SmartCliff
CAREER MOBILITY SOLUTIONS

SDE Readiness Training

Empowering Tomorrow's Innovators

Module I

Java Software Development: Effective Problem Solving

SmartCliff
Career Mobility Solutions

Generics in Java

Learning Level: Basic

DATE: 03.07.2025

Generics - Introduction

- Java generics are mainly used to impose **type safety in programs**.
- Type safety is when the **compiler validates** the **datatype of constants, variables, and methods whether it is rightly assigned or not.**
- Java Generics programming is introduced in **J2SE 5** to deal with **type-safe objects.**
- It makes the **code stable** by detecting the **bugs at compile time**.
- Before generics, we can store **any type of objects** in the **collection**, i.e., **non-generic**. Now generics force the java programmer to store a **specific type of objects**.

Generics - Introduction

- When using Java, we often write **classes and algorithms** that work around **certain data types**. Let's look at the following class as an example:

```
public class StringBox {
    public String myString;
}
```

- In the example above, we have a **StringBox** class which represents a **real-world box of words.**
- This class's methods **perform all of their computations** with regards to the **String myString field**.

What if we wanted a box of ints? We could create a new class:

```
public class IntegerBox {
    public int myInt;
}
```

Generics - Introduction

- The example above **meets our requirements**, but as the **program grows** and we need more types of **boxes, it will become unmanageable**. We can **solve this problem** by using **generics.**
- **Generics**, like the **name implies**, allow us to **create generic classes** and **methods** by specifying a type parameter. We can make **StringBox and IntegerBox into a generic Box class** like so:

```
public class Box<T> {
    private T data;
}
```

Generics - Introduction

- In the example above, we created a **generic Box** class with a **type parameter T**.
- All class methods perform their **computation around the T-type parameter.**
- We can now specify that we want a **String, Integer, or any other type of Box by specifying a type argument.**

Generics - Introduction

```
public class Box <T> {
    private T data;
    public Box(T data) {
        this.data = data;}
     public T getData() {
        return this.data; }
}
```

- **In the example above, notice that:**
  - The **type parameter** must be specified within the **diamond operator** (<>) **after the class name**.
  - The **type parameter**, T, is similar to a **method parameter but instead receives** a **class** or **interface** type as an **argument as opposed to a reference or primitive type.**

Generics - Introduction

- The **constructor** accepts a **T-type parameter** to **initialize data**.
- The **getter method** returns the **type parameter T when returning data**.

```
Box<String> myStringBox = new Box<>("Apple")
```

- In the example above, the **object myStringBox** is created like a **non-generic object**, but differs in:
  - Needing the **diamond operator** with the **class** or **interface type argument, <String>** in this example, after the class name.
  - Needing the **empty diamond operator before calling the constructor** new Box<>("Apple").

Generics - Introduction

- It’s **best practice** to make them **single, uppercase letters** to easily distinguish them from the **names of classes or variables.**
- By convention, type parameters are **E (Elements), K (Key), N (Number), T (Type), V (Value), and S (or U or V) for multiple type** parameters.

Generics - Introduction

```
// Storage class into a generic class so that it can store values of any type

public class Storage<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

```
// Storage an Integer

Storage<Integer> integerStorage = new Storage<>();

integerStorage.setValue(123);

Integer intValue = integerStorage.getValue();

System.out.println("Integer Value: " + intValue);

// Storage an string

Storage<String> stringStorage = new Storage<>();

stringStorage.setValue("Hello");

String strValue = stringStorage.getValue();

System.out.println("String Value: " + strValue);
```

Generics - Introduction

Generics Method

- Like the generic class, we can create a **generic method** that can **accept any type of arguments**.
- Here, the **scope of arguments** is **limited** to the method where it is declared.
- It **allows static as well as non-static methods.**

```
public class TestGenerics3{
   public static < E > void printArray(E[] elements){
    for ( E element : elements){
            System.out.println(element );
        }
        System.out.println();
    }
```

