Java Collection Framework

A **Collection** is an object that manages a group of other objects (their reference values).

Q) In Java applications, to manage a group of other objects we already have Arrays then what is the requirement to use Collection objects?

Ans: Collections address several limitations of Arrays:

- **Size:** Arrays are fixed-size; Collections are dynamically growable.
    - *Reasoning:* You can add elements beyond the initial capacity of a Collection without an `ArrayIndexOutOfBoundsException`, as the Collection automatically resizes.

    Java

    ```
    // Array Example (Fixed Size)
    // Student[] stds = new Student[3];
    // stds[0] = new Student();
    // stds[1] = new Student();
    // stds[2] = new Student();
    // stds[3] = new Student(); // --> ArrayIndexOutOfBoundsException
    ```

    Java

    ```
    // Collection Example (Dynamic Size)
    // import java.util.ArrayList;
    // ArrayList al = new ArrayList(3); // Initial capacity can be set,
    but it grows

    // al.add(new Student());
    // al.add(new Student());
    // al.add(new Student());
    // al.add(new Student()); // --> No Exception, ArrayList resizes
    ```

- **Element Types:** Arrays allow only homogeneous elements (same type); Collections allow heterogeneous elements (different types) by default (though Generics improve type safety).

    Java

    ```
    // Array Example (Homogeneous)
    // Student[] stds = new Student[3];
    // stds[0] = new Student();   // No Error
    // stds[1] = new Employee();  // --> Incompatible Types Error
    (compile-time)
    ```

    Java

    ```
    // Collection Example (Heterogeneous - Pre-Generics)
    // import java.util.ArrayList;
    // ArrayList al = new ArrayList();
    // al.add(new Student());  // No Error
    ```

```
// al.add(new Employee()); // No Error
// al.add(new Customer()); // No Error
```

- **Predefined Operations:** Arrays lack built-in methods for common operations like sorting or searching, requiring manual logic. Collections provide rich APIs for these operations.

  Java

  ```
  // Array Example (Manual Sort)
  // int[] ints = {50, 10, 40, 20, 30};
  // // Must provide logic explicitly to sort elements
  ```

  Java

  ```
  // Collection Example (Built-in Sort via data structure)
  // import java.util.TreeSet;
  // TreeSet ts = new TreeSet();
  // ts.add(50);
  // ts.add(10);
  // ts.add(40);
  // ts.add(20);
  // ts.add(30);
  // System.out.println(ts); // TreeSet automatically sorts elements

  // OP: [10, 20, 30, 40, 50]
  ```

- **Flexibility vs. Performance:** Collections are generally more flexible. Arrays can offer better performance if the size and type are known beforehand due to direct memory access.
- **API Dependency:** Arrays have less API dependency. Collections are more API dependent; most operations use predefined methods.
- **Typedness:** Arrays enforce typedness (homogeneous). Collections, before Generics, reduced typedness (heterogeneous allowed); Generics address this for type safety.

Java's **Collection Framework** provides a set of predefined interfaces and classes in the `java.util` package to represent and manipulate collections of objects.

Q) List the classes and interfaces of the Collection Framework which are available in the java.util package?

Ans: (Answer not provided in original notes. Key interfaces include `Collection`, `List`, `Set`, `Queue`, and `Map`, along with their various implementations like `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`, etc.)

Here is an overview of key interfaces in the `java.util` package:

Interface	Description	Key Characteristics
Collection	Root interface for collection hierarchy.	Group of elements.
List	Ordered collection (sequence). Allows duplicates.	Index-based, maintains insertion order.
Set	Collection that does not contain duplicate	No duplicates.

	elements.	
Queue	Collection used to hold elements prior to processing.	Typically FIFO (First-In, First-Out).
Map	Maps keys to values. **Not a true `Collection`** (doesn't inherit from `Collection`).	Stores elements as Key-Value pairs, keys are unique.

Export to Sheets

Q) What is the difference between Collection and Map?

Ans:

- **Collection:** Manages a group of elements as individual entities.
  - *Example:* To represent a list of all employees.
- **Map:** Manages a group of elements as Key-Value pairs.
  - *Example:* To represent a telephone dictionary where phone numbers are keys and customer names are values.

Q) What are the differences between List and Set?

Ans:

Feature	**List**	**Set**
Indexing	Index based.	Not index based (typically based on hash code).
Duplicates	Allows duplicate elements.	Does not allow duplicate elements.
Order	Follows insertion order.	Does not follow insertion order (generally).
Sorting Order	Does not follow sorting order (generally).	Does not follow sorting order (generally).
Heterogeneous	Allows heterogeneous elements.	Allows heterogeneous elements (generally).
Null Elements	Allows any number of null elements.	Allows only one null element.

Export to Sheets

- *Notes on exceptions:*
  - `LinkedHashSet` maintains insertion order (like `List`) but does not allow duplicates.
  - `SortedSet` and `NavigableSet` (implemented by `TreeSet`) follow sorting order and do not allow heterogeneous elements or null elements.

Collection Interface Methods

`java.util.Collection` is the root interface. Key common methods:

1. `public boolean add(Object obj)`: Adds element; returns `true` if added (always `true` for Lists, `false` for Sets if duplicate).

Java

```
import java.util.HashSet;
// import java.util.LinkedHashSet; // Examples use HashSet
// import java.util.TreeSet;

public class Main {
   public static void main(String[] args) {
      HashSet hashSet = new HashSet();
      System.out.println(hashSet.add("AAA")); // true (first time
added)
      System.out.println(hashSet.add("BBB")); // true
      System.out.println(hashSet.add("AAA")); // false (duplicate,
not added)
      System.out.println(hashSet.add("BBB")); // false (duplicate,
not added)
      System.out.println(hashSet); // Order is not guaranteed in
HashSet
   }
}
```

Output:

```
true
true
false
false
[AAA, BBB]
```

2. `public boolean addAll(Collection c)`: Adds all elements from another collection; returns `true` if at least one element was added.

Java

```
import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;

public class Main {
   public static void main(String[] args) {
      HashSet hs1 = new HashSet();
      hs1.add("AAA"); hs1.add("BBB"); hs1.add("CCC"); hs1.add("DDD");
      System.out.println(hs1);

      HashSet hs2 = new HashSet();
      hs2.add("XXX"); hs2.add("YYY"); hs2.add("ZZZ");
      System.out.println(hs2);
      System.out.println(hs1.addAll(hs2)); // true (elements from hs2
added to hs1)
      System.out.println(hs1.addAll(hs2)); // false (no new elements
added from hs2)

      HashSet hs3 = new HashSet();
      hs3.add("AAA"); hs3.add("XXX"); hs3.add("EEE"); hs3.add("FFF");
      System.out.println(hs1.addAll(hs3)); // true (EEE, FFF are new,
AAA, XXX already exist)
```

```
      System.out.println(hs1); // hs1 contains elements from hs1,
hs2, hs3 (no duplicates, order not guaranteed)
   }
}
```

Output:

```
[AAA, CCC, BBB, DDD]
[YYY, XXX, ZZZ]
true
false
true
[AAA, CCC, BBB, EEE, DDD, FFF, YYY, XXX, ZZZ]
```

3. `public boolean remove(Object obj)`: Removes element; returns `true` if removed.

