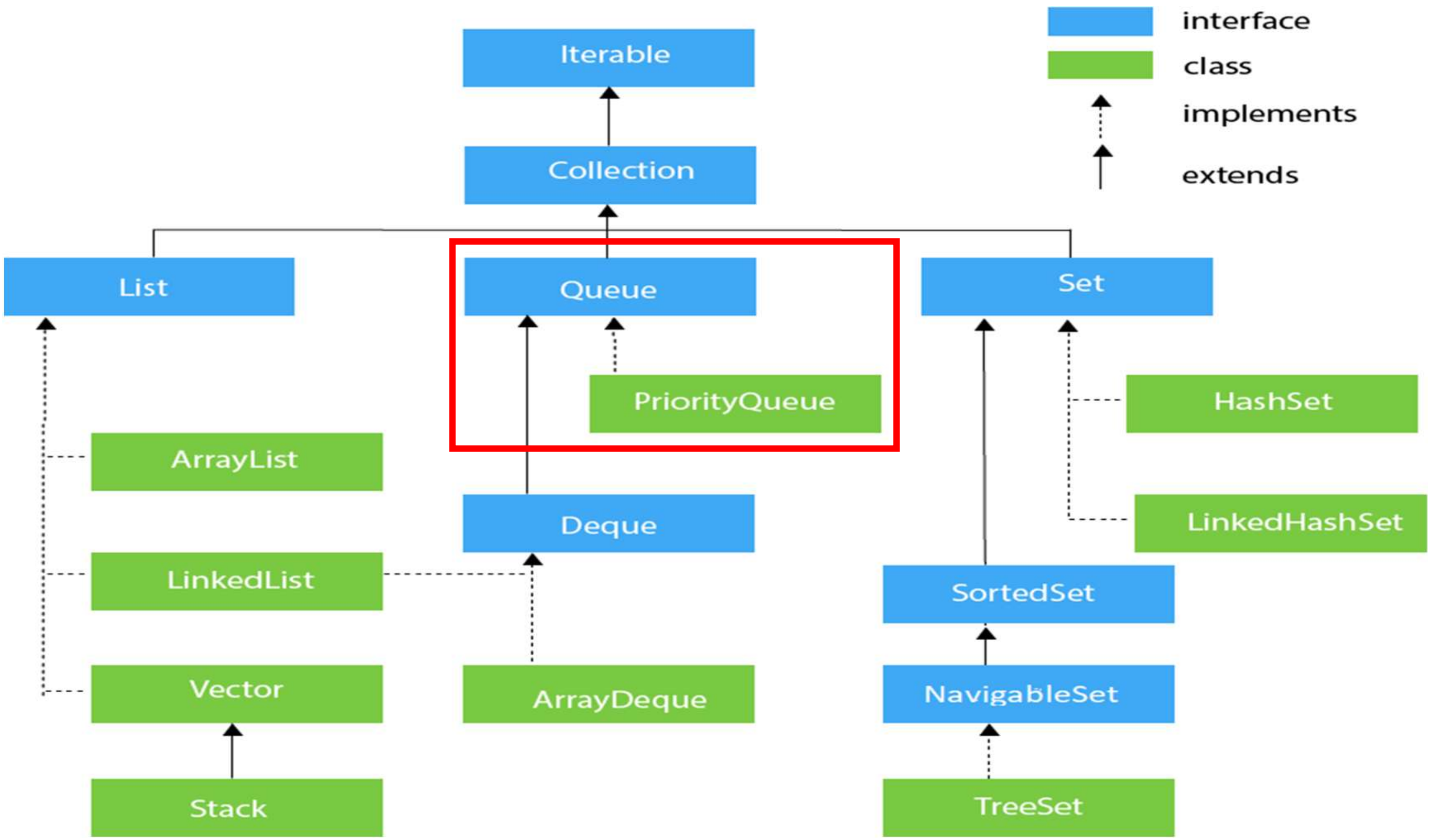
- The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list.
- There are types of queues in which the ordering is based upon other criteria. **Queue** is a generic interface that has this declaration:

  interface Queue<E>

  Here, **E** specifies the type of objects that the queue will hold.