```
public static void main( String args[] ) {
        Integer[] intArray = { 10, 20, 30, 40, 50 };
        Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I
','N','T' };
        System.out.println( "Printing Integer Array" );
        printArray( intArray  );
        System.out.println( "Printing Character Array" );
        printArray( charArray );
    }
}
```

SmartCliff
Career Mobility Solutions

Collections in Java

Learning Level: Basic

DATE:

Collections - Introduction

- Collection is an object or a container that stores a group of other objects
- Single unit that contains and manipulates a group of objects
- Used to standardize the way in which objects are handled in the class

Real-life use cases

- Linked list - browsing history, trains coaches who are connected to each other, etc.
- Stacks - stack of plates or trays in which the topmost one gets picked first.
- Queue - same as the real-life queues, the one who enters the queue first, leaves it first too.

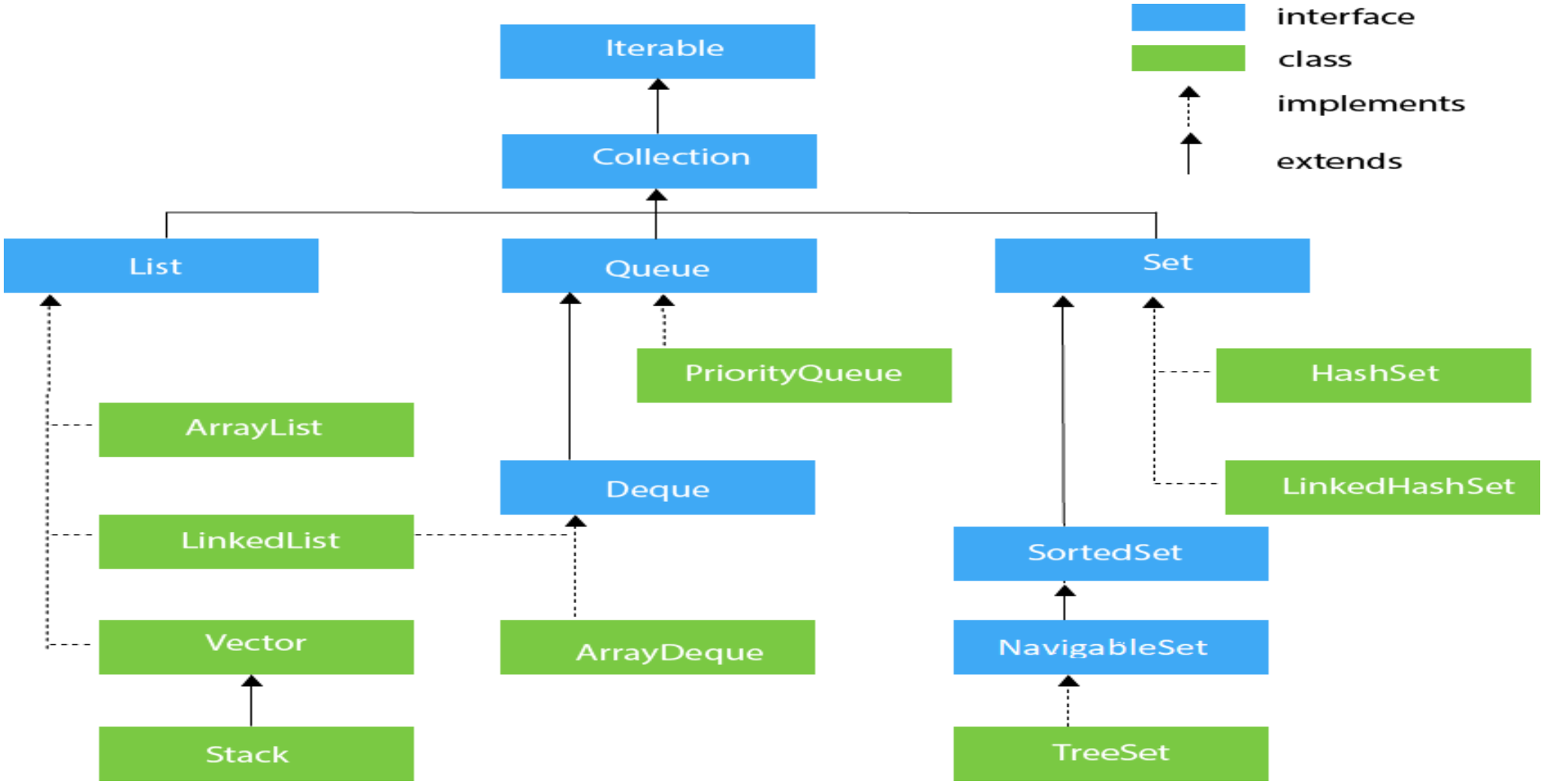
Collections - Introduction

- The Java Collections Framework - **unified architecture for representing and manipulating collection**
- **Standardizes the way in which groups of objects are handled** by your programs
- Hierarchy of interfaces and classes that provides **easy management** of a group of objects.
- Java provided ad hoc classes such as **Dictionary, Vector, Stack** and **Properties** to store and manipulate groups of objects

Collections - Introduction

- The Collections Framework was designed to meet several goals.
- First, the framework had to be **high-performance**. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are **highly efficient**.
- Second, the framework had to allow different types of collections to work in a similar manner and with a **high degree of interoperability**.
- Third, extending and/or adapting a collection had to be **easy**.
- Fourth, Facilitates code reusability

Hierarchy of Collection Framework



The Iterable Interface

- Root interface for the entire collection framework
- Used to iterate over the elements of the collection
- Iterator <E> iterator()

  Returns an iterator of type E for the collection

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.

- It provides basic operations like adding, removing, clearing the elements in a collection, checking whether the collection is empty, etc.
- **Collection** extends the **Iterable** interface.

Collections Interface

- 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.

Collections Interfaces

- To provide the greatest flexibility in their use, the collection interfaces allow some methods to be **optional**.
- The optional methods enable you to modify the contents of a collection.
- Collections that support these methods are called **modifiable**.
- Collections that do not allow their contents to be changed are called **unmodifiable**.
- If an attempt is made to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown.

The Collection Interface

- **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 Interface

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

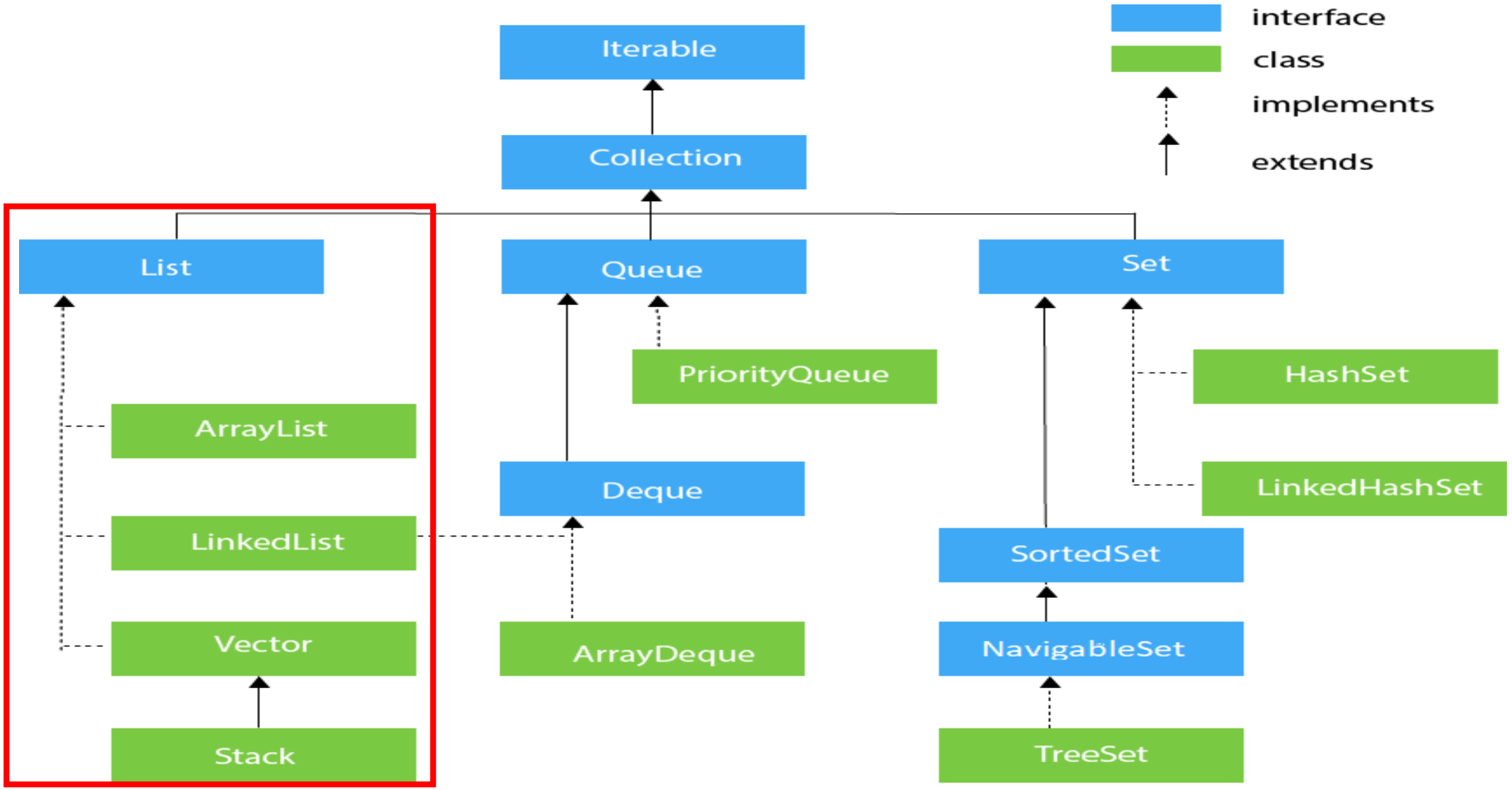
- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be **inserted or accessed** by their position in the list, using a **zero-based index**.
- List is used to store **ordered collection** of data and it may contain **duplicates 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

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 (increase or decrease in size)
- 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.
- ArrayList cannot be used for primitive data types like int, char, etc. , we need to **use a wrapper class**.

The Array List Class– Example #1

Write a Java program to demonstrate the basic usage of the ArrayList class from java.util package. The program should perform the following tasks:

- **Create an ArrayList:** Initialize an empty ArrayList of type String.
- **Display Initial Size:** Print the initial size of the ArrayList before adding any elements.
- **Add Elements to the ArrayList:**

  Add several String elements ("C", "A", "E", "B", "D", "F") to the ArrayList sequentially.

  Insert an additional element ("G") at a specific index (index 1).
- **Display Updated Size:** Print the size of the ArrayList after inserting all the elements.
- **Print Contents of the ArrayList:** Display the current elements of the ArrayList using System.out.println().
- **Remove Elements:**

  Remove a specific element "F" by value from the ArrayList.

  Remove an element at a specific index (2) from the ArrayList.
- **Display Updated Contents:**

  After the removal operations, display the updated elements of the ArrayList to reflect the changes.

The Array List Class– Example #1

```
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());
```

The Array List 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 Array List Class– Example #2

Write a Java program to perform the following tasks:

- **Create an ArrayList:** Instantiate an ArrayList of type Integer.
- **Display Initial Size:** Print the initial size of the ArrayList using the size() method before adding any elements.
- **Add Elements:** Insert integer elements (1, 2, 3, 4) into the ArrayList.
- **Display Updated Size:** Print the size of the ArrayList after adding elements.
- **Display Contents:** Print the elements of the ArrayList to show its current state.
- **Convert to Array:** Convert the ArrayList to a regular array using the toArray() method.
- **Calculate the Sum:** Iterate through the array and compute the sum of all its elements.
- **Display the Sum:** Print the calculated sum of the elements in the array.

The Array List Class– Example #2

```
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 Array List 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 Array List Class– Example #3

Write a Java program to demonstrate the following operations on an ArrayList:

- **Adding Elements:** Add multiple fruit names ("MANGO", "banana", "apple", "orange") to an ArrayList of type String.
- **Duplicate Replacement:**

  Re-add "banana" and "apple" to the list and print the new size of the list.

  Identify duplicate elements in the ArrayList and replace them with "1".
- **Removing Specific Elements:**

  Iterate through the ArrayList to remove elements marked as "1".

  Ensure that duplicates are handled properly.
- **Display List:** Display the list of unique fruits after removing duplicates.
- **Change Case: Change the case of all elements in the ArrayList:**

  Convert uppercase strings to lowercase.

  Convert lowercase strings to uppercase.
- **Output Final List:**

  Display the updated ArrayList after replacing duplicates.

The Array List Class– Example #3

```
import java.util.*;
public class ArrayListDemo {
public static void main(String args[]) {
                    ArrayList<String> fruits = new ArrayList<>();
                    fruits.add(“MANGO");fruits.add("banana"); fruits.add("apple"); fruits.add("orange");
                    fruits.add("banana");fruits.add("apple");
                    System.out.println("Fruits List Size:" + fruits.size());
                    System.out.println("List of Fruits:“ + fruits);
                    for(int i = 0 ; i < fruits.size() ; i++){
                              for(int j = i+1 ; j < fruits.size() ; j++){
                                        if(fruits.get(i).equals(fruits.get(j)))
                                                  fruits.set(j,"1");
                              }                    }
                   System.out.println("List of Fruits after Repalcement \n“ + fruits);
```

The Array List Class– Example #3

```
        int size = fruits.size(),i;
        for( i = 0 ; i < size ; i++){
            if(fruits.get(i).equals("1")){
                fruits.remove(i);
            }
            size = fruits.size();
        }
        i = i-1;
        if(fruits.get(i).equals("1"))
            fruits.remove(i);
        System.out.println("List of Unique Fruits\n“ +fruits);
        fruits.replaceAll(fruit ->fruit.equals(fruit.toUpperCase()) ? fruit.toLowerCase(): fruit.toUpperCase());
        System.out.println("List of Fruits - Changed case\n“ + fruits);
}
```

Output:

Fruits List Size:6

List of Fruits

[MANGO , banana , apple , orange , banana , apple]

List of Fruits after Repalcement

[MANGO , banana , apple , orange , 1 , 1]

List of Unique Fruits

[MANGO , banana , apple , orange]

List of Fruits - Changed case

[mango , BANANA , APPLE , ORANGE]

The Array List Class– Example #4

Create a class called **Student** with fields for storing **a student's name, roll number, and course name**.Implement the following functionalities using an **ArrayList**

- **Add** a new **student** to the system.
- **Remove** a **student** from the system using the roll number.
- **Search** for a **student** by roll number and display their details.
- **Display** all **students** currently enrolled in the system.

