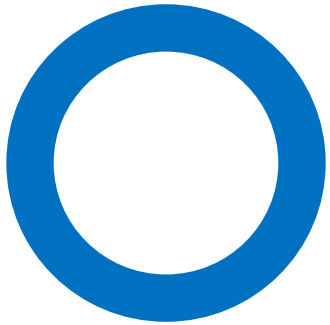
Generics Collections

Java Programming 1

Hồ Tuấn Thanh

htthanh@fit.hcmus.edu.vn

Java Generics



Introduction

- It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

- Java Generic methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Introduction

- Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

- Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.

Generic Methods

- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.

- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Live Demo

```
public class GenericMethodTest {
   // generic method printArray
   public static < E > void printArray( E[] inputArray ) {
      // Display array elements
      for(E element : inputArray) {
         System.out.printf("%s ", element);
      }
      System.out.println();
   }

   public static void main(String args[]) {
      // Create arrays of Integer, Double and Character
      Integer[] intArray = { 1, 2, 3, 4, 5 };
      Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
      Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

      System.out.println("Array integerArray contains:");
      printArray(intArray);   // pass an Integer array

      System.out.println("\nArray doubleArray contains:");
      printArray(doubleArray);   // pass a Double array

      System.out.println("\nArray characterArray contains:");
      printArray(charArray);   // pass a Character array
   }
}
```

Output

```
Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

Bounded Type Parameters

- There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter.
  - For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.
- This is what bounded type parameters are for.
- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Live Demo

```
public class MaximumTest {
   // determines the largest of three Comparable objects

   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x;   // assume x is initially the largest

      if(y.compareTo(max) > 0) {
         max = y;   // y is the largest so far
      }

      if(z.compareTo(max) > 0) {
         max = z;   // z is the largest now
      }
      return max;   // returns the largest object
   }

   public static void main(String args[]) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
         3, 4, 5, maximum( 3, 4, 5 ));

      System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

      System.out.printf("Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }
}
```

Output

```
Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear
```

Generic Classes

Live Demo

```
public class Box<T> {
   private T t;

   public void add(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }

   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}
```

<T> can not be a primitive type

```
// A Generic method example
static <T> void genericDisplay (T element)
{
    System.out.println(element.getClass().getName() +
                       " = " + element);
}

// Driver method
public static void main(String[] args)
{
     // Calling generic method with Integer argument
    genericDisplay(11);

    // Calling generic method with String argument
    genericDisplay("GeeksForGeeks");

    // Calling generic method with double argument
    genericDisplay(1.0);
}
```

```
java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0
```

Advantages of Generics

- Code Reuse: We can write a method/class/interface once and use for any type we want.

Advantages of Generics

- Type Safety : Generics make errors to appear compile time than at run time

- It’s always better to know problems in your code at compile time rather than making your code fail at run time.

Advantages of Generics

```
public static void main(String[] args)
{
    // Creatinga an ArrayList without any type specified
    ArrayList al = new ArrayList();

    al.add("Sachin");
    al.add("Rahul");
    al.add(10); // Compiler allows this

    String s1 = (String)al.get(0);
    String s2 = (String)al.get(1);

    // Causes Runtime Exception
    String s3 = (String)al.get(2);
}
```

Output :

```
Exception in thread "main" java.lang.ClassCastException:
   java.lang.Integer cannot be cast to java.lang.String
    at Test.main(Test.java:19)
```

Advantages of Generics

```
public static void main(String[] args)
{
    // Creating a an ArrayList with String specified
    ArrayList <String> al = new ArrayList<String> ();

    al.add("Sachin");
    al.add("Rahul");

    // Now Compiler doesn't allow this
    al.add(10);

    String s1 = (String)al.get(0);
    String s2 = (String)al.get(1);
    String s3 = (String)al.get(2);
}
```

Output:

```
15: error: no suitable method found for add(int)
        al.add(10);
          ^
```

Advantages of Generics

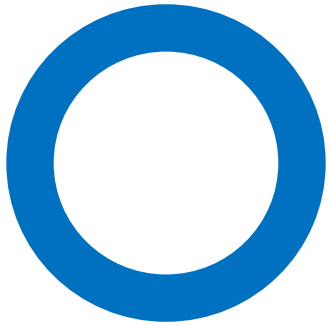
- Individual Type Casting is not needed

```
public static void main(String[] args)
{
    // Creating a an ArrayList with String specified
    ArrayList <String> al = new ArrayList<String> ();

    al.add("Sachin");
    al.add("Rahul");

    // Typecasting is not needed
    String s1 = al.get(0);
    String s2 = al.get(1);
}
```

Advantages of Generics

- Generics promotes code reusability.
- Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

Java Collections



Introduction

- Any group of individual objects which are represented as a single unit is known as the collection of the objects.

- In Java, a separate framework named the “Collection Framework” has been defined in JDK 1.2 which holds all the collection classes and interface in it.

Introduction

- The Collection interface (java.util.Collection) and Map interface (java.util.Map) are the two main “root” interfaces of Java collection classes.

Need for a Separate Collection Framework

- Before Collection Framework(or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables.

- All of these collections had no common interface.

Need for a Separate Collection Framework

- Therefore, though the main aim of all the collections are same, the implementation of all these collections were defined independently and had no correlation among them.

- And also, its very difficult for the users to remember all the different methods, syntax and constructors present in every collection class.