Collections in Java

The Queue Interface



The Queue Interface

The methods declared by **Queue** are

Method	Description
boolean **offer**(object)	It is used to insert the specified element into 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.

The PriorityQueue Class

- **PriorityQueue** extends **AbstractQueue** and implements the **Queue** interface.
- It creates a queue that is prioritized based on the queue’s comparator.
- **PriorityQueue** is a generic class that has this declaration:

  class PriorityQueue<E>

  Here, **E** specifies the type of objects stored in the queue.
- **PriorityQueue**s are dynamic, growing as necessary.

The PriorityQueue Class – Example#1

```
/*This example demonstrate the PriorityQueue Class*/
import java.util.*;

public class PriorityQueueDemo {
    public static void main(String args[]){
        PriorityQueue<String> pq=new PriorityQueue<String>();
        System.out.println("Size of the PriorityQueue is "+pq.size());
        //adding elements
        pq.add("C");
        pq.add("B");
        pq.add("A");
        pq.add("E");
        pq.add("F");
        pq.add("D");
        System.out.println("Elements in the PriorityQueue "+pq);
```

The PriorityQueue Class – Example#1

```
        System.out.println("Size of the PriorityQueue is "+pq.size());
        pq.remove("E");
        pq.poll();
        System.out.println("Elements in the PriorityQueue after remove "+pq);
        System.out.println("Size of the PriorityQueue after the removal is "+pq.size());
    }
}
```

Output:

```
Size of the PriorityQueue is 0
Elements in the PriorityQueue [A, C, B, E, F, D]
Size of the PriorityQueue is 6
Elements in the PriorityQueue after remove [B, C, F, D]
Size of the PriorityQueue after the removal is 4
```

The DeQue Interface

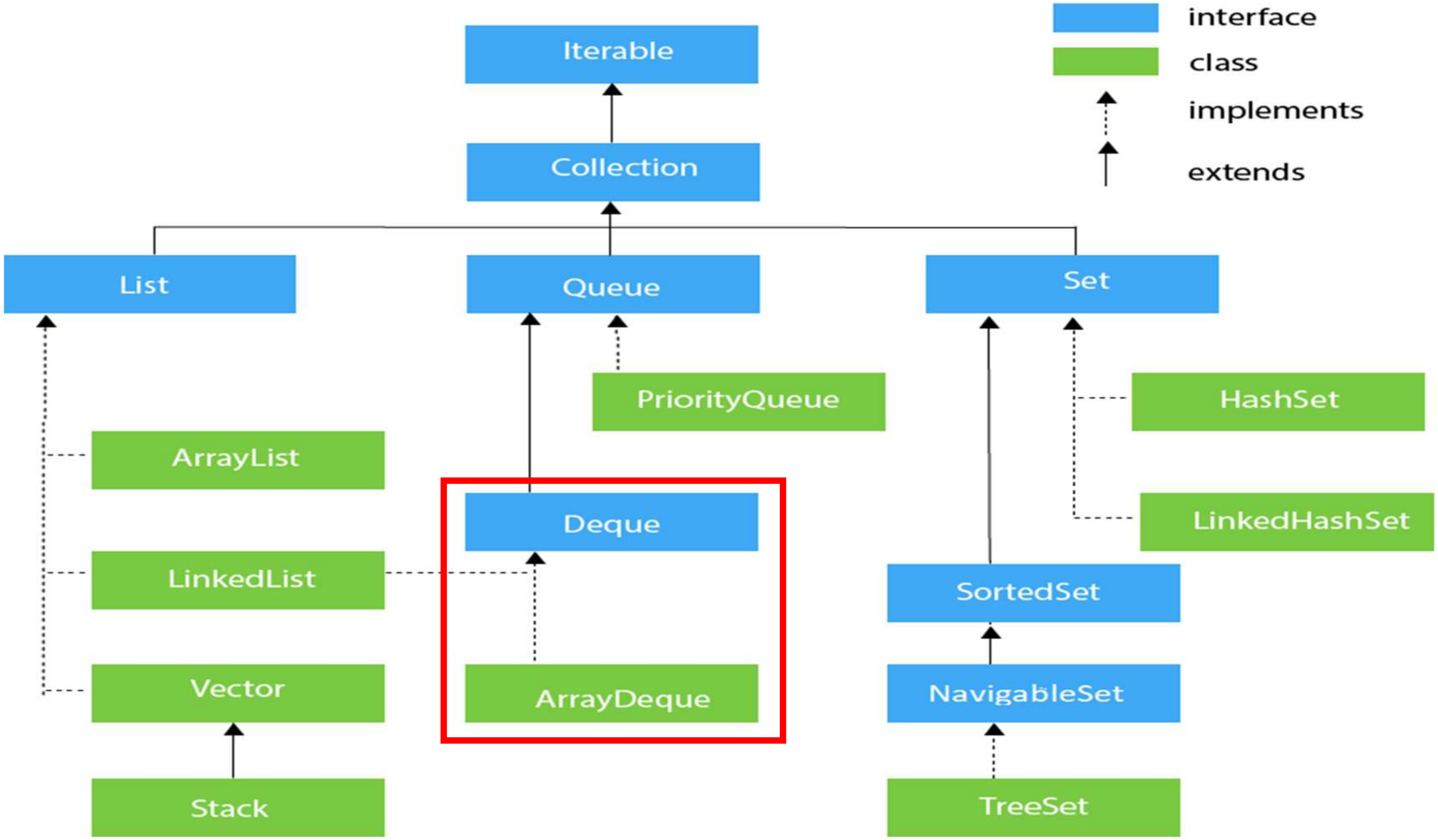
- The **Deque** interface extends **Queue** and declares the behavior of a doubleended queue.
- Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.
- **Deque** is a generic interface that has this declaration:

  interface Deque<E>

  Here, **E** specifies the type of objects that the deque will hold.

Collections in Java

The DeQue Interface



The DeQue Interface

- In addition to the methods that it inherits from **Queue**, **Deque** adds those methods summarized

Method	Description
addFirst(E e)	Inserts the specified element at the front of the deque.
addLast(E e)	Inserts the specified element at the end of the deque.
descendingIterator()	Returns an iterator over the elements in reverse sequential order.
element()	Retrieves the head of the queue represented by the deque.
getFirst()	Retrieves but does not remove the first element of the deque.
getLast()	Retrieves but does not remove the last element of the deque.
iterator()	Returns an iterator over the element in the deque in a proper sequence.
offer(E e)	Inserts the specified element into the deque, returning true upon success and false if no space is available.
offerFirst()	Inserts the specified element at the front of the deque unless it violates the capacity restriction.
offerLast()	Inserts the specified element at the end of the deque unless it violates the capacity restriction.
peek()	Retrieves but does not move the head of the queue represented by the deque or may return null if the deque is empty.
remove()	Retrieves and remove the head of the queue represented by the deque.

The DeQue Interface

Method	Description
peekFirst()	Retrieves but does not move the first element of the deque or may return null if the deque is empty.
peekLast()	Retrieves but does not move the last element of the deque or may return null if the deque is empty.
poll()	Retrieves and remove the head of the queue represented by the deque or may return null if the deque is empty.
pollFirst()	Retrieves and remove the first element of the deque or may return null if the deque is empty.
pollLast()	Retrieves and remove the last element of the deque or may return null if the deque is empty.
pop()	Pops an element from the stack represented by the deque.
push()	Pushes an element onto the stack represented by the deque.
removeFirst()	Retrieves and remove the first element from the deque.
removeFirstOccurrence(Object o)	Remove the first occurrence of the element from the deque.
removeLast()	Retrieve and remove the last element from the deque.
removeLastOccurrence(Object o)	Remove the last occurrence of the element from the deque.

The ArrayDeque Class

- The **ArrayDeque** class extends **AbstractCollection** and implements the **Deque** interface.
- It adds no methods of its own.
- **ArrayDeque** creates a dynamic array and has no capacity restrictions.
- **ArrayDeque** is a generic class that has this declaration:

  class ArrayDeque<E>

  Here, **E** specifies the type of objects stored in the collection.