   Java

```
import java.util.ArrayList;
// import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;

public class Main {
   public static void main(String[] args) {
      ArrayList al = new ArrayList();
      al.add("AAA"); al.add("BBB"); al.add("CCC"); al.add("DDD");
      System.out.println(al); // [AAA, BBB, CCC, DDD]

      System.out.println(al.remove("AAA")); // true (AAA was in the
list)
      System.out.println(al); // [BBB, CCC, DDD]

      System.out.println(al.remove("AAA")); // false (AAA is no
longer in the list)
      System.out.println(al); // [BBB, CCC, DDD]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
true
[BBB, CCC, DDD]
false
[BBB, CCC, DDD]
```

4. `public boolean removeAll(Collection c)`: Removes all elements present in the provided collection from the current collection; returns `true` if at least one element was removed.

   Java

```
import java.util.ArrayList;
// import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;
```

```
public class Main {
   public static void main(String[] args) {
      ArrayList al1 = new ArrayList();
      al1.add("AAA"); al1.add("BBB"); al1.add("CCC"); al1.add("DDD");
      System.out.println(al1); // [AAA, BBB, CCC, DDD]

      ArrayList al2 = new ArrayList();
      al2.add("AAA"); al2.add("BBB");
      System.out.println(al2); // [AAA, BBB]

      System.out.println(al1.removeAll(al2)); // true (AAA and BBB
removed from al1)
      System.out.println(al1.removeAll(al2)); // false (no elements
from al2 found in al1 anymore)
      System.out.println(al1); // [CCC, DDD]

      ArrayList al3 = new ArrayList();
      al3.add("CCC"); al3.add("DDD"); al3.add("XXX"); al3.add("YYY");
      System.out.println(al3); // [CCC, DDD, XXX, YYY]
      System.out.println(al1.removeAll(al3)); // true (CCC and DDD
removed from al1)
      System.out.println(al1); // [] // Added to show final state
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB]
true
false
[CCC, DDD]
[CCC, DDD, XXX, YYY]
true
[]
```

5. `public boolean retainAll(Collection c)`: Retains only the elements in the current collection that are also present in the provided collection (removes others); returns `true` if the collection was changed.

   Java

```
import java.util.ArrayList;
// import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;

public class Main {
   public static void main(String[] args) {
      ArrayList al1 = new ArrayList();
      al1.add("AAA"); al1.add("BBB"); al1.add("CCC"); al1.add("DDD");
      System.out.println(al1); // [AAA, BBB, CCC, DDD]

      ArrayList al2 = new ArrayList();
      al2.add("AAA"); al2.add("BBB");
      System.out.println(al2); // [AAA, BBB]
```

```
      System.out.println(al1.retainAll(al2)); // true (CCC and DDD
removed from al1)
      System.out.println(al1); // [AAA, BBB]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB]
true
[AAA, BBB]
```

6. `public int size()`: Returns the number of elements in the collection.
7. `public boolean isEmpty()`: Returns `true` if the collection contains no elements.
8. `public void clear()`: Removes all elements from the collection.

Java

```
import java.util.ArrayList;
// import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;

public class Main {
   public static void main(String[] args) {
      ArrayList al1 = new ArrayList();
      al1.add("AAA"); al1.add("BBB"); al1.add("CCC"); al1.add("DDD");
      System.out.println(al1); // [AAA, BBB, CCC, DDD]

      System.out.println(al1.size()); // 4
      System.out.println(al1.isEmpty()); // false
      al1.clear(); // Remove all elements
      System.out.println(al1); // []
      System.out.println(al1.isEmpty()); // true
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
4
false
[]
true
```

9. `public Object[] toArray()`: Converts the collection elements into an `Object` array.

Java

```
import java.util.ArrayList;
// import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;

public class Main {
```

```
   public static void main(String[] args) {
      ArrayList al1 = new ArrayList();
      al1.add("AAA"); al1.add("BBB"); al1.add("CCC"); al1.add("DDD");
      System.out.println(al1); // [AAA, BBB, CCC, DDD]

      Object[] elements = al1.toArray(); // Convert to Object array
      for (Object element : elements) { // Iterate through array
          System.out.println(element);
      }
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
AAA
BBB
CCC
DDD
```

10. `public boolean contains(Object obj)`: Returns `true` if the collection contains the specified element.

    Java

```
import java.util.ArrayList;
// import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;

public class Main {
   public static void main(String[] args) {
      ArrayList al1 = new ArrayList();
      al1.add("AAA"); al1.add("BBB"); al1.add("CCC"); al1.add("DDD");
      System.out.println(al1); // [AAA, BBB, CCC, DDD]
      System.out.println(al1.contains("BBB")); // true
      System.out.println(al1.contains("XXX")); // false
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
true
false
```

11. `public boolean containsAll(Collection c)`: Returns `true` if the current collection contains all elements of the specified collection.

    Java

```
import java.util.ArrayList;
// import java.util.HashSet;
// import java.util.LinkedHashSet;
// import java.util.TreeSet;

public class Main {
```

```
   public static void main(String[] args) {
      ArrayList al1 = new ArrayList();
      al1.add("AAA"); al1.add("BBB"); al1.add("CCC"); al1.add("DDD");
      System.out.println(al1); // [AAA, BBB, CCC, DDD]

      ArrayList al2 = new ArrayList();
      al2.add("AAA"); al2.add("BBB");
      System.out.println(al2); // [AAA, BBB]
      System.out.println(al1.containsAll(al2)); // true (al1 contains
both AAA and BBB)

      ArrayList al3 = new ArrayList();
      al3.add("XXX"); al3.add("YYY");
      System.out.println(al3); // [XXX, YYY]
      System.out.println(al1.containsAll(al3)); // false (al1 does
not contain XXX or YYY)

      ArrayList al4 = new ArrayList();
      al4.add("AAA"); al4.add("BBB"); al4.add("XXX"); al4.add("YYY");
      System.out.println(al4); // [AAA, BBB, XXX, YYY]
      System.out.println(al1.containsAll(al4)); // false (al1
contains AAA and BBB, but not XXX or YYY)
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB]
true
[XXX, YYY]
false
[AAA, BBB, XXX, YYY]
false
```

List Interface Methods

`java.util.List` is a child interface of `Collection` that represents an ordered collection (sequence). It is index-based and allows duplicate elements.