The Array List Class– Example #3

```
import java.util.ArrayList;import java.util.Scanner;

class Student {

    private String name;

    private int rollNumber;

    private String courseName;

    public Student(String name, int rollNumber, String courseName) {

        this.name = name;

        this.rollNumber = rollNumber;

        this.courseName = courseName;

    }
```

The Array List Class– Example #3

```
public int getRollNumber() {
        return rollNumber;
    }
   public String toString() {
        return "Name: " + name + ", Roll Number: " + rollNumber + ", Course: " + courseName;
    }
}
```

The Array List Class– Example #3

```
public class StudentManagementSystem {
    private ArrayList<Student> students;
    public StudentManagementSystem() {
        students = new ArrayList<>();
    }
    public void addStudent(String name, int rollNumber, String course) {
        Student student = new Student(name, rollNumber, course);
        students.add(student);
        System.out.println("Student added successfully.");
    }
```

The Array List Class– Example #3

```
public void removeStudent(int rollNumber) {
        for (Student student : students) {
            if (student.getRollNumber() == rollNumber) {
                students.remove(student);
                System.out.println("Student removed successfully.");
                return;
            }
        }
        System.out.println("Student with roll number " + rollNumber + " not found.");
    }
```

The Array List Class– Example #3

```
public void searchStudent(int rollNumber) {
        for (Student student : students) {
            if (student.getRollNumber() == rollNumber) {
                System.out.println(student);
                return;
            }
        }
        System.out.println("Student with roll number " + rollNumber + " not found.");
    }
```

The Array List Class– Example #3

```
public void displayAllStudents() {
        if (students.isEmpty()) {
            System.out.println("No students enrolled.");
        } else {
            for (Student student : students) {
                System.out.println(student);
            }
        }
    }
```

The LinkedList Class

- The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces.
- It provides a linked-list data structure (elements are called as nodes)
- Elements are not stored in a contiguous memory
- LinkedList uses Doubly Linked List to store its elements while ArrayList internally uses a dynamic array to store its elements.
- LinkedList is **faster** in the manipulation of data as it is node-based
- LinkedList is non-synchronized means multiple threads at a time can access the code
- **LinkedList** is a generic class that has this declaration:

class LinkedList<E>

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

The LinkedList Class

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 Linked List Class– Example #1

```
/*This example demonstrate the LinkedList Classes*/
import java.util.*;
public class LinkedListDemo01 {
        public static void main(String args[]) {
        //creating a LinkedList
            LinkedList<String> list= new LinkedList<String>();
            //displaying the initial size
            System.out.println("Size at the beginning "+list.size());
            //add elements
            list.add("Java");    list.add("C++");    list.add("JavaScript");
            list.addFirst("C#");    list.addLast("Kotlin");    list.add(2,"Python");
```

The Linked List Class– Example #1

```
    //displaying the LinkedList
System.out.println("Original Linked List " + list);
//displaying the size
System.out.println("Size after addition "+list.size());
//remove element at index 5
list.remove(5);
list.remove("C#");
//display the new LinkedList
System.out.println("New Linked List "+ list);
//display the new size
System.out.println("Size after removal "+list.size());
}
}
```

Output:

Size at the beginning 0

Original Linked List [C#, Java, Python, C++, JavaScript, Kotlin]

Size after addition 6

New Linked List [Java, Python, C++, JavaScript]

Size after removal 4

The Linked List Class– Example #2

In a **music player application**, users often have large playlists containing a series of songs they want to listen to. The **playlist is dynamic**, meaning songs can be added or removed, and the user may want to scroll through or skip between songs. The player should allow users to:

- **Add new songs to the playlist.**
- **Play songs in a specified order.**
- **Skip to the next or previous song.**
- **Remove songs from the playlist.**
- **Shuffle the playlist.**
- **Repeat the current song or the entire playlist.**

The Linked List Class– Example #2

```
import java.util.LinkedList;
import java.util.Random;

class Song {
    String title;
    String artist;

    Song(String title, String artist) {
        this.title = title;
        this.artist = artist;
    }

    @Override
    public String toString() {
        return title + " by " + artist;
    }
}
```

The Linked List Class– Example #2

```
public class MusicPlayer {
    private LinkedList<Song> playlist;
    private int currentSongIndex;

    public MusicPlayer() {
        playlist = new LinkedList<>();
        currentSongIndex = 0;
    }

    // Add a song to the playlist
    public void addSong(String title, String artist) {
        playlist.addLast(new Song(title, artist));
    }

    // Remove a song from the playlist by title
    public void removeSong(String title) {
        playlist.removeIf(song -> song.title.equals(title));
    }
```

The Linked List Class– Example #2

```
// Play the next song
    public void playNext() {
        if (currentSongIndex < playlist.size() - 1) {
            currentSongIndex++;
            System.out.println("Now playing…..\n " + playlist.get(currentSongIndex));
        } else {   System.out.println("End of playlist.");    }
    }
// Play the previous song
    public void playPrevious() {
        if (currentSongIndex > 0) {
            currentSongIndex--;
            System.out.println("Now playing….\n " + playlist.get(currentSongIndex));
        } else {   System.out.println("Already at the beginning of the playlist."); }
    }
```

The Linked List Class– Example #2

```
// Skip a specified number of songs forward
    public void skipSongs(int num) {
        if (currentSongIndex + num < playlist.size()) {
            currentSongIndex += num;
            System.out.println("Now playing…\n " + playlist.get(currentSongIndex));
        } else {        System.out.println("End of playlist.");      }
    }
     // Shuffle the playlist
    public void shuffle() {
        Random rand = new Random();
        for (int i = playlist.size() - 1; i > 0; i--) {
            int j = rand.nextInt(i + 1);
            Song temp = playlist.get(i); playlist.set(i, playlist.get(j));  playlist.set(j, temp);
        }   System.out.println("Playlist shuffled.");   }
```

The Linked List Class– Example #2

```
// Repeat the current song
  public void repeatCurrentSong() {
      System.out.println("Repeating…\n" + playlist.get(currentSongIndex));
  }
   // Display the current playlist
  public void displayPlaylist() {
      System.out.println("Current Playlist:");
      for (Song song : playlist) {
          System.out.println(song);
      }
  }
   // Play the current song
  public void playCurrentSong() {
      System.out.println("Now playing…\n " + playlist.get(currentSongIndex));    }
```

The Linked List Class– Example #2

```
public static void main(String[] args) {
        MusicPlayer player = new MusicPlayer();
         // Adding songs to the playlist
        player.addSong("Shape of You", "Ed Sheeran");
        player.addSong("Blinding Lights", "The Weeknd");
        player.addSong("Levitating", "Dua Lipa");
        player.addSong("Stay", "The Kid LAROI, Justin Bieber");
         // Displaying the playlist
        player.displayPlaylist();
         // Playing songs
        player.playCurrentSong();
        player.playNext();
        player.playNext();
        player.playPrevious();
```

```
Current Playlist:
Shape of You by Ed Sheeran
Blinding Lights by The Weeknd
Levitating by Dua Lipa
Stay by The Kid LAROI, Justin Bieber
Now playing: Shape of You by Ed Sheeran
Now playing: Blinding Lights by The Weeknd
Now playing: Levitating by Dua Lipa
Now playing: Blinding Lights by The Weeknd
Skipping 2 songs...
Now playing: Stay by The Kid LAROI, Justin Bieber
```

The Linked List Class– Example #2

```
        // Skipping 2 songs
        player.skipSongs(2);
        // Shuffling the playlist
        player.shuffle();
        // Repeat the current song
        player.repeatCurrentSong();
        // Remove a song
        player.removeSong("Blinding Lights");
        player.displayPlaylist();
    }
}
```

Playlist shuffled.

Repeating: Stay by The Kid LAROI, Justin Bieber

Current Playlist:

Shape of You by Ed Sheeran

Levitating by Dua Lipa

Stay by The Kid LAROI, Justin Bieber

The Vector Class

- Vector uses a **dynamic array** to store the data elements and it is similar to **ArrayList**.
- It is **synchronized** and contains many methods that are not the part of Collection framework.