The ArrayDeque Class – Example#1

```
/*This example demonstrate the ArrayDeque Class*/
import java.util.*;

public class ArrayDequeDemo01 {
    public static void main(String args[]){
        ArrayDeque<String> ad=new ArrayDeque<String>();
        System.out.println("Size of the ArrayDeque is "+ad.size());
        ad.push("A");
        ad.push("B");
        ad.push("C");
        ad.push("D");
        System.out.println("Elements in the ArrayDeque "+ad);
        System.out.println("Size of the ArrayDeque is "+ad.size());
```

The ArrayDeque Class – Example#1

```
        ad.pop();
        System.out.println("Elements in the ArrayDeque after remove "+ad);
        System.out.println("Size of the ArrayDeque after the removal is "+ad.size());
    }
}
```

Output:

```
Size of the ArrayDeque is 0
Elements in the ArrayDeque [D, C, B, A]
Size of the ArrayDeque is 4
Elements in the ArrayDeque after remove [C, B, A]
Size of the ArrayDeque after the removal is 3
```

Collections in Java

The Map Interface

- The **Map** interface maps unique keys to values.
- A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object.
- After the value is stored, you can retrieve it by using its key.
- **Map** is generic and is declared as shown here:

interface Map<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The Map Interface

- The methods declared by **Map** are summarized

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).

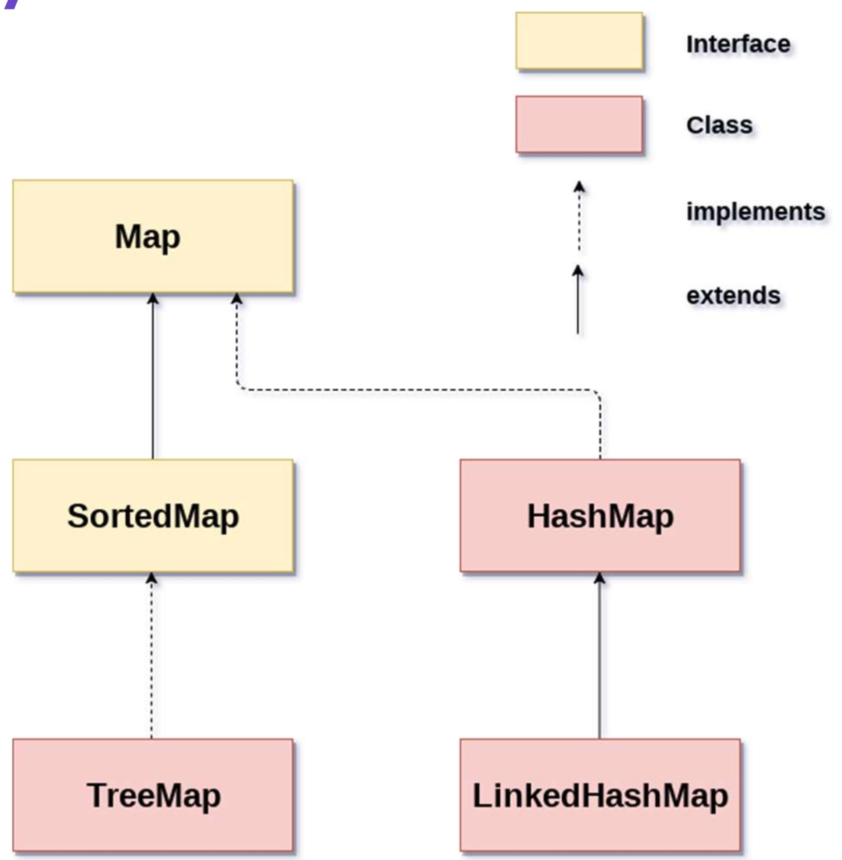
The Map Interface

Method	Description
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.

The Map Interface

Method	Description
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 **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.