List provides `Collection` methods plus index-based operations:

1. `public void add(int index, Object element)`: Adds element at the specified index.

   Java

   ```
   import java.util.*; // Imports List and ArrayList

   public class Main {
      public static void main(String[] args) {
         List list = new ArrayList();
         list.add(0, "AAA");
         list.add(1, "BBB");
         list.add(2, "CCC");
         list.add(3, "DDD");
         System.out.println(list); // [AAA, BBB, CCC, DDD]
   ```

```
    }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
```

2. `public boolean addAll(int index, Collection c)`: Adds all elements from a collection starting at the specified index.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list1 = new ArrayList();
      list1.add(0, "AAA"); list1.add(1, "BBB"); list1.add(2, "CCC");
list1.add(3, "DDD");
      System.out.println(list1); // [AAA, BBB, CCC, DDD]

      List list2 = new ArrayList();
      list2.add("XXX"); list2.add("YYY"); list2.add("ZZZ");
      System.out.println(list2); // [XXX, YYY, ZZZ]

      list1.addAll(2, list2); // Insert elements of list2 starting at
index 2 in list1
      System.out.println(list1); // [AAA, BBB, XXX, YYY, ZZZ, CCC,
DDD]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[XXX, YYY, ZZZ]
[AAA, BBB, XXX, YYY, ZZZ, CCC, DDD]
```

3. `public Object get(int index)`: Returns the element at the specified index.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list1 = new ArrayList();
      list1.add(0, "AAA"); list1.add(1, "BBB"); list1.add(2, "CCC");
list1.add(3, "DDD");
      System.out.println(list1); // [AAA, BBB, CCC, DDD]
      System.out.println(); // Add a newline

      System.out.println(list1.get(0));
      System.out.println(list1.get(1));
      System.out.println(list1.get(2));
      System.out.println(list1.get(3));
      System.out.println();
```

```
      for (int i = 0; i < list1.size(); i++) {
          System.out.println(list1.get(i)); // Iterating using index
      }
      System.out.println();
      for(Object o : list1) { // Iterating using enhanced for loop
          System.out.println(o);
      }
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]

AAA
BBB
CCC
DDD

AAA
BBB
CCC
DDD

AAA
BBB
CCC
DDD
```

4. `public Object remove(int index)`: Removes the element at the specified index and returns the removed element.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list1 = new ArrayList();
      list1.add(0, "AAA"); list1.add(1, "BBB"); list1.add(2, "CCC");
list1.add(3, "DDD");
      System.out.println(list1); // [AAA, BBB, CCC, DDD]

      System.out.println(list1.remove(1)); // Remove element at index
1 ("BBB")
      System.out.println(list1); // [AAA, CCC, DDD]

      System.out.println(list1.remove(2)); // Remove element at index
2 ("DDD")
      System.out.println(list1); // [AAA, CCC]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
BBB
```

```
[AAA, CCC, DDD]
DDD
[AAA, CCC]
```

5. `public int indexOf(Object obj)`: Returns the index of the first occurrence of the element, or -1 if not found.
6. `public int lastIndexOf(Object obj)`: Returns the index of the last occurrence of the element, or -1 if not found.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list1 = new ArrayList();
      list1.add(0, "AAA"); list1.add(1, "BBB"); list1.add(2, "CCC");
      list1.add(3, "DDD"); list1.add(4, "BBB"); list1.add(5, "EEE");
list1.add(6, "BBB");
      System.out.println(list1); // [AAA, BBB, CCC, DDD, BBB, EEE,
BBB]

      System.out.println(list1.indexOf("BBB")); // 1 (First
occurrence)
      System.out.println(list1.lastIndexOf("BBB")); // 6 (Last
occurrence)
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD, BBB, EEE, BBB]
1
6
```

7. `public Object set(int index, Object element)`: Replaces the element at the specified index with the new element. Returns the element previously at that position. Throws `IndexOutOfBoundsException` if the index is invalid (outside 0 to size()-1).

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list1 = new ArrayList();
      list1.add(0, "AAA"); list1.add(1, "BBB"); list1.add(2, "CCC");
list1.add(3, "DDD");
      System.out.println(list1); // [AAA, BBB, CCC, DDD]

      System.out.println(list1.set(2, "XXX")); // Replace element at
index 2 ("CCC") with "XXX"
      System.out.println(list1); // [AAA, BBB, XXX, DDD]

      // System.out.println(list1.set(4, "YYY")); // -->
IndexOutOfBoundsException (index 4 is out of bounds)
   }
```

```
}
```

Output:

```
[AAA, BBB, CCC, DDD]
CCC
[AAA, BBB, XXX, DDD]
```

Okay, I have processed the notes document comparing `ArrayList` and `Vector`. I have summarised the content, incorporated the comparison into a table, included all constructors and their code examples, and formatted the output according to your requirements (minimal comments, structure, points, tables, code, Word-friendly).

Here are your summarised and enhanced notes:

Java Collections: ArrayList and Vector

This section covers two common implementations of the `List` interface: `ArrayList` and `Vector`.

ArrayList

- Provided in JDK 1.2.
- Not a legacy collection.
- Implements the `List` interface.
- Index-based.
- Allows duplicate elements.
- Follows insertion order.
- Does not follow sorting order.
- Allows heterogeneous elements (can be restricted with Generics).
- Allows any number of null elements.
- Internal data structure: Resizable Array.
- Initial capacity: 10 elements.
- Incremental capacity ratio: `CurrentCapacity * 3 / 2 + 1`.
- Not synchronized.
- Not thread-safe (allows multiple threads access simultaneously).
- Generally faster due to no synchronization overhead.
- Not guaranteed for data consistency in multi-threaded environments without external synchronization.
- Suggestible for frequent retrieval operations.

Constructors:

1. `public ArrayList()`: Creates an empty ArrayList with initial capacity 10.

   Java

```
import java.util.ArrayList;

public class Main {
   public static void main(String[] args) {
      ArrayList al = new ArrayList();
      // Capacity is implicitly 10
   }
}
```

2. `public ArrayList(int capacity)`: Creates an empty ArrayList with the specified initial capacity.

   Java