Methods to create the constructor of Vectors

- **Vector<Data-type> v = new Vector<Data-Type>()** - default size of Vectors is 10
- **Vector<Data-type> v = new Vector<Data-Type>(int size)** - specifying the desired size
- **Vector<Data-type> v = new Vector<Data-Type>(int size,int incr)** - initial capacity is declared by size, increment specifies number of elements to allocate each time that vector gets resized upward
- **Vector<Data-type> v = new Vector<Data-Type>(Collection C)** - Creating a Vector from Collection

The Vector Class– Example #1

```
/*This example demonstrate the Vector Class*/
import java.util.*;
public class VectorDemo {
        public static void main(String args[]){
                Vector<String> v=new Vector<String>();
                System.out.println("Size of the vector is "+v.size());
                v.add("A");  v.add("B");  v.add("C");  v.add("D");
                System.out.println("Elements in the vector "+v);
                System.out.println("Size of the vector is "+v.size());
                v.remove("A");
                System.out.println("Elements in the vector after remove "+v);
                System.out.println("Size of the vector after the removal is "+v.size());
        }
}
```

Output:

Size of the vector is 0

Elements in the vector [A, B, C, D]

Size of the vector is 4

Elements in the vector after remove [B, C, D]

Size of the vector after the removal is 3

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.
- Stack is thread-safe

The Stack Class– Example #1

```
/*This example demonstrate the reverse a string using loop */
public class StringReverser {
    public static void main(String[] args) {
        String input = "hello"; // Example input
        String reversed = reverseString(input);
        System.out.println("Original String: " + input);
        System.out.println("Reversed String: " + reversed);
    }
    public static String reverseString(String str) {
        String reversed = ""; // Initialize an empty string to hold the reversed string
```

The Stack Class– Example #1

```
// Iterate from the end of the string to the beginning
        for (int i = str.length() - 1; i >= 0; i--) {
            reversed += str.charAt(i); // Append each character to the reversed string
        }
        return reversed; // Return the reversed string
    }
}
```

The Stack Class– Example #2

```
/*This example demonstrate the reverse a string using Stack Class*/
import java.util.Stack;
import java.util.Scanner;
public class StackDemo {
        public static void main(String args[]){
                System.out.print("Input String:");
                String str = new Scanner(System.in).next();
                Stack<Character> stack = new Stack<>();
                for (char ch : str.toCharArray()) {
                        stack.push(ch);
                }
```

The Stack Class– Example #2

```
StringBuilder reversed = new StringBuilder();
while (!stack.isEmpty()) {
        reversed.append(stack.pop());
}
System.out.println("Reversed: " + reversed.toString());
}
}
```

Output:

Input String : reverse

Reversed : esrever

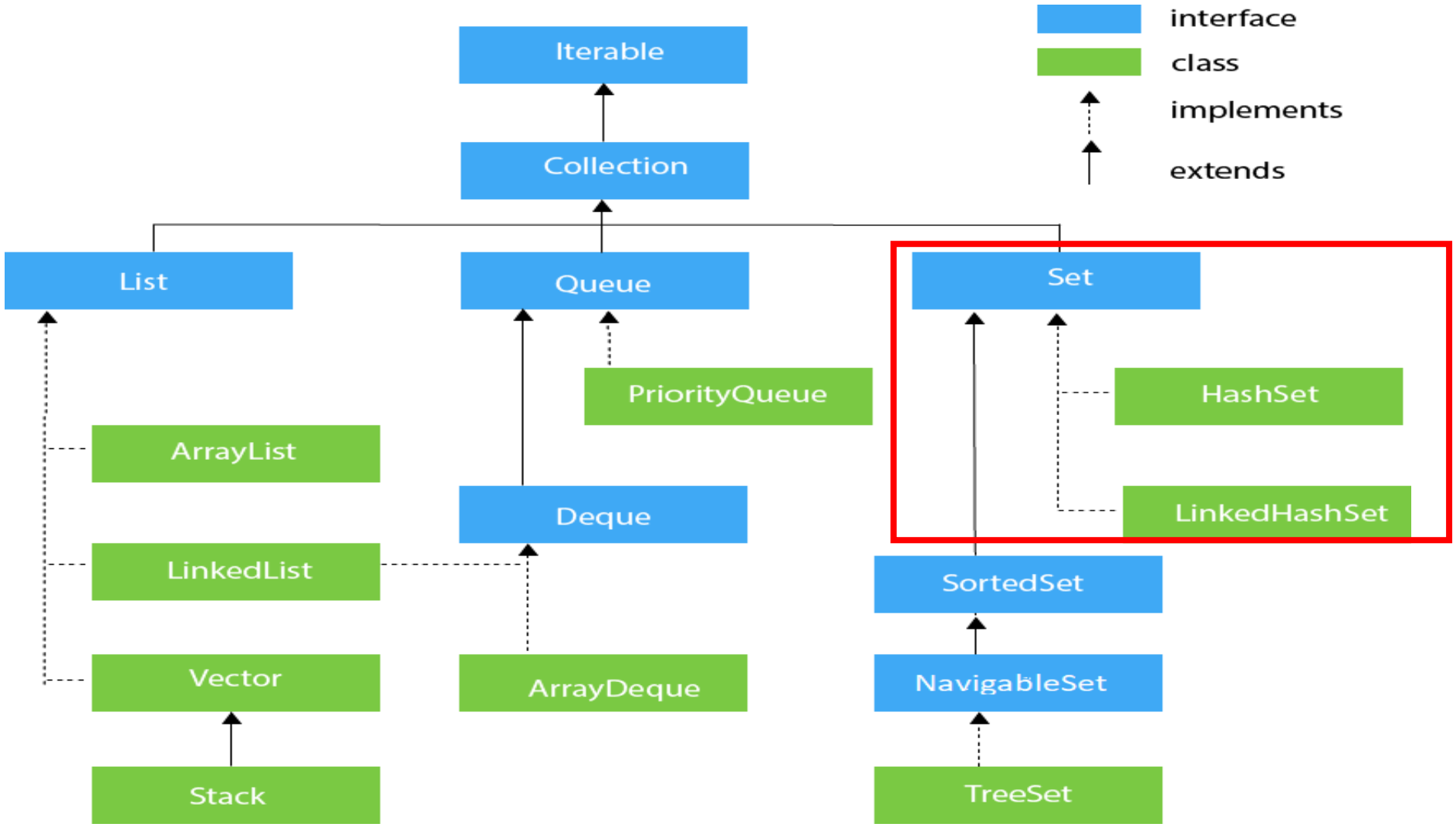
The Set Interface

- The **Set** interface defines a set (unordered collection)
- 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.

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 which uses a mechanism called **Hashing**
- **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**.
- The hash code is then used as an index, at which the data associated with the key is stored

The HashSet Class

- When we insert elements into the HashSet, it is **not guaranteed that it gets stored in the same order** and we can store Null values in this
- HashSet is **non-synchronized** means multiple threads at a time can access the code
- **HashSet** does not define any additional methods beyond those provided by its superclasses and interfaces.

Methods to create the constructors of HashSet

- **Creating an empty HashSet - default initial capacity is 16.**

  HashSet<Data-type> hs = new HashSet<Data-type>()

The HashSet Class

- **Creating a HashSet with a specified size**

  HashSet<Data-type> hs = new HashSet<Data-type>(int size)

- **Creating a HashSet with a specified size and fill ratio**

  HashSet<Data-type> hs = new HashSet<Data-type>(int size,float fillRatio)

- **Creating a HashSet from Collection**

  HashSet<Data-type> hs = new HashSet<Data-type>(Collection C)

The HashSet Class– Example #1

Write a program to identify **duplicate elements** in an **array of strings** using a **HashSet** in Java. The program should determine which strings in the array appear more than once and display them as a **list of duplicates.**

The HashSet Class– Example #1

```
import java.util.*;
public class HashSetDemo {
        public static void main(String args[]){
                String[] arr = {"Alpha", "Beta", “Alpha", "Epsilon", "Epsilon", "Omega"};
                HashSet<String> set = new HashSet<>();
                List<String> duplicates = new ArrayList<>();
                for (String num : arr) {
                        if (set.contains(num)) {
                                duplicates.add(num);
                        }else{     set.add(num);          }
                }
                System.out.println("Duplicates: " + duplicates);
        }
}
```

Output:

Duplicates: [Alpha, Epsilon]

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.**
- LinkedHashSet is **non-synchronized** means multiple threads at a time can access the code

The HashSet Class– Example #1

Write a program to **remove duplicate elements** from a list of strings while **preserving the original order** of their first appearance. The program should utilize a **LinkedHashSet to achieve** this functionality

The LinkedHashSet Class– Example #1

```
import java.util.*;
public class LinkedHashSetDemo01 {
        public static void main(String args[]){
                List<String> items = new ArrayList<>();
                items.add("apple");
                items.add("banana");
                items.add("apple");
                items.add("orange");
                items.add("banana");
                Set<String> fruits = new LinkedHashSet<>(items);
                System.out.println(“ Remove Duplicates -While Preserving Insertion Order:”+ new ArrayList<>(fruits));
        }
}
```

Output:

Remove Duplicates - Preserving Insertion Order

[apple, banana, orange]

The Sorted Set 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.