The Map Hierarchy



The HashMap Class

- The **HashMap** class extends **AbstractMap** and implements the **Map** interface.
- It uses a hash table to store the map.
- This allows the execution time of **get( )** and **put( )** to remain constant even for large sets.
- **HashMap** is a generic class that has this declaration:

  class HashMap<K, V>

  Here, **K** specifies the type of keys, and **V** specifies the type of values

The HashMap Class – Example#1

```
/*This example demonstrate the HashMap Class*/
import java.util.*;

public class HashMapDemo {
    public static void main(String args[]){
        HashMap<String, Double> tm = new HashMap<String, Double>();
        System.out.println("Size of the HashMap is "+tm.size());
        tm.put("John Doe", 4343.43);
        tm.put("Tom Smith",145.23);
        tm.put("Jane Baker", 1450.78);
        tm.put("Ralph Smith",-18.76);
        System.out.println("Elements in the HashMap "+tm);
        System.out.println("Size of the HashMap is "+tm.size());
        Set<Map.Entry<String, Double>> set = tm.entrySet();
```

The HashMap Class – Example#1

```
        for(Map.Entry<String, Double> me:set) {
            System.out.print(me.getKey()+":");
            System.out.println(me.getValue());
        }
    }
}
```

Output:

```
Size of the HashMap is 0
Elements in the HashMap {John Doe=4343.43, Ralph Smith=-18.76, Tom Smith=145.23, Jane Baker=1450.78}
Size of the HashMap is 4
John Doe:4343.43
Ralph Smith:-18.76
Tom Smith:145.23
Jane Baker:1450.78
```

The LinkedHashMap Class

- **LinkedHashMap** extends **HashMap**.
- It maintains a linked list of the entries in the map, in the order in which they were inserted.
- When iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.
- You can also create a **LinkedHashMap** that returns its elements in the order in which they were last accessed.
- **LinkedHashMap** is a generic class that has this declaration:

```
class LinkedHashMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The LinkedHashMap Class – Example#1

```
/*This example demonstrate the LinkedHashMap Class*/
import java.util.*;

public class LinkedHashMapDemo {
        public static void main(String args[]){
                LinkedHashMap<String, Double> lhm = new LinkedHashMap<String, Double>();
                System.out.println("Size of the LinkedHashMap is "+lhm.size());
                lhm.put("John Doe", 4343.43);
                lhm.put("Tom Smith",145.23);
                lhm.put("Jane Baker", 1450.78);
                lhm.put("Ralph Smith",-18.76);
                System.out.println("Elements in the LinkedHashMap "+lhm);
                System.out.println("Size of the LinkedHashMap is "+lhm.size());
                Set<Map.Entry<String, Double>> set = lhm.entrySet();
```

The LinkedHashMap Class – Example#1

```
        for(Map.Entry<String, Double> me:set) {
            System.out.print(me.getKey()+":");
            System.out.println(me.getValue());
        }
    }
}
```

Output:

```
Size of the LinkedHashMap is 0
Elements in the LinkedHashMap {John Doe=4343.43, Tom Smith=145.23, Jane Baker=1450.78, Ralph Smith=-18.76}
Size of the LinkedHashMap is 4
John Doe:4343.43
Tom Smith:145.23
Jane Baker:1450.78
Ralph Smith:-18.76
```

The SortedMap Interface

- The **SortedMap** interface extends **Map**.
- It ensures that the entries are maintained in ascending order based on the keys.
- **SortedMap** is generic and is declared as shown here:

interface SortedMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

Collections in Java

The SortedMap Interface

- The methods declared by **SortedMap** are summarized

Method	Description
Comparator<? super K> **comparator**()	Returns the comparator used to order the keys in this map, or null if this map uses the **natural ordering** of its keys.
K **firstKey**()	Returns the **first (lowest) key** currently in this map.
SortedMap<K,V> **headMap**(K toKey)	Returns a view of the **portion of this map** whose keys are strictly **less than toKey**.
K **lastKey**()	Returns the **last (highest) key** currently in this map.
SortedMap<K,V> **subMap**(**K** fromKey, **K** toKey)	Returns a view of the **portion of this map** whose keys range from **fromKey**, inclusive, **to toKey**, exclusive.
SortedMap<K,V> **tailMap**(**K** fromKey)	Returns a view of the **portion of this map** whose keys are **greater than or equal to fromKey.**

The TreeMap Class

- The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface.
- It creates maps stored in a tree structure.
- A **TreeMap** provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval.
- Unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.
- **TreeMap** is a generic class that has this declaration:

  class TreeMap<K, V>

  Here, **K** specifies the type of keys, and **V** specifies the type of values.

The TreeMap Class – Example#1