```
import java.util.ArrayList;

public class Main {
   public static void main(String[] args) {
      ArrayList al = new ArrayList(15);
      // Capacity is 15
   }
}
```

3. `public ArrayList(Collection c)`: Creates an ArrayList with all elements from the provided Collection.
   - *Reasoning:* Useful for converting other Collection types to ArrayList.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC");
list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]
      ArrayList arrayList1 = new ArrayList(list);
      System.out.println(arrayList1); // [AAA, BBB, CCC, DDD]

      Set set = new HashSet();
      set.add(10); set.add(20); set.add(30); set.add(40);
      System.out.println(set); // Order might differ: [20, 40, 10,
30]
      ArrayList arrayList2 = new ArrayList(set);
      System.out.println(arrayList2); // Order might differ: [20, 40,
10, 30]

      Queue queue = new PriorityQueue();
      queue.add(1000); queue.add(2000); queue.add(3000);
queue.add(4000);
      System.out.println(queue); // Order based on priority: [1000,
2000, 3000, 4000]
      ArrayList arrayList3 = new ArrayList(queue);
      System.out.println(arrayList3); // Order from queue iteration:
[1000, 2000, 3000, 4000]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
[20, 40, 10, 30]
[20, 40, 10, 30]
[1000, 2000, 3000, 4000]
[1000, 2000, 3000, 4000]
```

ArrayList Example (General Use):

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      ArrayList arrayList = new ArrayList();
      arrayList.add("AAA");
      arrayList.add("BBB");
      arrayList.add("CCC");
      arrayList.add("DDD");
      System.out.println(arrayList); // [AAA, BBB, CCC, DDD]
      arrayList.add("BBB"); // Add duplicate
      arrayList.add("CCC"); // Add duplicate
      System.out.println(arrayList); // [AAA, BBB, CCC, DDD, BBB, CCC]
      arrayList.add(10); // Add heterogeneous
      arrayList.add(22.22f);
      arrayList.add(true);
      System.out.println(arrayList); // [AAA, BBB, CCC, DDD, BBB, CCC, 10,
22.22, true]
      arrayList.add(null); // Add null
      arrayList.add(null); // Add null
      System.out.println(arrayList); // [AAA, BBB, CCC, DDD, BBB, CCC, 10,
22.22, true, null, null]

      System.out.println(arrayList.get(0)); // Access by index
      System.out.println(arrayList.get(1));
      System.out.println();

      for (int i = 0; i < arrayList.size(); i++) {
          System.out.println(arrayList.get(i)); // Iterate by index
      }
      System.out.println();
      for(Object o : arrayList) { // Iterate using enhanced for loop
          System.out.println(o);
      }
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD, BBB, CCC]
[AAA, BBB, CCC, DDD, BBB, CCC, 10, 22.22, true]
[AAA, BBB, CCC, DDD, BBB, CCC, 10, 22.22, true, null, null]
AAA
BBB

AAA
```

```
BBB
CCC
DDD
BBB
CCC
10
22.22
true
null
null

AAA
BBB
CCC
DDD
BBB
CCC
10
22.22
true
null
null
```

Vector

- Provided in JDK 1.0.
- A legacy collection.
- Implements the `List` interface.
- Index-based.
- Allows duplicate elements.
- Follows insertion order.
- Does not follow sorting order.
- Allows heterogeneous elements (can be restricted with Generics).
- Allows any number of null elements.
- Internal data structure: Resizable Array.
- Initial capacity: 10 elements.
- Incremental capacity ratio: `2 * CurrentCapacity` (doubles by default).
- **Synchronized.**
- **Thread-safe** (only one thread accesses at a time).
- Generally slower due to synchronization overhead.
- Guaranteed for data consistency in multi-threaded environments.
- Suggestible for frequent retrieval operations *in a thread-safe context*.

Constructors:

1. `public Vector()`: Creates an empty Vector with default initial capacity 10.

   Java

   ```
   import java.util.*;

   public class Main {
      public static void main(String[] args) {
         Vector vector = new Vector();
   ```

```
      System.out.println(vector.capacity()); // 10
   }
}
```

Output:

```
10
```

2. `public Vector(int capacity)`: Creates an empty Vector with the specified initial capacity.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Vector vector = new Vector(20);
      System.out.println(vector.capacity()); // 20
   }
}
```

   Output:

```
20
```

3. `public Vector(int capacity, int incrementalRatio)`: Creates a Vector with the specified initial capacity and incremental ratio.
   - *Reasoning:* Allows custom control over how much the Vector grows when needed.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Vector vector = new Vector(5); // Initial capacity 5
      System.out.println(vector.capacity()); // 5
      for (int i = 1; i <= 6; i++) { // Add more than initial
capacity
         vector.add(i);
      }
      System.out.println(vector.capacity()); // 10 (Doubled)
      for (int i = 7; i <= 11; i++) { // Add more than current
capacity
         vector.add(i);
      }
      System.out.println(vector.capacity()); // 20 (Doubled again)
      for (int i = 12; i <= 21; i++) { // Add more
         vector.add(i);
      }
      System.out.println(vector.capacity()); // 40 (Doubled again)
   }
}
```

Output:

```
5
10
20
40
```

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Vector vector = new Vector(5, 5); // Initial capacity 5,
increment 5
      System.out.println(vector.capacity()); // 5
      for (int i = 1; i <= 6; i++) { // Add more than initial
capacity
         vector.add(i);
      }
      System.out.println(vector.capacity()); // 10 (Increased by 5)
      for (int i = 7; i <= 11; i++) { // Add more
         vector.add(i);
      }
      System.out.println(vector.capacity()); // 15 (Increased by 5)
      for (int i = 12; i <= 16; i++) { // Add more
         vector.add(i);
      }
      System.out.println(vector.capacity()); // 20 (Increased by 5)
   }
}
```

Output:

```
5
10
15
20
```

4. `public Vector(Collection c)`: Creates a Vector with all elements from the provided Collection.
    - *Reasoning:* Useful for converting other Collection types to Vector.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC");
list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]
      Vector vector1 = new Vector(list);
      System.out.println(vector1); // [AAA, BBB, CCC, DDD]

      Set set = new HashSet();
      set.add(10); set.add(20); set.add(30); set.add(40);
```

```
      System.out.println(set); // Order might differ: [20, 40, 10,
30]
      Vector vector2 = new Vector(set);
      System.out.println(vector2); // Order might differ: [20, 40,
10, 30]