```
public static void main(String[] args)
{
    // Creating instances of the array,
    // vector and hashtable
    int arr[] = new int[] { 1, 2, 3, 4 };
    Vector<Integer> v = new Vector();
    Hashtable<Integer, String> h
        = new Hashtable();

    // Adding the elements into the
    // vector
    v.addElement(1);
    v.addElement(2);

    // Adding the element into the
    // hashtable
    h.put(1, "geeks");
    h.put(2, "4geeks");

    // Array instance creation requires [],
    // while Vector and hastable require ()
    // Vector element insertion requires addElement(),
    // but hashtable element insertion requires put()

    // Accessing the first element of the
    // array, vector and hashtable
    System.out.println(arr[0]);
    System.out.println(v.elementAt(0));
    System.out.println(h.get(1));

    // Array elements are accessed using [],
    // vector elements using elementAt()
    // and hashtable elements using get()
}
```

Need for a Separate Collection Framework

- As we can observe, none of these collections(Array, Vector or Hashtable) implements a standard member access interface, it was very difficult for programmers to write algorithms that can work for all kinds of Collections.

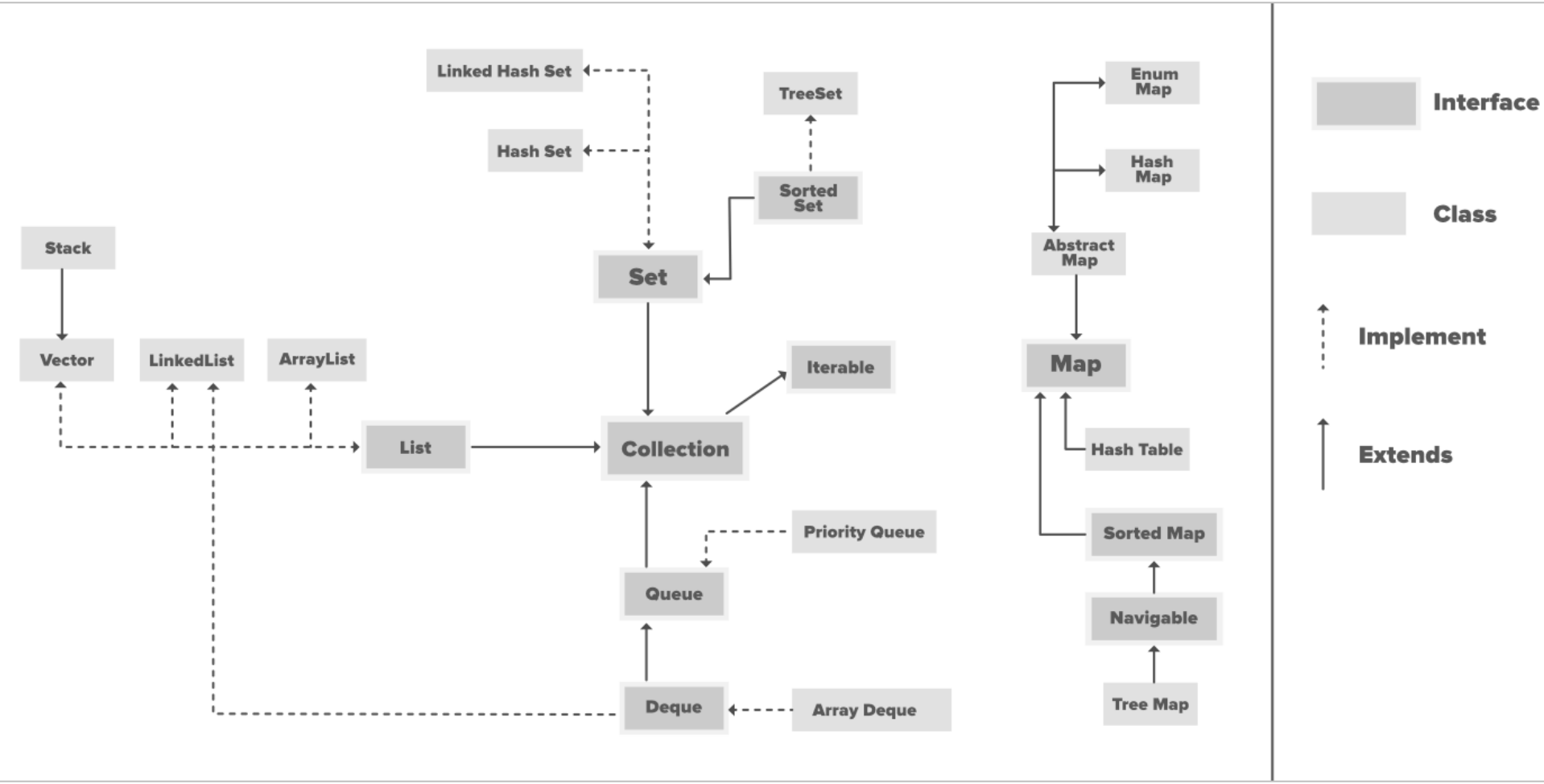
Need for a Separate Collection Framework

- Another drawback is that most of the ‘Vector’ methods are final, meaning we cannot extend the ’Vector’ class to implement a similar kind of Collection.

Need for a Separate Collection Framework

- Therefore, Java developers decided to come up with a common interface to deal with the above-mentioned problems and introduced the Collection Framework in JDK 1.2 post which both, legacy Vectors and Hashtables were modified to conform to the Collection Framework.

Hierarchy of the Collection Framework



Collection Interface

add(Object)	This method is used to add an object to the collection.
addAll(Collection c)	This method adds all the elements in the given collection to this collection.
clear()	This method removes all of the elements from this collection.
contains(Object o)	This method returns true if the collection contains the specified element.
containsAll(Collection c)	This method returns true if the collection contains all of the elements in the given collection.