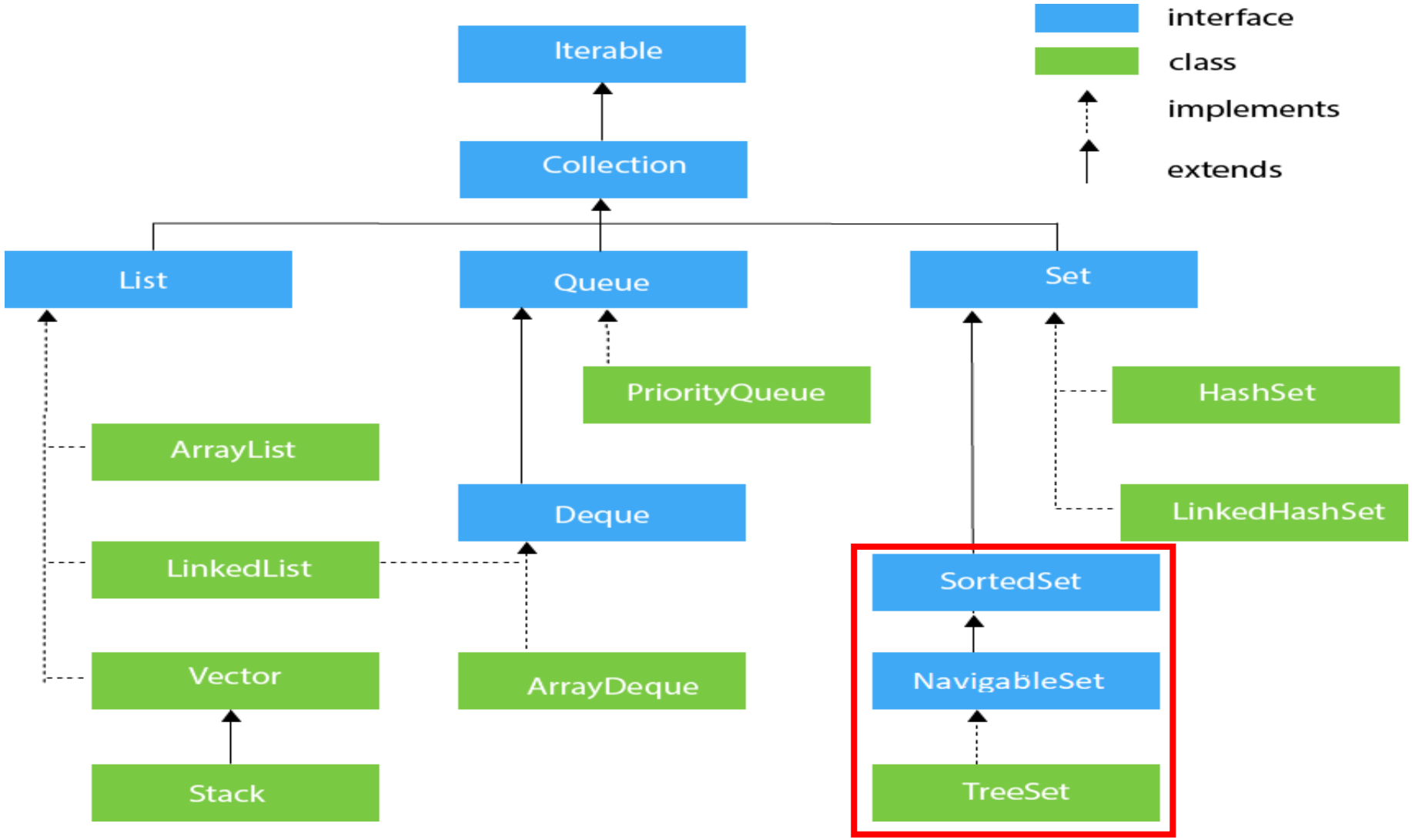
- The **TreeSet class implements** this interface
- In addition to those methods provided by **Set**, the **SortedSet** interface declares the methods summarized

The Sorted Set Interface

- In addition to the methods defined by **Collection**, **Sorted Set** 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 Sorted Set Interface



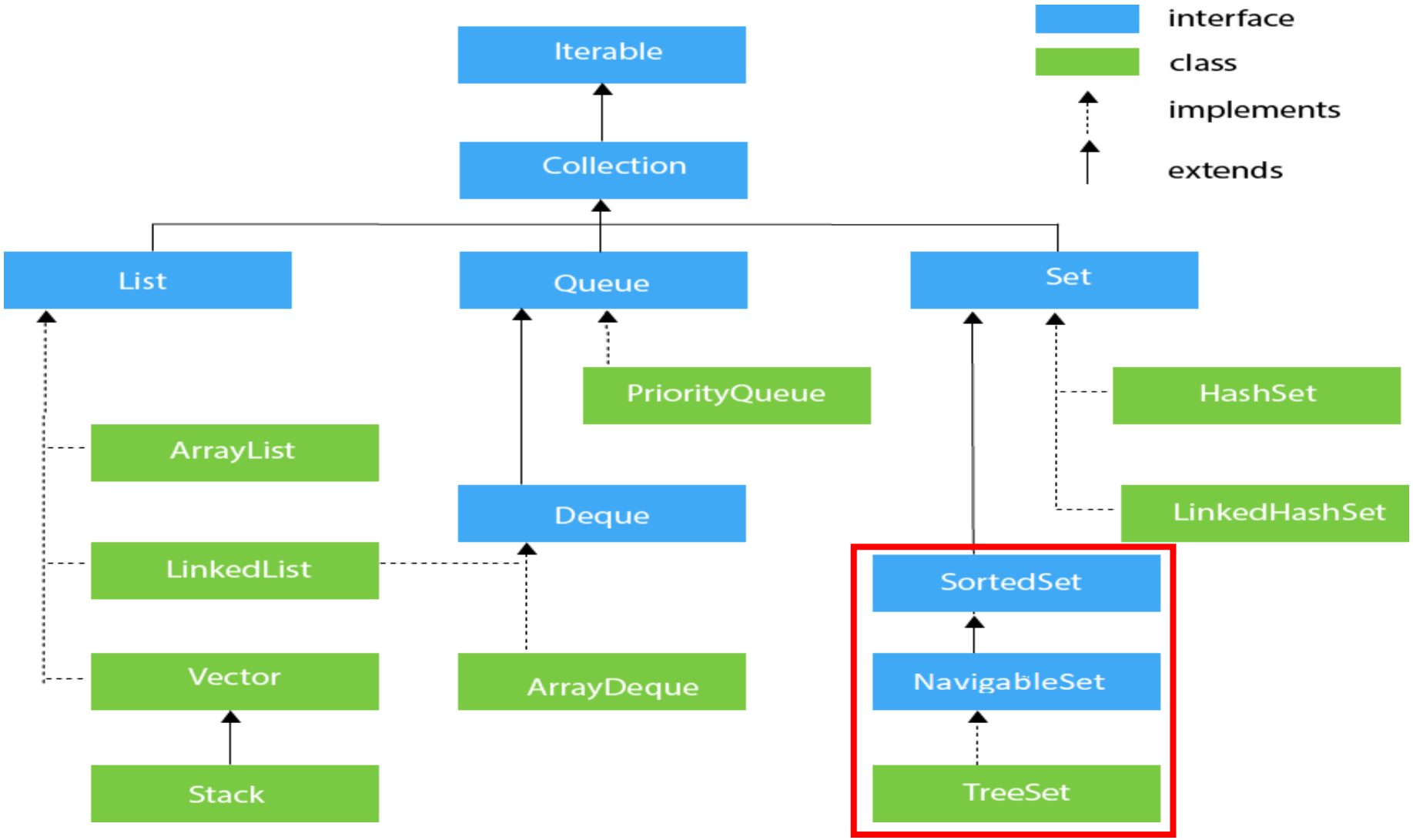
The NavigableSet Interface

- 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.

The NavigableSet Interface



The Navigable Set 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 Navigable Set 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.

- TreeSet is **non-synchronized** - Multiple operations on TreeSet can be performed at a time.

The TreeSet Class– Example #1

Write a program to demonstrate the use of a **TreeSet for extracting a subset of elements** within a **specified range**. The program should store a **collection of integers in a TreeSet**, which maintains elements **in sorted order**, and retrieve elements within a defined range **(inclusive of boundaries).**

The TreeSet Class– Example #1

```
import java.util.TreeSet;
import java.util.List;
public class Main {
        public static void main(String[] args) {
                TreeSet<Integer> set = new TreeSet<>(List.of(2, 5, 8, 12, 15, 20));
                 System.out.println("Elements in range: " + set);
                 int low = 5,high = 15;
                TreeSet<Integer> subSet = new TreeSet<>(set.subSet(low,true,high,true));
                System.out.println("Elements in range: " + subSet);
        }
}
```

Output:

Elements in range: [2, 5, 8, 12, 15, 20]

Elements in range: [5, 8, 12, 15]