      Queue queue = new PriorityQueue();
      queue.add(1000); queue.add(2000); queue.add(3000);
queue.add(4000);
      System.out.println(queue); // Order based on priority: [1000,
2000, 3000, 4000]
      Vector vector3 = new Vector(queue);
      System.out.println(vector3); // Order from queue iteration:
[1000, 2000, 3000, 4000]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
[20, 40, 10, 30]
[20, 40, 10, 30]
[1000, 2000, 3000, 4000]
[1000, 2000, 3000, 4000]
```

Vector-specific Methods: Vector retains some older method names from its legacy past, in addition to implementing the `List` interface methods.

1. `public void addElement(Object element)`: Adds an element to the end of the Vector (similar to `add()`).
2. `public boolean removeElement(Object obj)`: Removes the first occurrence of the element (similar to `remove()`).
3. `public void removeElementAt(int index)`: Removes the element at the specified index (similar to `remove(int)`).
4. `public void removeAllElements()`: Removes all elements (similar to `clear()`).

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Vector vector = new Vector();
      vector.addElement("AAA");
      vector.addElement("BBB");
      vector.addElement("CCC");
      vector.addElement("DDD");
      System.out.println(vector); // [AAA, BBB, CCC, DDD]
      vector.removeElement("BBB");
      System.out.println(vector); // [AAA, CCC, DDD]
      vector.removeElementAt(2); // Removes element at index 2 (which
is "DDD" now)
      System.out.println(vector); // [AAA, CCC]
      vector.removeAllElements();
      System.out.println(vector); // []
   }
```

```
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, CCC, DDD]
[AAA, CCC]
[]
```

5. `public Object elementAt(int index)`: Returns the element at the specified index (similar to `get()`).
6. `public Object firstElement()`: Returns the first element.
7. `public Object lastElement()`: Returns the last element.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Vector vector = new Vector();
      vector.addElement("AAA");
      vector.addElement("BBB");
      vector.addElement("CCC");
      vector.addElement("DDD");
      System.out.println(vector); // [AAA, BBB, CCC, DDD]
      System.out.println(vector.elementAt(1)); // BBB
      System.out.println(vector.firstElement()); // AAA
      System.out.println(vector.lastElement()); // DDD
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
BBB
AAA
DDD
```

ArrayList vs. Vector Comparison

Q) What are the differences between ArrayList and Vector?

Ans:

Feature	**ArrayList**	**Vector**	***Reasoning/Notes***
Introduced in	JDK 1.2	JDK 1.0	Vector is an older, "legacy" class.
Legacy Status	No	Yes	Vector is part of the original collection classes.
Synchronization	Not synchronized	**Synchronized**	Key difference impacting thread safety & performance.

Thread Safety	Not thread-safe	**Thread-safe**	Vector's methods are synchronized.
Performance	Faster (no sync overhead)	Slower (sync overhead)	Synchronization adds performance cost.
Data Consistency	Not guaranteed (multi-thread)	Guaranteed (multi-thread)	Synchronization ensures consistency.
Thread Access	Multiple threads parallel	One thread sequential	Due to synchronization.
`capacity()` Method	No	Yes	Vector exposes its internal capacity.
Incremental Capacity	Fixed (Current*1.5 + 1)	Customizable (default doubles)	Vector has a constructor for custom growth factor.
Default Growth	`(CurrentCapacity * 3 / 2) + 1`	`2 * CurrentCapacity`	How they resize when capacity is exceeded.

ava Collections: Stack, LinkedList, Cursors, Set, HashSet

Stack

- Extends `java.util.Vector`.
- Follows **LIFO** (Last-In, First-Out) principle.

Constructor:

1. `public Stack()`: Creates an empty Stack.

Methods:

1. `public E push(E item)`: Adds element to the top.
2. `public E pop()`: Reads and removes the top element. Throws `EmptyStackException` if stack is empty.
3. `public E peek()`: Reads the top element without removing. Throws `EmptyStackException` if stack is empty.
4. `public int search(Object obj)`: Searches for element, returns 1-based position from the top if found, -1 otherwise.

Java
```
import java.util.*; // Imports Stack

public class Main {
   public static void main(String[] args) {
      Stack stack = new Stack();
      stack.push("AAA");
      stack.push("BBB");
      stack.push("CCC");
      stack.push("DDD");
      stack.push("EEE");
      stack.push("FFF");
```

```
      System.out.println(stack); // [AAA, BBB, CCC, DDD, EEE, FFF]
(Elements are listed from bottom to top)
      System.out.println(stack.peek()); // FFF (Top element)
      System.out.println(stack.pop()); // FFF (Read and remove top element)
      System.out.println(stack); // [AAA, BBB, CCC, DDD, EEE] (FFF is
removed)
      System.out.println(stack.search("EEE")); // 1 (EEE is 1st from top)
      System.out.println(stack.search("XXX")); // -1 (XXX not found)
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD, EEE, FFF]
FFF
FFF
[AAA, BBB, CCC, DDD, EEE]
1
-1
```

LinkedList

- Provided in JDK 1.2.
- Not a legacy collection.
- Implements `List` and `Deque` interfaces.
- Index-based (from List).
- Allows duplicate elements.
- Follows insertion order.
- Does not follow sorting order.
- Allows heterogeneous elements (can be restricted with Generics).
- Allows any number of null elements.
- Internal data structure: Doubly Linked List.
- Suggestible for frequent insertions and deletions (efficient due to linked structure).
- Not synchronized.
- Not thread-safe.
- Generally faster than Vector but slower than ArrayList for random access (get/set) due to traversal.

Constructors:

1. `public LinkedList()`: Creates an empty LinkedList.

   Java

   ```
   import java.util.*;

   public class Main {
      public static void main(String[] args) {
         LinkedList ll = new LinkedList();
         System.out.println(ll); // []
      }
   }
   ```

   Output:

`[]`

2. `public LinkedList(Collection c)`: Creates a LinkedList with all elements from the provided Collection.
    - *Reasoning:* Used to convert elements from other collections to LinkedList.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC");
list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]
      LinkedList linkedList1 = new LinkedList(list);
      System.out.println(linkedList1); // [AAA, BBB, CCC, DDD]

      Set set = new HashSet();
      set.add(10); set.add(20); set.add(30); set.add(40);
      System.out.println(set); // Order might differ: [20, 40, 10,
30]
      LinkedList linkedList2 = new LinkedList(set);
      System.out.println(linkedList2); // Order might differ: [20,
40, 10, 30]

      Queue queue = new PriorityQueue();
      queue.add(1000); queue.add(2000); queue.add(3000);
queue.add(4000);
      System.out.println(queue); // Order based on priority: [1000,
2000, 3000, 4000]
      LinkedList linkedList3 = new LinkedList(queue);
      System.out.println(linkedList3); // Order from queue iteration:
[1000, 2000, 3000, 4000]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
[20, 40, 10, 30]
[20, 40, 10, 30]
[1000, 2000, 3000, 4000]
[1000, 2000, 3000, 4000]
```

LinkedList-specific Methods: LinkedList implements methods from `List`, `Deque`, `Queue`. Some common ones:

1. `public void addFirst(E e)`: Adds element to the front.
2. `public void addLast(E e)`: Adds element to the end (same as `add()`).
3. `public E getFirst()`: Returns first element.
4. `public E getLast()`: Returns last element.
5. `public E removeFirst()`: Removes and returns the first element.
6. `public E removeLast()`: Removes and returns the last element.