Collection Interface

equals(Object o)	This method compares the specified object with this collection for equality.
hashCode()	This method is used to return the hash code value for this collection.
isEmpty()	This method returns true if this collection contains no elements.
iterator()	This method returns an iterator over the elements in this collection.
max()	This method is used to return the maximium value present in the collection.

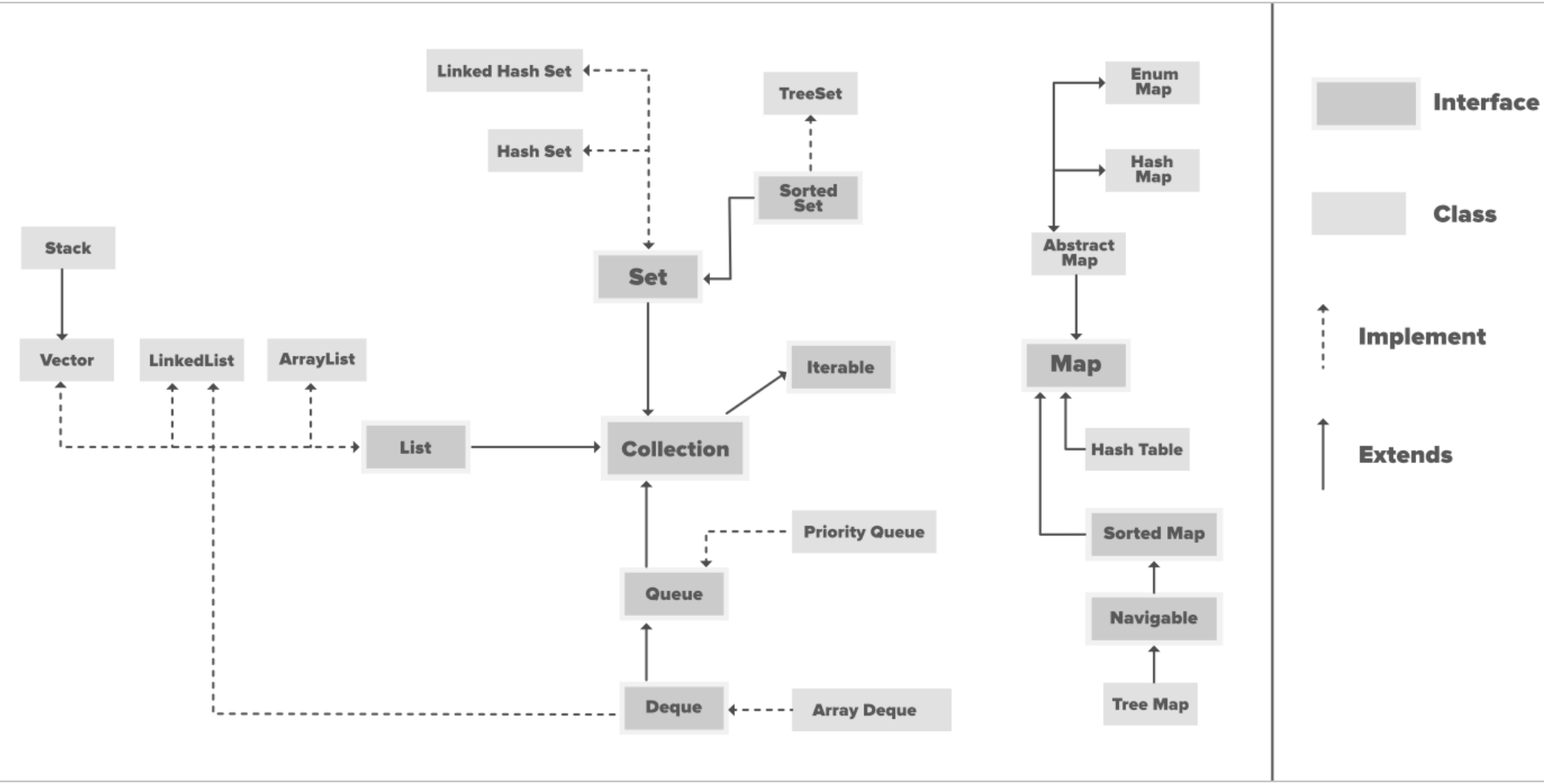
Collection Interface

parallelStream()	This method returns a parallel Stream with this collection as its source.
remove(Object o)	This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.
removeAll(Collection c)	This method is used to remove all the objects mentioned in the given collection from the collection.
removeIf(Predicate filter)	This method is used to removes all the elements of this collection that satisfy the given predicate.
retainAll(Collection c)	This method is used to retains only the elements in this collection that are contained in the specified collection.

Collection Interface

size()	This method is used to return the number of elements in the collection.
spliterator()	This method is used to create a Spliterator over the elements in this collection.
stream()	This method is used to return a sequential Stream with this collection as its source.
toArray()	This method is used to return an array containing all of the elements in this collection.

Hierarchy of the Collection Framework



List interface

- This is a child interface of the collection interface.
- This interface is dedicated to the data of the list type in which we can store all the ordered collection of the objects.
- This also allows duplicate data to be present in it.
- This list interface is implemented by various classes like ArrayList, Vector, Stack, etc.
- Since all the subclasses implement the list, we can instantiate a list object with any of these classes.

ArrayList Class

- ArrayList provides us with dynamic arrays in Java.
- Though, it may be **slower** than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
- The size of an ArrayList is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList **can not** be used for **primitive types**, like int, char, etc.
- We will need a wrapper class for such cases