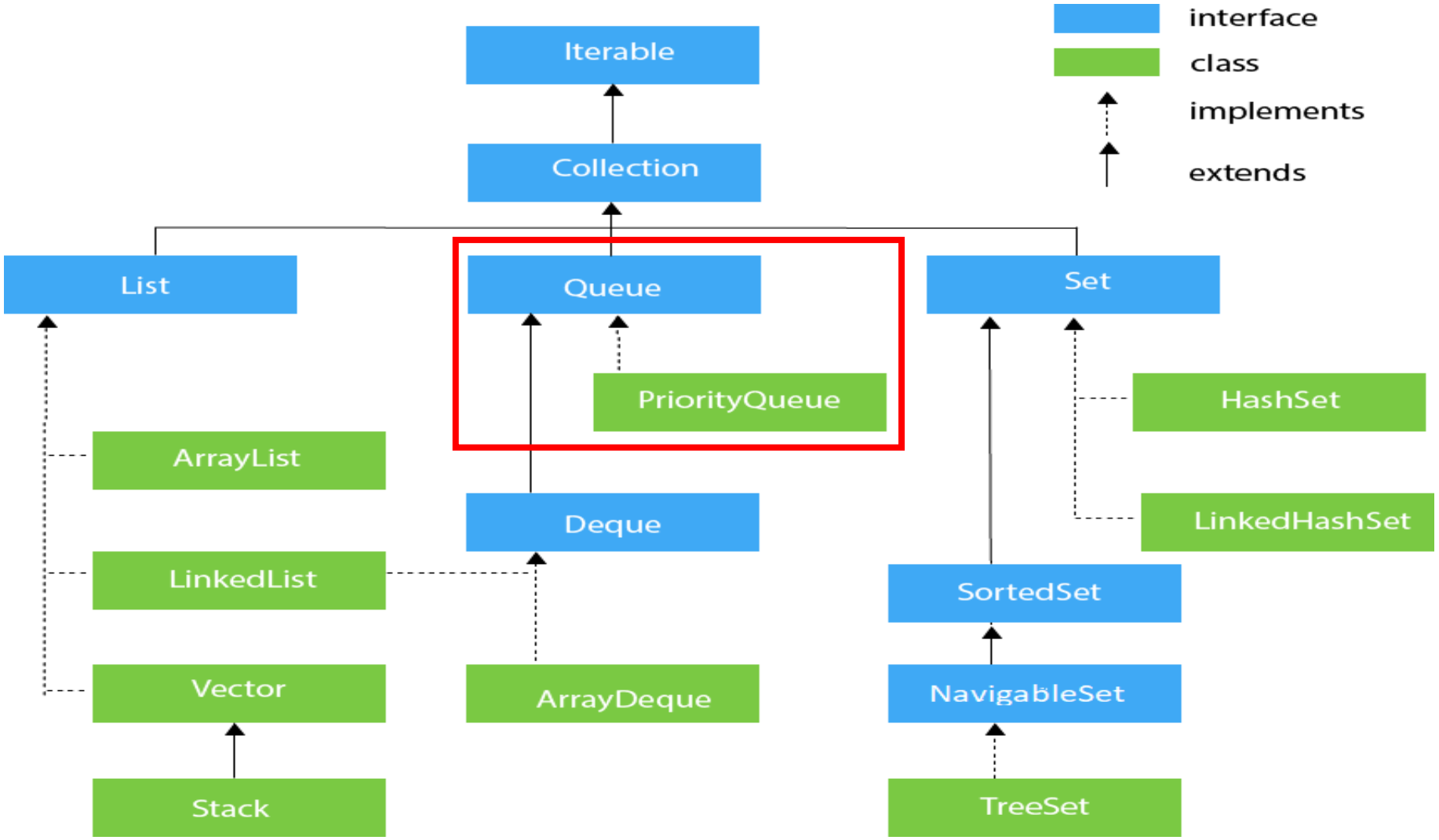
The Queue Interface

- 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.

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 Priority Queue 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.
- It does not allow null values to be stored inside it

The Priority Queue Class– Example #1

Write a program to find the **k-th largest element** in an **unsorted array of integers using a PriorityQueue** in Java. The program should leverage the **min-heap property of the PriorityQueue** to efficiently identify the desired element without **fully sorting the array.**

```
import java.util.PriorityQueue;
class Main {
        public static void main(String[] args) {
                int[] nums = {3, 2, 1, 5, 6, 4};
                int k = 2;
                PriorityQueue<Integer> minHeap = new PriorityQueue<>(k);
                for (int num : nums) {
                        minHeap.add(num); // Add the current number to the heap
                }
```

The Priority Queue Class– Example #1

```
            // If the heap size exceeds k, remove the smallest element
             if (minHeap.size() > k) {
                  minHeap.poll();
              }
           }
             System.out.println("The " + k + "th largest element is: " + minHeap.peek());
        }
}
```

Output:

The 2th largest element is: 5

The Priority Queue Class– Example #2

Write a program to **manage a set of tasks** with varying priorities using a **PriorityQueue** in Java. The program should add tasks to the queue, **sort them by priority (highest priority first),** and execute them in the **correct order.**

```
import java.util.*;
class Task implements Comparable<Task> {
    private String name;
    private int priority;
    public Task(String name, int priority) {
        this.name = name;  this.priority = priority;
    }
    public String getName() {        return name;    }
     public int getPriority() {        return priority;    }
```

The Priority Queue Class– Example #2

```
@Override
    public int compareTo(Task other)
        // Higher priority tasks should come first
        return Integer.compare(other.priority, this.priority);
    }}
public class Main {
    public static void main(String[] args) {
        // Instantiate a PriorityQueue of Task objects
        PriorityQueue<Task> priorityQueue = new PriorityQueue<>();
        // Add elements with specified priority
        priorityQueue.add(new Task("Task 1", 3));
        priorityQueue.add(new Task("Task 2", 1));
        priorityQueue.add(new Task("Task 3", 2));
```

The Priority Queue Class– Example #2

```
        // Poll elements from the priority queue
        while (!priorityQueue.isEmpty()) {
            Task task = priorityQueue.poll();
            System.out.println("Executing: " + task.getName() + " (Priority: " + task.getPriority() + ")");
        }
    }
}
```

Output:

Executing: Task 1 (Priority: 3)

Executing: Task 3 (Priority: 2)

Executing: Task 2 (Priority: 1)

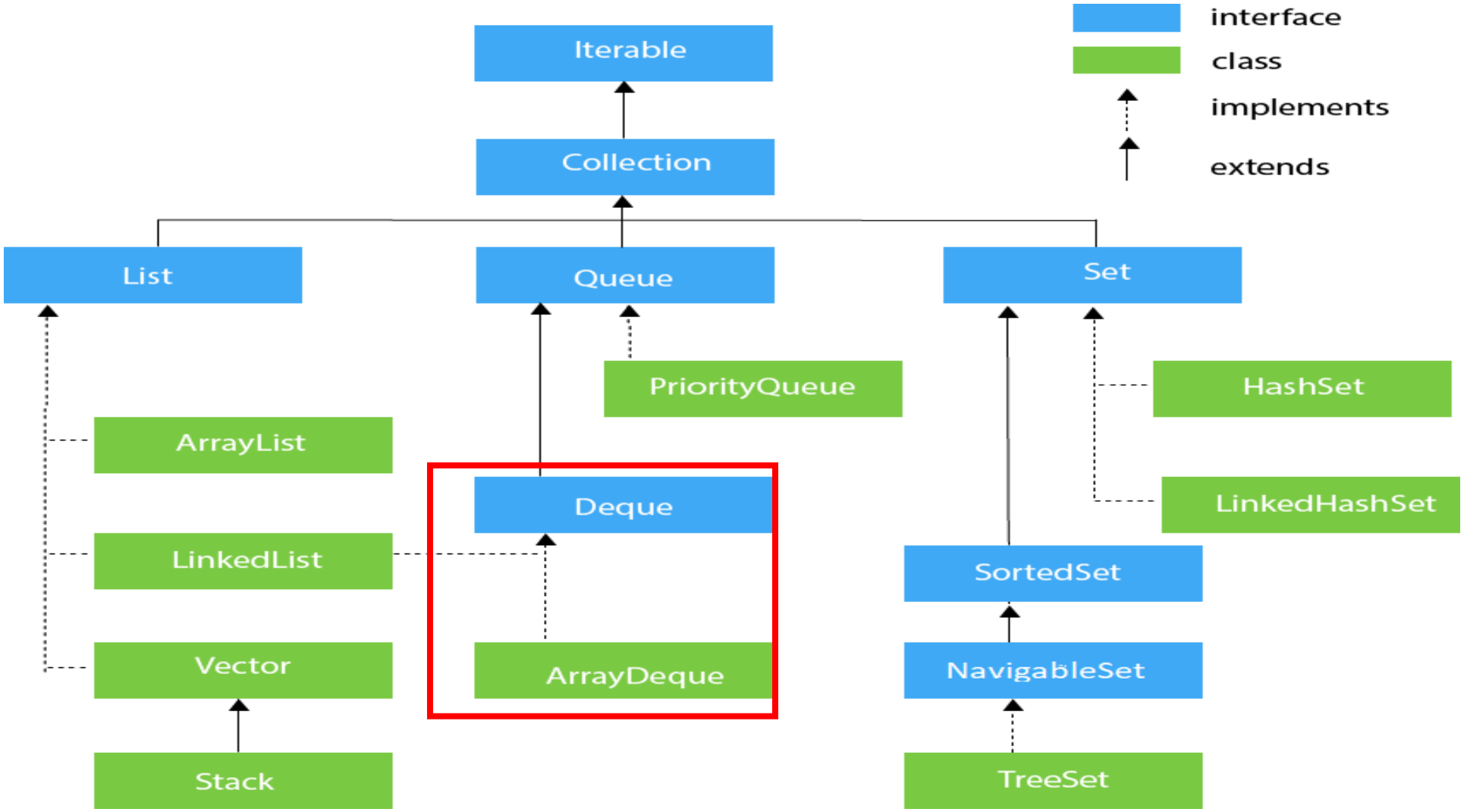
The DeQue Interface

- The **Deque** interface extends **Queue** and declares the behavior of a **double-ended 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.