Java
```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      LinkedList linkedList = new LinkedList();
      linkedList.add("AAA");
      linkedList.add("BBB");
      linkedList.add("CCC");
      linkedList.add("DDD");
      System.out.println(linkedList); // [AAA, BBB, CCC, DDD]
      linkedList.addFirst("XXX"); // Add to front
      linkedList.addLast("YYY"); // Add to end
      System.out.println(linkedList); // [XXX, AAA, BBB, CCC, DDD, YYY]

      linkedList.removeFirst(); // Remove from front
      linkedList.removeLast(); // Remove from end
      System.out.println(linkedList); // [AAA, BBB, CCC, DDD]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[XXX, AAA, BBB, CCC, DDD, YYY]
[AAA, BBB, CCC, DDD]
```

Cursors (Iterators) in Collections

Cursors or Iterators are used to traverse collection elements one by one.

- *Reasoning:* Iterators provide a standard way to access elements sequentially, unlike `toString()` which prints all elements at once.

Java
```
// Example showing toString() behavior
import java.util.*;
class A{ } // Custom class without overridden toString()
public class Main {
   public static void main(String[] args) {
      A a = new A();
      System.out.println(a); // Calls Object.toString() ->
ClassName@hashCode
      ArrayList list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC"); list.add("DDD");
      System.out.println(list); // Calls ArrayList.toString() -> prints
elements
   }
}
```

Output Example: (HashCode will vary)

```
A@abcdef12
[AAA, BBB, CCC, DDD]
```

Three types of Cursors in Collection Framework: `Enumeration`, `Iterator`, `ListIterator`.

Enumeration

- Interface provided in JDK 1.0.
- Legacy Cursor, used mainly for Legacy Collections like `Vector`.
- Allows only **read** operation.
- Allows reading in **forward** direction only.

Steps:

1. Create a Legacy Collection (e.g., `Vector`).
2. Get `Enumeration` object using `collection.elements()`.
3. Read elements using `hasMoreElements()` (check if more elements) and `nextElement()` (get next element).

Java
```
import java.util.*; // Imports Vector and Enumeration

public class Main {
   public static void main(String[] args) {
      Vector v = new Vector();
      v.add("AAA"); v.add("BBB"); v.add("CCC"); v.add("DDD"); v.add("EEE");
v.add("FFF");
      System.out.println(v); // [AAA, BBB, CCC, DDD, EEE, FFF]

      Enumeration e = v.elements(); // Get Enumeration
      while (e.hasMoreElements()) { // Check if more elements
          System.out.println(e.nextElement()); // Get next element
      }
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD, EEE, FFF]
AAA
BBB
CCC
DDD
EEE
FFF
```

Drawbacks: Read-only, forward-only, limited to Legacy Collections.

Iterator

- Interface provided in JDK 1.2.
- Not a legacy cursor.
- **Universal Cursor** - applicable for all Collections.
- Allows **read and remove** operations.
- Allows reading in **forward** direction only.

Steps:

1. Get `Iterator` object using `collection.iterator()`.

2. Read/Remove using `hasNext()` (check if next element), `next()` (get next element), and `remove()` (remove the *last element returned by next*).

Java
```
import java.util.*; // Imports ArrayList and Iterator

public class Main {
   public static void main(String[] args) {
      ArrayList list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC"); list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]

      Iterator iterator = list.iterator(); // Get Iterator
      while (iterator.hasNext()) { // Check for next element
          System.out.println(iterator.next()); // Get next element
      }
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
AAA
BBB
CCC
DDD
```

Example (Iterator remove):

Java
```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      ArrayList list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC"); list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]

      Iterator iterator = list.iterator();
      while (iterator.hasNext()) {
          String element = (String) iterator.next(); // Get next element
(must cast if not using Generics)
          if (element.equals("CCC")) {
              iterator.remove(); // Remove the element that was just
returned by next()
          }
          System.out.println(element); // Print element AFTER potential
removal
      }
      System.out.println(list); // Print list after iteration and removal
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
AAA
```

```
BBB
CCC
DDD
[AAA, BBB, DDD]
```

Drawbacks: Read and remove only, forward-only.

ListIterator

- Interface provided in JDK 1.2.
- Not a legacy cursor.
- Applicable **only for List implementations**.
- Allows reading in **both forward and backward** directions.
- Allows **read, remove, insert (add), and replace (set)** operations.

Methods:

1. `public boolean hasNext()`: Checks for next element in forward direction.
2. `public boolean hasPrevious()`: Checks for previous element in backward direction.
3. `public E next()`: Reads next element in forward direction.
4. `public E previous()`: Reads previous element in backward direction.
5. `public int nextIndex()`: Returns index of element that would be returned by `next()`.
6. `public int previousIndex()`: Returns index of element that would be returned by `previous()`.
7. `public void remove()`: Removes the last element returned by `next()` or `previous()`.
8. `public void set(E e)`: Replaces the last element returned by `next()` or `previous()`.
9. `public void add(E e)`: Inserts element at the current position (before the element that would be returned by `next()`).

Java

```
import java.util.*; // Imports ArrayList and ListIterator

public class Main {
   public static void main(String[] args) {
      ArrayList list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC"); list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]
      System.out.println();

      ListIterator listIterator = list.listIterator(); // Get ListIterator

      // Forward Traversal
      while (listIterator.hasNext()) {
          System.out.println(listIterator.nextIndex() + " : " +
listIterator.next());
      }
      System.out.println();

      // Backward Traversal (starting from the end, where the iterator is
positioned after hasNext loop)
      while (listIterator.hasPrevious()) {
          System.out.println(listIterator.previousIndex() + " : " +
listIterator.previous());
      }
   }
```

```
}
```

Output:

```
[AAA, BBB, CCC, DDD]

0 : AAA
1 : BBB
2 : CCC
3 : DDD

2 : DDD
1 : CCC
0 : BBB
-1 : AAA // Note: previousIndex() returns index *before* the element, so
for the first element it's -1
```

Example (ListIterator modifications: add, set, remove):

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      ArrayList list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC"); list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]
      System.out.println(); // Add a newline for separation

      ListIterator listIterator = list.listIterator();
      while (listIterator.hasNext()) {
          String element = (String) listIterator.next(); // Get next
element

          if (element.equals("BBB")) {
              listIterator.add("XXX"); // Insert "XXX" after "BBB"
          }
          if (element.equals("CCC")) {
              listIterator.set("YYY"); // Replace "CCC" with "YYY"
          }
          if (element.equals("DDD")) {
              listIterator.remove(); // Remove "DDD"
          }
      }
      System.out.println(list); // [AAA, BBB, XXX, YYY]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]