```
public static void main(String[] args)
{

    // Declaring the ArrayList with
    // initial size n
    ArrayList<Integer> al
        = new ArrayList<Integer>();

    // Appending new elements at
    // the end of the list
    for (int i = 1; i <= 5; i++)
        al.add(i);

    // Printing elements
    System.out.println(al);

    // Remove element at index 3
    al.remove(3);

    // Displaying the ArrayList
    // after deletion
    System.out.println(al);

    // Printing elements one by one
    for (int i = 0; i < al.size(); i++)
        System.out.print(al.get(i) + " ");
}
```

LinkedList Class

- LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part.

- The elements are linked using pointers and addresses. Each element is known as a node.

```
public static void main(String[] args)
{

    // Declaring the LinkedList
    LinkedList<Integer> ll
        = new LinkedList<Integer>();

    // Appending new elements at
    // the end of the list
    for (int i = 1; i <= 5; i++)
        ll.add(i);

    // Printing elements
    System.out.println(ll);

    // Remove element at index 3
    ll.remove(3);

    // Displaying the List
    // after deletion
    System.out.println(ll);

    // Printing elements one by one
    for (int i = 0; i < ll.size(); i++)
        System.out.print(ll.get(i) + " ");
}
```

Vector Class

- A vector provides us with dynamic arrays in Java.
- Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.
- This is identical to ArrayList in terms of implementation.
- However, the primary difference between a vector and an ArrayList is that a **Vector is synchronized** and an ArrayList is non-synchronized.

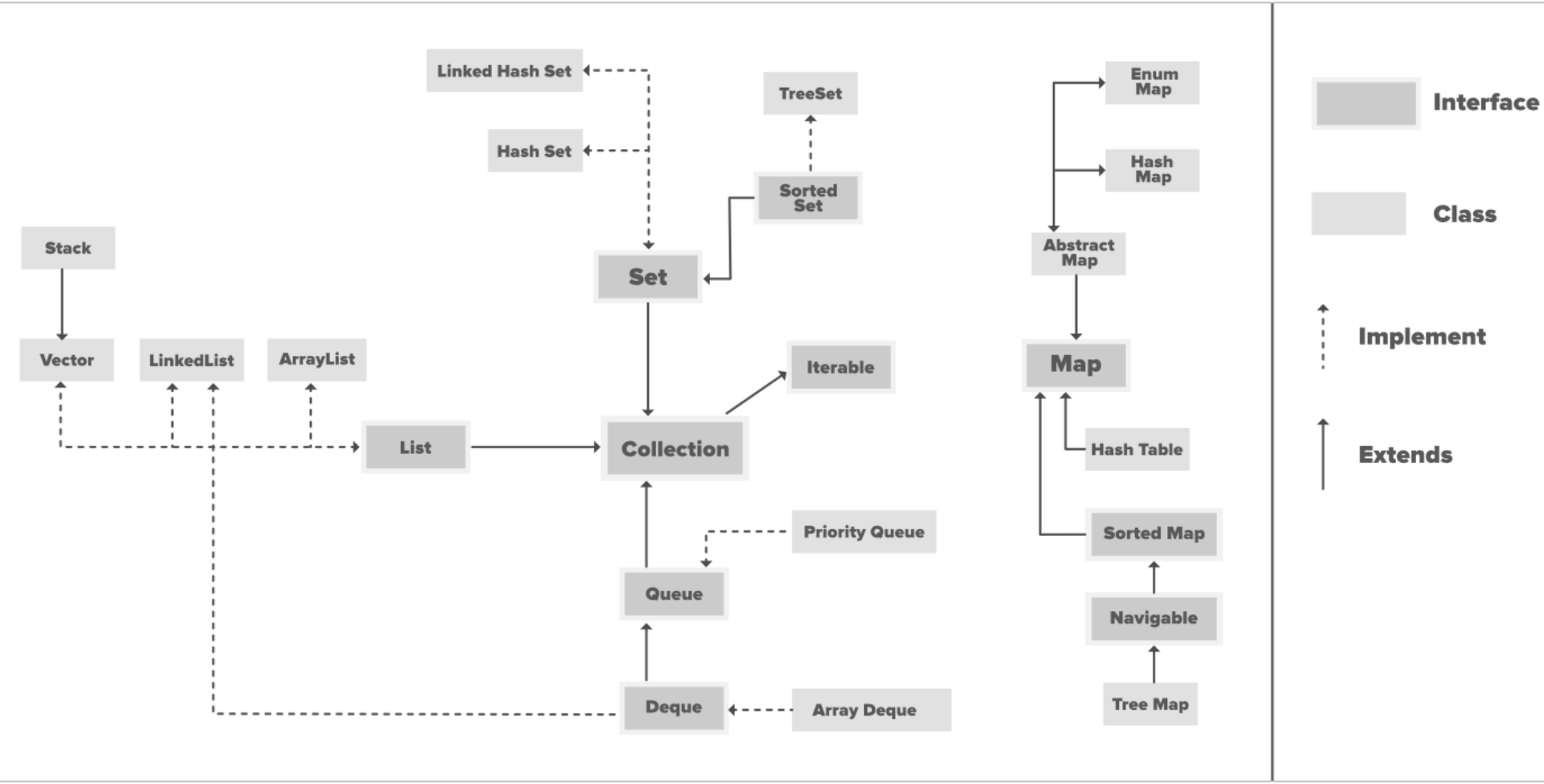
```
public static void main(String[] args)
{

    // Declaring the Vector
    Vector<Integer> v
        = new Vector<Integer>();

    // Appending new elements at
    // the end of the list
    for (int i = 1; i <= 5; i++)
        v.add(i);

    // Printing elements
    System.out.println(v);

    // Remove element at index 3
    v.remove(3);

    // Displaying the Vector
    // after deletion
    System.out.println(v);