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 Array DeQue 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 Array Deque Class– Example #1

```
/*Write a program to check whether a given string is a palindrome using a double-ended queue (Deque).*/
import java.util.*;
class Main {
        public static void main(String[] args) {
                System.out.println("Input String");
                String s = new Scanner(System.in).next();
                boolean flag = true;
                if (s == null || s.length() <= 1) {
                        System.out.println("Invalid");
                }
                Deque<Character> deque = new ArrayDeque<>();
                for (char c : s.toCharArray()) {
                        deque.add(c);
                }
```

The Array Deque Class– Example #1

```
        while (deque.size() > 1)
        {
                char front = deque.pollFirst(); // remove from front
                char back = deque.pollLast();   // remove from back
                if (front != back) {
                    flag = false;
                }
         }
        System.out.println("Is "+ s + " a palindrome? " + flag);
    }
}
```

Output:

Input String

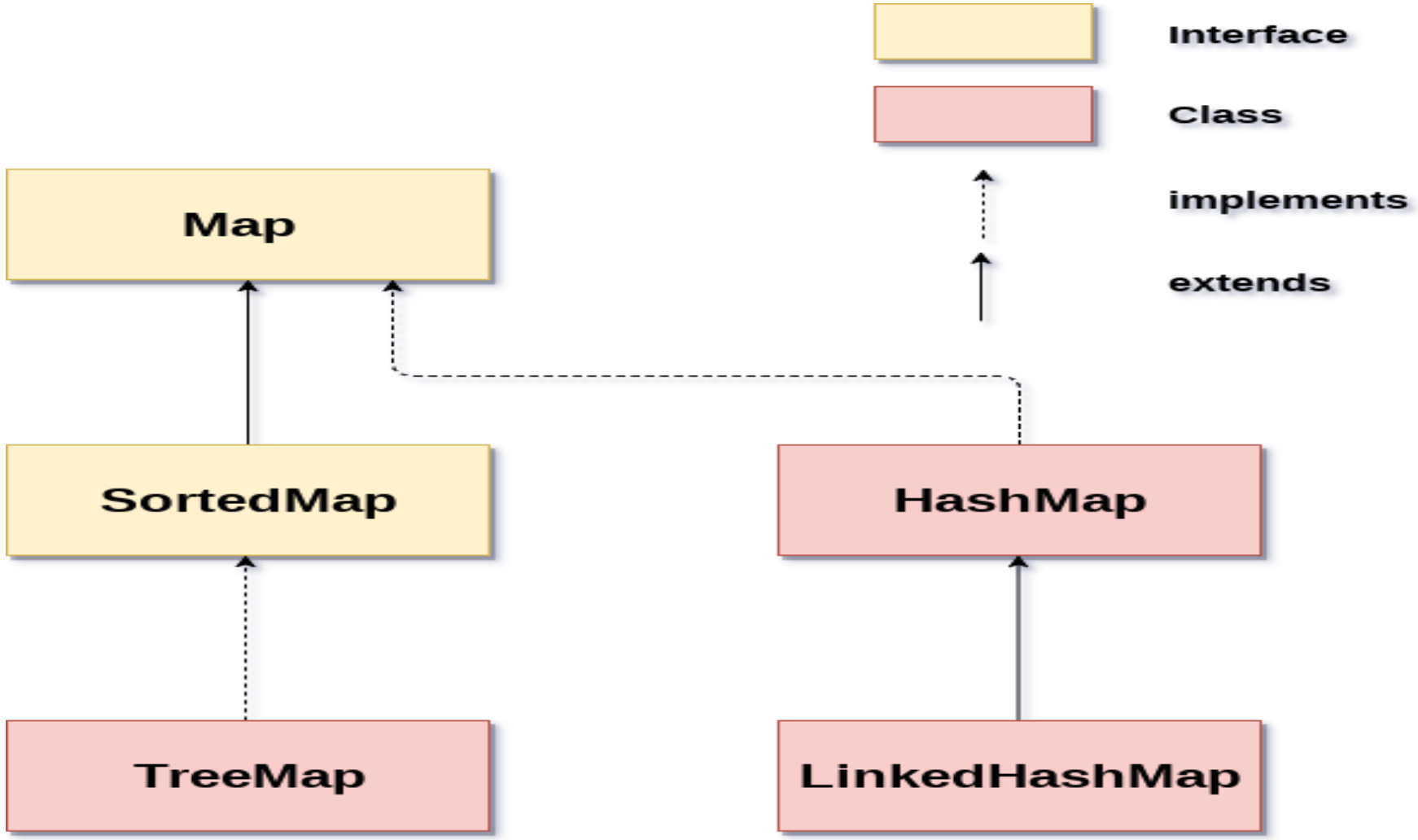
racecar

Is racecar a palindrome? true

The Map Interface

- The **maps interface** in java is a **collection** that **links a key with value pairs**.
- Each entry in the **map store the data** in a **key** and its **corresponding value**.
- **Map interface** contains only **unique keys** and **does not allow any duplicate keys**.
- The **map interface** in Java is a part of the **java.util.map interface**.
- A **key is an object** that you use to **access the value** later, it is associated with a **single value**.
- A map is used when you **need to search, edit, or remove elements** based on a **key**.
- A Map **cannot be traversed**, therefore you must use the **keySet() or entrySet**() method to **convert it into a Set.**
- There are **two interfaces** for implementing Map in java: **Map and SortedMap**, and **three classes**: **HashMap, LinkedHashMap, and TreeMap** as follows:

The Map Hierarchy



Classes that implements Map

Class	Description
HashMap	Although HashMap implements Map, it doesn't maintain any kind of order.
LinkedHashMap	The implementation of Map is LinkedHashMap. It inherits the class HashMap. The insertion order is maintained.
TreeMap	Both the map and the SortedMap interfaces are implemented. In TreeMap, the order is always maintained in ascending order.
Map	Duplicate keys are not allowed in a map, although duplicate values are.
SortedMap	In SortedMap, elements can be traversed in the sorted order of their keys.

Creating Map Objects

- Since Map is an **interface**, **objects** of the **type map cannot be created**.
- To **create an object**, we always **need a class that extends this map**. Also, because Generics were added in Java 1.5, it is now **possible to restrict the types of objects** that can be **stored in the Map**.

Syntax:

```
// Defining Type-safe Map
Map hm = new HashMap();
```

Example:

```
Map<String, String> hm = new HashMap<>();

hm.put("India", "New Delhi");
hm.put("USA", "Washington");
hm.put("United Kingdom", "London");
```

Characteristics of a Map Interface

- Each **key can map to a maximum of one value**, and a **map cannot contain multiple keys**. While some implementations, like the **HashMap and LinkedHashMap**, allow **null keys** and **null values**, others, like the TreeMap, do not.
- In the **map interface** in java, the **order depends on the specific implementations**. For eg, TreeMap and LinkedHashMap have a predictable order, whereas HashMap does not.
- Java provides **two interfaces** for implementing Map. They consist of three classes: HashMap, LinkedHashMap, and TreeMap as well as Map and a SortedMap.
- Set of keys, Set of **Key-Value Mappings**, and **Collection of Values** are the three collection views a map interface provides.
- Since the **Map interface** is **not a subtype of the Collection interface**. Thus, it differs from the other collection types in terms of features and behaviors.

When to use the Map interface in Java

- When someone has to **retrieve and update elements based on keys** or **execute lookups by keys**, **the maps are used**. Additionally, For key-value association mapping like dictionaries, maps are useful. The following are a few common scenarios:

    1. A map of cities and their zip codes.

    2. A map of error codes with their descriptions.

    3. A map of school classes and the students names. Each key (class) is associated with a list of values (student).

    4. A map of managers and employees in a company.

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 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 Hash Map 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 Hash Map 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 Linked HashMap 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 Linked Hash Map Class – Example #1

```
/*Write a program to extract and print the unique characters from a given input string, preserving their
order of first appearance.*/
import java.util.*;
public class Main {
    public static void main(String[] args) {
            System.out.println("Input String:");
            String s = new Scanner(System.in).next();
            LinkedHashMap<Character, Boolean> map = new LinkedHashMap<>();
            for (char c : s.toCharArray()){
                map.putIfAbsent(c, true); // Only adds the character if it's not already present
            StringBuilder result = new StringBuilder();
            for (Character key : map.keySet())
                result.append(key);
            System.out.println("Unique characters in order for "+ s + ": " + result.toString());
    }
}
```

Output:

Input String:mississippi

Unique characters in order for mississippi: misp

The Sorted Map 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.