```
/*This example demonstrate the TreeMap Class*/
import java.util.*;

public class TreeMapDemo {
    public static void main(String args[]){
        TreeMap<String, Double> tm = new TreeMap<String, Double>();
        System.out.println("Size of the TreeMap is "+tm.size());
        tm.put("John Doe", 4343.43);
        tm.put("Tom Smith",145.23);
        tm.put("Jane Baker", 1450.78);
        tm.put("Ralph Smith",-18.76);
        System.out.println("Elements in the TreeMap "+tm);
        System.out.println("Size of the TreeMap is "+tm.size());
        Set<Map.Entry<String, Double>> set = tm.entrySet();
```

The TreeMap Class – Example#1

```
        for(Map.Entry<String, Double> me:set) {
            System.out.print(me.getKey()+":");
            System.out.println(me.getValue());
        }
    }
}
```

Output:

```
Size of the TreeMap is 0
Elements in the TreeMap {Jane Baker=1450.78, John Doe=4343.43, Ralph Smith=-18.76, Tom Smith=145.23}
Size of the TreeMap is 4
Jane Baker:1450.78
John Doe:4343.43
Ralph Smith:-18.76
Tom Smith:145.23
```

Accessing Collections using Iterator

- Iterator is an object that implements either the **Iterator** or the **ListIterator interface**.
- **Iterator** enables you to cycle through a collection, obtaining or removing elements.
- **ListIterator** extends Iterator to allow bidirectional traversal of a list, and the modification of elements.
- Iterator and ListIterator are generic interfaces which are declared as shown here:

interface Iterator <E>

interface ListIterator <E>

Here, E specifies the type of objects being iterated.

The Iterator Interface

- The methods declared by **Iterator** are summarized

Method	Description
default void **forEachRemaining**(**Consumer** <? super **E**> action)	Performs the given action for each remaining element until all elements have been processed or the action throws an exception.
Boolean **hasNext**()	Returns true if the iteration has more elements.
E next()	Returns the next element in the iteration.
default void **remove**()	Removes from the underlying collection the last element returned by this iterator (optional operation).

The ListIterator Interface

- The methods declared by **ListIterator** are summarized

Method	Description
void **add**(**E** e)	**Inserts the specified element** into the list (optional operation).
boolean **hasNext**()	Returns true if this list iterator **has more elements** when traversing the list in the **forward direction**.
boolean **hasPrevious**()	Returns true if this list iterator **has more elements** when traversing the list in the **reverse direction**.
E next()	Returns the **next element** in the list and advances the cursor position.
int **nextIndex**()	Returns the index of the element that would be returned by a subsequent call to **next()**.
E previous()	Returns the **previous element** in the list and moves the cursor position backwards.
int **previousIndex**()	Returns the index of the element that would be returned by a subsequent call to **previous()**.
Void **remove**()	Removes from the list the last element that was returned by **next()** or **previous()** (optional operation).
Void **set**(**E** e)	Replaces the last element returned by **next()** or **previous()** with the specified element (optional operation).

Accessing Collections using Iterator – Example#1

```
/*This example demonstrate the Collections using Iterator*/
import java.util.*;

public class IteratorDemo01 {
    public static void main(String args[]) {
        ArrayList <String> Arr = new ArrayList<String>(); //Create an Arraylist
        System.out.println("Initial Size of Array List is "+Arr.size());
        Arr.add("C");
        Arr.add("A");
        Arr.add("E");
        Arr.add("B");
        Arr.add("D");
        Arr.add("F");
        Arr.add(1, "G");
        System.out.println("After Insert the Size of Array List is "+Arr.size());
```

Accessing Collections using Iterator – Example#1