    // Printing elements one by one
    for (int i = 0; i < v.size(); i++)
        System.out.print(v.get(i) + " ");
}
```

Stack Class

- Stack class models and implements the Stack data structure.
- The class is based on the basic principle of **last-in-first-out**.
- In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek.
- The class can also be referred to as the subclass of Vector.

```
public static void main(String args[])
{
    Stack<String> stack = new Stack<String>();
    stack.push("Geeks");
    stack.push("For");
    stack.push("Geeks");
    stack.push("Geeks");

    // Iterator for the stack
    Iterator<String> itr
        = stack.iterator();

    // Printing the stack
    while (itr.hasNext()) {
        System.out.print(itr.next() + " ");
    }

    System.out.println();

    stack.pop();

    // Iterator for the stack
    itr
        = stack.iterator();

    // Printing the stack
    while (itr.hasNext()) {
        System.out.print(itr.next() + " ");
    }
}
```

Hierarchy of the Collection Framework



Queue Interface

- As the name suggests, a queue interface maintains the FIFO(First In First Out) order similar to a real-world queue line.
- This interface is dedicated to storing all the elements where the order of the elements matter.
- There are various classes like PriorityQueue, Deque, ArrayDeque, etc. Since all these subclasses implement the queue, we can instantiate a queue object with any of these classes

Priority Queue Class

- A PriorityQueue is used when the objects are supposed to be processed based on the priority.
- It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases.
- The PriorityQueue is based on the priority heap.
- The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

```
public static void main(String args[])
{
    // Creating empty priority queue
    PriorityQueue<Integer> pQueue
        = new PriorityQueue<Integer>();

    // Adding items to the pQueue using add()
    pQueue.add(10);
    pQueue.add(20);
    pQueue.add(15);

    // Printing the top element of PriorityQueue
    System.out.println(pQueue.peek());