The Sorted Map Interface

- The methods declared by **Sorted Map** 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 Tree Map 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 Tree Map Class – Example #1

/*Write a program that accepts a single-line sentence as input, counts the occurrences of each unique word in the sentence, and outputs the words and their respective counts in lexicographical order*/

```
import java.util.TreeMap;
import java.util.Map;
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        String sentence = new Scanner(System.in).nextLine();
        Map<String, Integer> wordCountMap = new TreeMap<>();
        for (String word : sentence.split(" ")) {
             wordCountMap.put(word, wordCountMap.getOrDefault(word, 0) + 1);
        }
       for (Map.Entry<String, Integer> entry : wordCountMap.entrySet()) {
            System.out.print(entry.getKey() + ": " + entry.getValue() +"\t");
        }}}
```

Output:

p r o g r a m m i n g

a: 1 g: 2 i: 1 m: 2 n: 1 o: 1 p: 1 r: 2

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 List Iterator Interface

- The methods declared by **List Iterator** 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 – Output

```
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+
```

Spliterators

- JDK 8 added another type of iterator called a **spliterator** that is defined by the Spliterator interface.
- A spliterator **cycles through a sequence of elements** and it is similar to the iterators. However, the techniques required to use it differ.
- It offers substantially **more functionality** than does either Iterator or ListIterator.
- It provide support for parallel iteration of portions of the sequence. Thus, Spliterator supports **parallel programming**.
- It offers a **streamlined approach** that combines the hasNext and next operations into one method.

The Spliterator Interface

- Spliterator is a **generic interface** that is declared like this:

interface Spliterator<T>

- Here, **T** is the type of elements being iterated.
- The methods declared by **Spliterator** are summarized

Method	Description
int **characteristics**()	Returns a set of **characteristics** of this Spliterator and its elements.
long **estimateSize**()	Returns an estimate of the number of elements that would be encountered by a **forEachRemaining(java.util.function.Consumer<? super T>)** traversal, or returns **Long.MAX_VALUE** if infinite, unknown, or too expensive to compute.
default void **forEachRemaining** (Consumer<? super T> action)	Performs the given **action for each remaining element**, sequentially in the current thread, until all elements have been processed or the action throws an exception.

The Spliterator Interface

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

Method	Description
default Comparator<? super T> **getComparator**()	If this Spliterator's source is **SORTED** by a **Comparator**, returns that Comparator.
default long **getExactSizeIfKnown**()	Convenience method that returns **estimateSize()** if this Spliterator is **SIZED**, else -1.
default boolean **hasCharacteristics** (int characteristics)	Returns true if this Spliterator's **characteristics()** contain all of the given characteristics.
boolean **tryAdvance** (Consumer<? super T> action)	If a **remaining element exists**, **performs the given action on it**, returning true; else returns false.
Spliterator<T> **trySplit**()	If this spliterator can be partitioned, returns a Spliterator covering elements, that will, upon return from this method, not be covered by this Spliterator.

The Spliterator Class – Example #1

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

public class Main {
        public static void main(String args[]) {
                //Create an Arraylist
                ArrayList <Double> doubleValues = new ArrayList<Double>();
                doubleValues.add(1.0);
                doubleValues.add(2.0);
                doubleValues.add(3.0);
                doubleValues.add(4.0);
                doubleValues.add(5.0);
                doubleValues.add(6.0);
```

The Spliterator Class – Example #1

```
                Spliterator<Double> firstHalf = doubleValues.spliterator();
                Spliterator<Double> secondtHalf = firstHalf.trySplit();

                System.out.println("Contents of ArrayList using tryAdvance ");
                while(firstHalf.tryAdvance((n)->System.out.print(n+" ")));
                System.out.println();

                System.out.println("Contents of ArrayList using forEachRemaining ");
                secondtHalf.forEachRemaining((n)->System.out.print(n+" "));
                System.out.println();
        }
}
```

Output:

Contents of ArrayList using tryAdvance

4.0 5.0 6.0

Contents of ArrayList using forEachRemaining

1.0 2.0 3.0

Comparable

- **Java Comparable interface** is used to order the objects of the user-defined class.
- This interface is found in **java.lang** package and contains only one method named **compareTo(Object).**
- It provides a **single sorting sequence** only, i.e., we can sort the elements on the basis of single data member only. **For example**, Sort based on student regno or name or any other member.

Syntax:

```
public int compareTo(Object obj)
```

- It is used to compare the current object with the specified object. It returns :
  - Positive integer, if the **current object is greater than the specified object.**
  - Negative integer, if the **current object is less than the specified objec**t.
  - Zero, if the **current object is equal** to the specified object.

Comparable

```
/***This Example demonstrate sorting using Comparable*/
import java.io.*;
import java.util.*;
// A class 'Mobile' that implements Comparable
class Mobile implements Comparable<Mobile>{
        private String name;
        private int ram;
        private int price;
        Mobile(String name, int ram, int price){
                this.name = name;
                this.ram = ram;
                this.price = price;
        }
        String getName() {
                return name;
        }
        int getRam() {
                return ram;
        }
```

Comparable

```
        void setRam(int ram) {
                this.ram = ram;
        }
        void setName(String name) {
                this.name = name;
        }
        int getPrice() {
                return price;
        }
        void setPrice(int price) {
                 this.price = price;
        }
        //compare the current object with the specified object.
        public int compareTo(Mobile o) {
                if (this.ram > o.getRam())
                        return 1;
                else
                        return -1;
        }
}
```

Comparable

```
//Main Class
class ComparableExample {
        public static void main(String[] args) {
                List<Mobile> mobileList = new ArrayList<>();
                mobileList.add(new Mobile("RedMe", 16, 800));
                mobileList.add(new Mobile("Apple", 8, 100));
                mobileList.add(new Mobile("Samsung", 4, 600));
                Collections.sort(mobileList);
                System.out.println("Mobiles after sorting : ");
                System.out.println("Name"+"\t"+"Ram"+"\t"+"Price");
                for (Mobile mb : mobileList) {
                        System.out.println(mb.getName() + "\t" +
                        mb.getRam() + "\t" + mb.getPrice());
                }
        }
}
```

Comparator

- **Java Comparator interface** is used to **order the objects of a user-defined class.**
- This interface is found in **java.util package** and contains the following method:
  - **compare(Object obj1,Object obj2) -** It compares the first object with the second object.
- It provides **multiple sorting sequence**s, i.e., we can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Comparator

```
/***This Example demonstrate sorting using Comparable and Comparator*/
import java.io.*;
import java.util.*;
// A class 'Mobile' that implements Comparable
class Mobile implements Comparable<Mobile>{
        private String name;
        private int ram;
        private int price;
        Mobile(String name, int ram, int price){
                this.name = name;
                this.ram = ram;
                this.price = price;
        }
```

Comparator

```
        String getName() {
                return name;
        }
        int getRam() {
                return ram;
        }
        int getPrice() {
                return price;
        }
        //compare Mobiles by Ram size
        public int compareTo(Mobile o) {
                if (this.ram > o.getRam())
                        return 1;
                else
                        return -1;

}
```

Comparator

```
// Class to compare Mobiles by price
class PriceCompare implements Comparator<Mobile>{
        public int compare(Mobile m1, Mobile m2){
                if (m1.getPrice() < m2.getPrice()) return -1;
                if (m1.getPrice() > m2.getPrice()) return 1;
                else return 0;
        }
}
// Class to compare Mobiles by name
class NameCompare implements Comparator<Mobile> {
        public int compare(Mobile m1, Mobile m2) {
                return m1.getName().compareTo(m2.getName());
        }
}
```

Comparator

```
// Main class
class ComparatorExample{
        public static void main(String[] args){
                List<Mobile> mobileList = new ArrayList<>();
                mobileList.add(new Mobile("RedMe", 16, 800));
                mobileList.add(new Mobile("Apple", 8, 100));
                mobileList.add(new Mobile("Samsung", 4, 600));
                System.out.println("Sorted by Price");
                PriceCompare priceCompare = new PriceCompare();
                Collections.sort(mobileList, priceCompare);
                System.out.println("Mobiles after price sorting : ");
                System.out.println("Name"+"\t"+"Ram"+"\t"+"Price");
                for (Mobile mb : mobileList){
                        System.out.println(mb.getName() + "\t" +
                                                mb.getRam() + "\t" +
                                                mb.getPrice());
                }
```

Comparator

```
System.out.println("\nSorted by Name");
NameCompare nameCompare = new NameCompare();
Collections.sort(mobileList, nameCompare);
System.out.println("Mobiles after price sorting : ");
System.out.println("Name"+"\t"+"Ram"+"\t"+"Price");
for (Mobile mb : mobileList){
        System.out.println(mb.getName() + "\t" +
                                mb.getRam() + "\t" +
                                mb.getPrice());
}
```

Comparator

```
                    // Uses Comparable to sort by Ram
                    System.out.println("\nSorted by Ram Size");
                    Collections.sort(mobileList);
                    System.out.println("Name"+"\t"+"Ram"+"\t"+"Price");
                    for (Mobile mb : mobileList){
                            System.out.println(mb.getName() + "\t" +
                                                    mb.getRam() + "\t" +
                                                    mb.getPrice());
                    }
            }
}
```

Comparable Vs Comparator

S.No	Comparable	Comparator
1.	The comparable interface has a method **compareTo(Object a)**	The comparator interface has a method **compare(Object O1, Object O2)**
2.	Comparable interface belongs to **java.lang** package.	Comparator interface belongs to **java.util** package.
3.	**Collection.sort(List)** method can be used to sort the collection of Comparable type objects.	**Collection.sort(List, Comparator)** method can be used to sort the collection of Comparator type objects.
4.	Comparable provides **single sorting** sequence.	The comparator provides a **multiple sorting** sequence.

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

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

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

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

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

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

Success doesn't come to you, you have to go to it.

- Marva Collins


THANK YOU
SmartCliff
Career Mobility Solutions