```
System.out.println("Contents of ArrayList using Iterator");
Iterator<String> itr = Arr.iterator(); //Iterator
while(itr.hasNext()) {
        String element = itr.next();
        System.out.print(element+" ");
}
System.out.println();
ListIterator<String> litr = Arr.listIterator(); //ListIterator
while(litr.hasNext()) {
        String element = litr.next();
        litr.set(element+"+");
}
System.out.println("Modified Contents of ArrayList using Iterator");
itr = Arr.iterator(); //Iterator
```

Accessing Collections using Iterator – Example#1

```
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element+" ");
        }
        System.out.println();
        System.out.println("Modified Contents of ArrayList in Backward using ListIterator");
        while(litr.hasPrevious()) {
            String element = litr.previous();
            System.out.print(element+" ");
        }
    }
}
```

Accessing Collections using Iterator – Example#1

Output:

```
Initial Size of Array List is 0
After Insert the Size of Array List is 7
Contents of ArrayList using Iterator
C G A E B D F
Modified Contents of ArrayList using Iterator
C+ G+ A+ E+ B+ D+ F+
Modified Contents of ArrayList in Backward using ListIterator
F+ D+ B+ E+ A+ G+ C+
```

Storing User defined Classes in Collections - Example

```
/*This example demonstrate the Collections with User defined class*/
import java.util.*;
class Address{
      private String name;
      private String street;
      private String city;
      private String state;
      private String code;
      Address(String n, String s,String c,String st,String cd){
            name = n;
            street = s;
            city = c;
            state = st;
            code = cd;
      }
```

Storing User defined Classes in Collections - Example

```
    public String toString() {
        return name +" "+street+" "+city+" "+state+" "+code;
    } }
public class UserDefinedCollectionsDemo01 {
    public static void main(String args[]) {
        LinkedList<Address> lst = new LinkedList<Address>();
        //Add element to the linked list
        lst.add(new Address("John Doe","11 Oak Ave","Urbana","IL","61801"));
        lst.add(new Address("Ralph Baker","1142 Maple Lane","Mahomet","IL","61853"));
        lst.add(new Address("Tom Carlton","867 Elm st","Champaign","IL","61820"));
        //Display the Details
        for(Address element:lst) {
                System.out.println(element);
        }
    } }
```

Storing User defined Classes in Collections - Example

Output:

```
John Doe 11 Oak Ave Urbana IL 61801
Ralph Baker 1142 Maple Lane Mahomet IL 61853
Tom Carlton 867 Elm st Champaign IL 61820
```

Collections in Java

Quiz

1) What is Collection in Java?

a) A group of Classes

b) A group of objects

c) A group of interfaces

d) None of the above

b) A group of objects

Collections in Java

Quiz

2) Which of the following is not in the Collections in Java?

- a) Array
- b) Vector
- c) Stack
- d) HashSet

a) Array

Quiz

3) Which of the following is the interface?

a) ArrayList

b) HashSet

c) Queue

d) TreeMap

c) Queue

Collections in Java

Quiz

4) The Dictionary class provides the capability to store

a) key

b) key-value pair

c) value

d) None of these

b) key-value pair

Collections in Java

Quiz

5) Which of the following classes are used to avoid duplicates?

a) ArrayList

b) HashSet

c) LinkedList

d) LinkedHashSet

b) HashSet & d) LinkedHashSet

Quiz

6) In which of the following package, are all of the collection classes present?

a) java.net

b) java.lang

c) java.awt

d) java.util

d) java.util

Collections in Java

Quiz

7) Which of the following interface is not a part of Java’s collection framework?

a) SortedList

b) Set

c) List

d) SortedMap

a) SortedList

Collections in Java

Quiz

8) Which of these interface handle sequences?

a) Set

b) Comparator

c) Collection

d) List

d) List

Quiz

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

a) clear()

b) reset()

c) delete()

d) refresh()

a) clear()

Quiz

10) What is the output of the following code

```
import java.util.*;
public class Test {
    public static void main(String args[]) {
        ArrayList <Integer> al = new ArrayList<Integer>();
        for (int i = 5; i > 0; i--)
            al.add(i);
        for(Integer ele:al) {
            System.out.print(ele+" ");
        }
    }
}
```

a) 12345

b) 54321

c) 13579

d) 02468

b) 54321