    // Printing the top element and removing it
    // from the PriorityQueue container
    System.out.println(pQueue.poll());

    // Printing the top element again
    System.out.println(pQueue.peek());
}
```

Output:

```
10
10
15
```

Example of Comparator Interface

```
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                                String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                             " " + this.address;
    }
}
```

```
class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

class Sortbyname implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll name
    public int compare(Student a, Student b)
    {
        return a.name.compareTo(b.name);
    }
}
```

```
public static void main (String[] args)
{
    ArrayList<Student> ar = new ArrayList<Student>();
    ar.add(new Student(111, "bbbb", "london"));
    ar.add(new Student(131, "aaaa", "nyc"));
    ar.add(new Student(121, "cccc", "jaipur"));

    System.out.println("Unsorted");
    for (int i=0; i<ar.size(); i++)
        System.out.println(ar.get(i));

    Collections.sort(ar, new Sortbyroll());

    System.out.println("\nSorted by rollno");
    for (int i=0; i<ar.size(); i++)
        System.out.println(ar.get(i));

    Collections.sort(ar, new Sortbyname());

    System.out.println("\nSorted by name");
    for (int i=0; i<ar.size(); i++)
        System.out.println(ar.get(i));
}
```

Output:

```
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc

Sorted by name
131 aaaa nyc
111 bbbb london
121 cccc jaipur
```

Deque Interface

- This is a very slight variation of the queue data structure.
- Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both the ends of the queue.
- This interface extends the queue interface.
- The class which implements this interface is ArrayDeque.
- Since this class implements the deque, we can instantiate a deque object with this class.

ArrayDeque Class

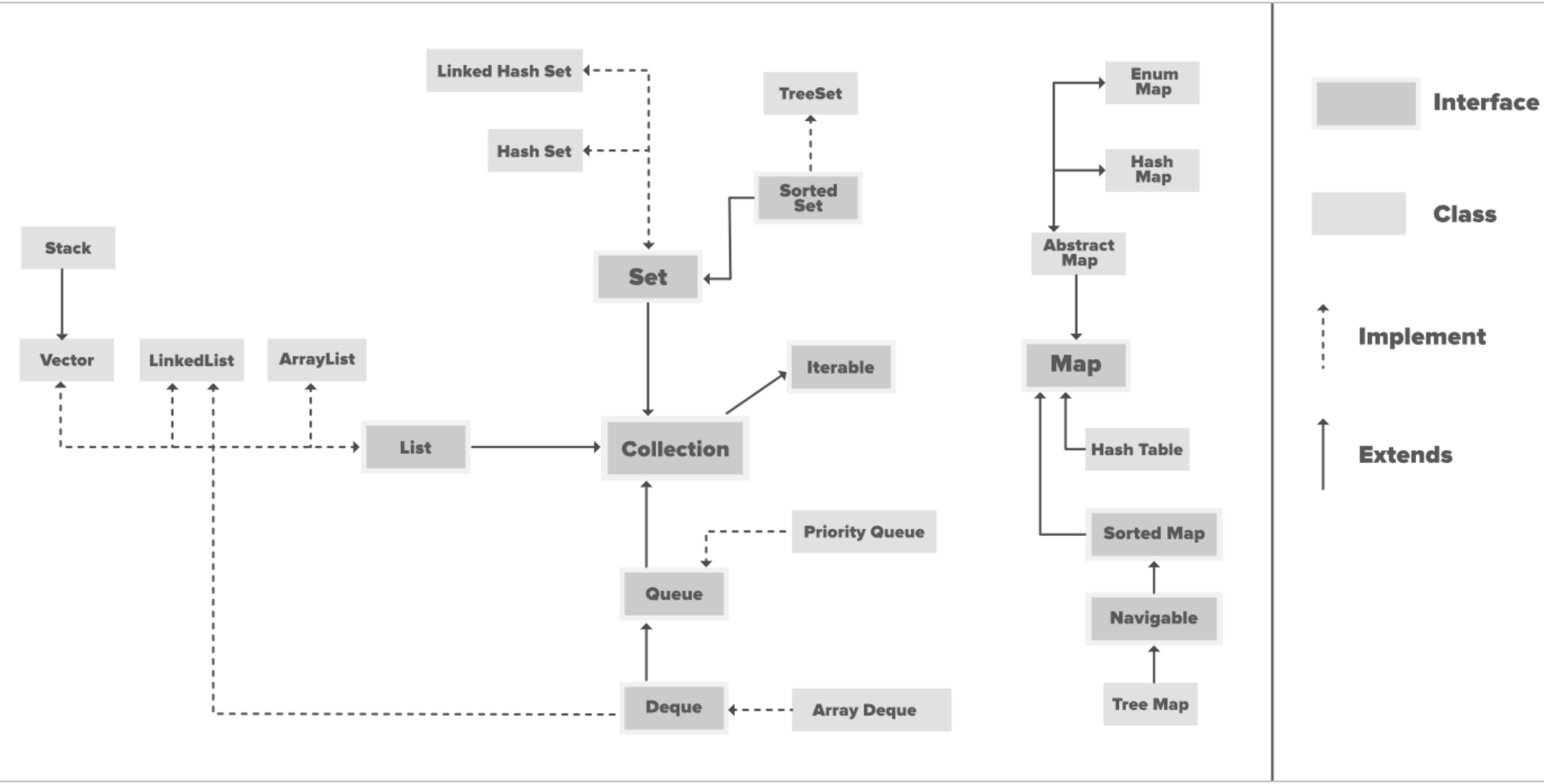
- ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array.
- This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.
- Array deques have no capacity restrictions and they grow as necessary to support usage.

```
public static void main(String[] args)
{
    // Initializing an deque
    ArrayDeque<Integer> de_que
        = new ArrayDeque<Integer>(10);

    // add() method to insert
    de_que.add(10);
    de_que.add(20);
    de_que.add(30);
    de_que.add(40);
    de_que.add(50);

    System.out.println(de_que);

    // clear() method
    de_que.clear();

    // addFirst() method to insert the
    // elements at the head
    de_que.addFirst(564);
    de_que.addFirst(291);

    // addLast() method to insert the
    // elements at the tail
    de_que.addLast(24);
    de_que.addLast(14);

    System.out.println(de_que);
}
```

Output:

```
[10, 20, 30, 40, 50]
[291, 564, 24, 14]
```

Hierarchy of the Collection Framework



Set Interface

- A set is an unordered collection of objects in which duplicate values cannot be stored.
- This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects.
- This set interface is implemented by various classes like HashSet, TreeSet, LinkedHashSet, etc.

HashSet Class

- The HashSet class is an inherent implementation of the hash table data structure.
- The objects that we insert into the HashSet **do not guarantee** to be inserted in the same **order**.
- The objects are inserted based on their hashcode.
- This class also allows the insertion of NULL elements.

```
public static void main(String args[])
{
    // Creating HashSet and
    // adding elements
    HashSet<String> hs = new HashSet<String>();

    hs.add("Geeks");
    hs.add("For");
    hs.add("Geeks");
    hs.add("Is");
    hs.add("Very helpful");

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

Output:

```
Very helpful
Geeks
For
Is
```

- A LinkedHashSet is very similar to a HashSet.

- The difference is that this uses a doubly linked list to store the data and **retains** the **ordering** of the elements.

```
public static void main(String args[])
{
    // Creating LinkedHashSet and
    // adding elements
    LinkedHashSet<String> lhs
        = new LinkedHashSet<String>();

    lhs.add("Geeks");
    lhs.add("For");
    lhs.add("Geeks");
    lhs.add("Is");
    lhs.add("Very helpful");

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

Output:

```
Geeks
For
Is
Very helpful
```

SortedSet Interface

- This interface is very similar to the set interface.
- The only difference is that this interface has extra methods that **maintain** the **ordering** of the elements.
- The sorted set interface extends the set interface and is used to handle the data which needs to be sorted.
- The class which implements this interface is TreeSet.

TreeSet Class

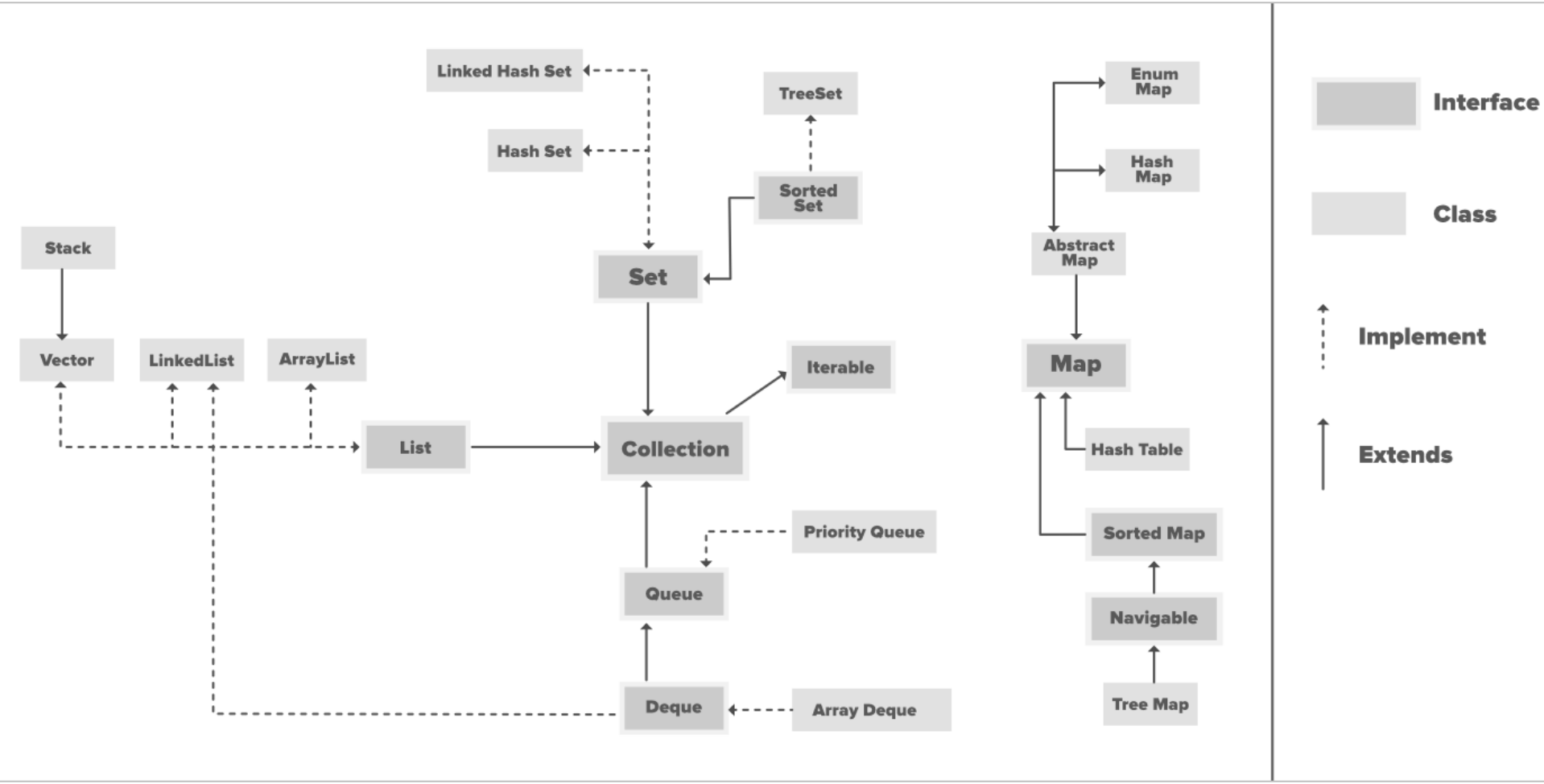
- The TreeSet class uses a Tree for storage.
- The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided.

```
public static void main(String args[])
{
    // Creating TreeSet and
    // adding elements
    TreeSet<String> ts
        = new TreeSet<String>();

    ts.add("Geeks");
    ts.add("For");
    ts.add("Geeks");
    ts.add("Is");
    ts.add("Very helpful");

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

Output:

```
For
Geeks
Is
Very helpful
```

Hierarchy of the Collection Framework



Map Interface

- A map is a data structure which supports the key-value pair mapping for the data.
- This interface doesn’t support duplicate keys because the same key cannot have multiple mappings.
- A map is useful if there is a data and we wish to perform operations on the basis of the key.
- This map interface is implemented by various classes like HashMap, TreeMap etc.

HashMap Class

- HashMap provides the basic implementation of the Map interface of Java.
- It stores the data in (Key, Value) pairs.
- To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing.
- Hashing is a technique of converting a large String to small String that represents the same String so that the indexing and search operations are faster.
- HashSet also uses HashMap internally.

```
public static void main(String args[])
{
    // Creating HashMap and
    // adding elements
    HashMap<Integer, String> hm
        = new HashMap<Integer, String>();

    hm.put(1, "Geeks");
    hm.put(2, "For");
    hm.put(3, "Geeks");

    // Finding the value for a key
    System.out.println("Value for 1 is " + hm.get(1));

    // Traversing through the HashMap
    for (Map.Entry<Integer, String> e : hm.entrySet())
        System.out.println(e.getKey() + " " + e.getValue());
}
```

Output:

```
Value for 1 is Geeks
1 Geeks
2 For
3 Geeks
```

Internally

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Live Demo

```
import java.util.*;
public class AlgorithmsDemo {

   public static void main(String args[]) {

      // Create and initialize linked list
      LinkedList ll = new LinkedList();
      ll.add(new Integer(-8));
      ll.add(new Integer(20));
      ll.add(new Integer(-20));
      ll.add(new Integer(8));

      // Create a reverse order comparator
      Comparator r = Collections.reverseOrder();

      // Sort list by using the comparator
      Collections.sort(ll, r);

      // Get iterator
      Iterator li = ll.iterator();
      System.out.print("List sorted in reverse: ");

      while(li.hasNext()) {
         System.out.print(li.next() + " ");
      }
      System.out.println();
      Collections.shuffle(ll);

      // display randomized list
      li = ll.iterator();
      System.out.print("List shuffled: ");

      while(li.hasNext()) {
         System.out.print(li.next() + " ");
      }

      System.out.println();
      System.out.println("Minimum: " + Collections.min(ll));
      System.out.println("Maximum: " + Collections.max(ll));
   }
}
```


Q
&
A