[AAA, BBB, XXX, YYY]
```

Cursor Comparison

Q) What are the differences between Enumeration, Iterator and ListIterator?

Ans:

Feature	Enumeration	Iterator	ListIterator	*Reasoning/Notes*
Introduced in	JDK 1.0	JDK 1.2	JDK 1.2	Modern iterators introduced later.
Legacy Status	Yes	No	No	Enumeration is from the original Collection classes.
Applicable For	Legacy Collections only	All Collections	List implementations only	Iterator is universal; ListIterator is List-specific.
Operations	Read only	Read, Remove	Read, Remove, Insert, Replace	ListIterator offers full modification control.
Direction	Forward only	Forward only	Forward and Backward	ListIterator allows bi-directional traversal.
Hierarchy	Independent interface	Independent interface	Child of Iterator	ListIterator extends Iterator's capabilities.

Export to Sheets

Set Interface

- Interface provided in JDK 1.2.
- Child interface of `Collection`.
- Does **not** allow duplicate elements.

Characteristics:

- Not index-based (elements stored based on hash codes).
- Does not follow insertion order (generally).
- Does not follow sorting order (generally).
- Allows heterogeneous elements (generally, can be restricted with Generics).
- Allows maximum one null element.
- *Notes on exceptions:* `LinkedHashSet` maintains insertion order. `SortedSet` (`TreeSet`) follows sorting order, does not allow heterogeneous elements or nulls.

HashSet

- Class provided in JDK 1.2.
- Not a legacy collection.
- Implements the `Set` interface.
- Not index-based.
- Does not follow insertion order.
- Does not follow sorting order.
- Does not allow duplicate elements.
- Allows heterogeneous elements.
- Allows one null element.

- Internal data structure: `HashMap`.
- Suggestible for frequent search operations (due to hashing).
- Default initial capacity: 16 elements.
- Default load factor: 0.75 (75%).
- Not synchronized.
- Not thread-safe.
- Faster (no sync overhead).
- Not guaranteed for data consistency in multi-threaded environments.

Constructors:

1. `public HashSet()`: Creates an empty HashSet with default capacity (16) and load factor (0.75).

   Java

   ```
   import java.util.HashSet;

   public class Main {
      public static void main(String[] args) {
         HashSet hs = new HashSet();
         System.out.println(hs); // []
      }
   }
   ```

   Output:

   ```
   []
   ```

2. `public HashSet(int capacity)`: Creates an empty HashSet with specified initial capacity and default load factor (0.75).

   Java

   ```
   import java.util.HashSet;

   public class Main {
      public static void main(String[] args) {
         HashSet hs = new HashSet(20);
         System.out.println(hs); // []
      }
   }
   ```

   Output:

   ```
   []
   ```

3. `public HashSet(int capacity, float loadFactor)`: Creates an empty HashSet with specified initial capacity and load factor.

   Java

   ```
   import java.util.HashSet;
   ```

```
public class Main {
   public static void main(String[] args) {
      HashSet hs = new HashSet(20, 0.85f);
      System.out.println(hs); // []
   }
}
```

Output:

```
[]
```

4. `public HashSet(Collection c)`: Creates a HashSet with all elements from the provided Collection.
    - *Reasoning:* Useful for converting elements from other collection types to HashSet.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      List list = new ArrayList();
      list.add("AAA"); list.add("BBB"); list.add("CCC");
list.add("DDD");
      System.out.println(list); // [AAA, BBB, CCC, DDD]
      HashSet hashSet1 = new HashSet(list);
      System.out.println(hashSet1); // Order is not guaranteed for
HashSet: [AAA, CCC, BBB, DDD]
      System.out.println();

      Set set = new HashSet(); // Create a source Set
      set.add(10); set.add(20); set.add(30); set.add(40);
      System.out.println(set); // Order might differ: [20, 40, 10,
30]
      HashSet hashSet2 = new HashSet(set); // Convert Set to HashSet
(no change in content/order guarantees)
      System.out.println(hashSet2); // Order might differ: [20, 40,
10, 30]
      System.out.println();

      Queue queue = new PriorityQueue();
      queue.add(1000); queue.add(2000); queue.add(3000);
queue.add(4000);
      System.out.println(queue); // Order based on priority: [1000,
2000, 3000, 4000]
      HashSet hashSet3 = new HashSet(queue); // Convert Queue to
HashSet (order not guaranteed)
      System.out.println(hashSet3); // Order might differ: [2000,
4000, 1000, 3000]
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[AAA, CCC, BBB, DDD]
```

```
[20, 40, 10, 30]
[20, 40, 10, 30]

[1000, 2000, 3000, 4000]
[2000, 4000, 1000, 3000]
```

HashSet Example (General Use):

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      HashSet hashSet = new HashSet();
      hashSet.add("AAA");
      hashSet.add("BBB");
      hashSet.add("CCC");
      hashSet.add("DDD");
      hashSet.add("EEE");
      hashSet.add("FFF");
      System.out.println(hashSet); // [AAA, CCC, BBB, EEE, DDD, FFF] (Order
not guaranteed)
      hashSet.add("EEE"); // Add duplicate (will be ignored)
      hashSet.add("FFF"); // Add duplicate (will be ignored)
      System.out.println(hashSet); // [AAA, CCC, BBB, EEE, DDD, FFF] (No
change)
      hashSet.add(null); // Add null
      hashSet.add(null); // Add null (duplicate null will be ignored)
      System.out.println(hashSet); // [null, AAA, CCC, BBB, EEE, DDD, FFF]
(null added, order might shift)
      hashSet.add(100); // Add heterogeneous
      hashSet.add(22.22f); // Add heterogeneous
      System.out.println(hashSet); // [null, AAA, CCC, BBB, EEE, 100, DDD,
FFF, 22.22] (Order might shift)
   }
}
```

Output:

```
[AAA, CCC, BBB, EEE, DDD, FFF]
[AAA, CCC, BBB, EEE, DDD, FFF]
[null, AAA, CCC, BBB, EEE, DDD, FFF]
[null, AAA, CCC, BBB, EEE, 100, DDD, FFF, 22.22]
```

Java Collections: Set Implementations and Map

This section covers `LinkedHashSet`, sorted `Set` interfaces (`SortedSet`, `NavigableSet`), their implementation (`TreeSet`), and the `Map` interface.

LinkedHashSet

- Provided in JDK 1.4.

- Implements `Set` interface.
- Maintains insertion order.
- Internal data structure: Hash table and Linked list.

Q) What are the differences between HashSet and LinkedHashSet?

Ans:

Feature	HashSet	LinkedHashSet	*Reasoning/Notes*
Introduced in	JDK 1.2	JDK 1.4	LinkedHashSet is newer.
Insertion Order	Does not follow insertion order	**Follows insertion order**	LinkedHashSet uses a linked list to maintain order.
Internal DS	HashMap	HashMap + LinkedList	LinkedHashSet uses a linked list for iteration order.

Export to Sheets

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      HashSet hashSet = new HashSet(); // Order not guaranteed
      hashSet.add("AAA"); hashSet.add("BBB"); hashSet.add("CCC");
      hashSet.add("DDD"); hashSet.add("EEE"); hashSet.add("FFF");
      System.out.println(hashSet);

      LinkedHashSet linkedHashSet = new LinkedHashSet(); // Maintains
insertion order
      linkedHashSet.add("AAA"); linkedHashSet.add("BBB");
linkedHashSet.add("CCC");
      linkedHashSet.add("DDD"); linkedHashSet.add("EEE");
linkedHashSet.add("FFF");
      System.out.println(linkedHashSet);
   }
}
```

Output Example: (HashSet order may vary, LinkedHashSet order is fixed)

```
[AAA, CCC, BBB, EEE, DDD, FFF] // HashSet order
[AAA, BBB, CCC, DDD, EEE, FFF] // LinkedHashSet order
```

SortedSet Interface

- Interface provided in JDK 1.2.
- Child interface of `Set`.
- Does not allow duplicate elements.
- **Follows Sorting Order**.
- Requires elements to be **Homogeneous** (`ClassCastException` otherwise).
- Does not allow null elements (`NullPointerException` otherwise).
- Not index-based.
- Does not follow insertion order.

Methods:

1. `public Object first()`: Returns the first element (lowest in sorting order).
2. `public Object last()`: Returns the last element (highest in sorting order).
3. `public SortedSet headSet(Object toElement)`: Returns a view of the portion of this set whose elements are strictly less than `toElement`.
4. `public SortedSet tailSet(Object fromElement)`: Returns a view of the portion of this set whose elements are greater than or equal to `fromElement`.
5. `public SortedSet subSet(Object fromElement, Object toElement)`: Returns a view of the portion of this set whose elements range from `fromElement` (inclusive) to `toElement` (exclusive).

Java
```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      SortedSet sortedSet = new TreeSet(); // TreeSet is a common
implementation
      sortedSet.add("AAA"); sortedSet.add("EEE"); sortedSet.add("BBB");
      sortedSet.add("DDD"); sortedSet.add("CCC"); sortedSet.add("FFF");
      System.out.println(sortedSet); // Automatically sorted

      System.out.println(sortedSet.first()); // First element
      System.out.println(sortedSet.last()); // Last element
      System.out.println(sortedSet.headSet("DDD")); // Elements < "DDD"
      System.out.println(sortedSet.tailSet("DDD")); // Elements >= "DDD"
      System.out.println(sortedSet.subSet("BBB", "EEE")); // Elements >=
"BBB" and < "EEE"
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD, EEE, FFF]
AAA
FFF
[AAA, BBB, CCC]
[DDD, EEE, FFF]
[BBB, CCC, DDD]
```

NavigableSet Interface

- Interface provided in JDK 1.6.
- Child interface of `SortedSet`.
- Provides navigation methods for finding elements based on their relationship to a given element (e.g., closest greater/lesser).

Methods:

1. `public NavigableSet descendingSet()`: Returns a reverse order view of the elements.
2. `public E ceiling(E e)`: Returns the least element greater than or equal to `e`.
3. `public E higher(E e)`: Returns the least element strictly greater than `e`.
4. `public E floor(E e)`: Returns the greatest element less than or equal to `e`.

5. `public E lower(E e)`: Returns the greatest element strictly less than `e`.
6. `public E pollFirst()`: Retrieves and removes the first (lowest) element.
7. `public E pollLast()`: Retrieves and removes the last (highest) element.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      NavigableSet navigableSet = new TreeSet(); // TreeSet implements
NavigableSet
      navigableSet.add("AAA"); navigableSet.add("FFF");
navigableSet.add("BBB");
      navigableSet.add("EEE"); navigableSet.add("CCC");
navigableSet.add("DDD");
      System.out.println(navigableSet); // [AAA, BBB, CCC, DDD, EEE, FFF]

      System.out.println(navigableSet.descendingSet()); // [FFF, EEE, DDD,
CCC, BBB, AAA]
      System.out.println(navigableSet.ceiling("DDD")); // DDD
      System.out.println(navigableSet.higher("DDD")); // EEE
      System.out.println(navigableSet.floor("DDD")); // DDD
      System.out.println(navigableSet.lower("DDD")); // CCC

      System.out.println(navigableSet.pollFirst()); // AAA (Removed)
      System.out.println(navigableSet.pollLast()); // FFF (Removed)
      System.out.println(navigableSet); // [BBB, CCC, DDD, EEE] (Remaining
elements)
   }
}
```

Output:

```
[AAA, BBB, CCC, DDD, EEE, FFF]
[FFF, EEE, DDD, CCC, BBB, AAA]
DDD
EEE
DDD
CCC
AAA
FFF
[BBB, CCC, DDD, EEE]
```

TreeSet

- Class provided in JDK 1.2.
- Not a legacy collection.
- Implements `NavigableSet`, `SortedSet`, and `Set`.
- Internal data structure: Balanced Tree (like a Red-Black Tree).
- Stores elements in **sorting order**.
- Does not allow duplicate elements.
- Does not allow heterogeneous elements (`ClassCastException`).
- Does not allow null elements (`NullPointerException`).
- Requires elements to be **Comparable** or use a **Comparator**.
- Not synchronized, not thread-safe. Generally fast for search/retrieval of sorted data.

Reasoning: TreeSet uses the `compareTo()` method (from `Comparable`) or `compare()` method (from `Comparator`) to determine the order of elements and to check for duplicates (if `compareTo`/`compare` returns 0, elements are considered equal).

Comparable vs. Comparator:

- **`java.lang.Comparable`:** Defines the "natural" sorting order for a class. A class implements `Comparable` to define how its *instances* should be compared to *other instances of the same class*. Has a single method `compareTo(Object obj)`.
    - `str1.compareTo(str2)`: Returns negative if `str1` comes before `str2`, positive if `str1` comes after `str2`, and zero if they are equal.

  Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      String str1 = new String("abc");
      String str2 = new String("def");
      String str3 = new String("abc");

      System.out.println(str1.compareTo(str2)); // -ve (abc comes
before def)
      System.out.println(str2.compareTo(str3)); // +ve (def comes
after abc)
      System.out.println(str3.compareTo(str1)); // 0 (abc is same as
abc)
   }
}
```

  Output:

```
-3
3
0
```

- **`java.util.Comparator`:** Defines a custom sorting order. You create a separate class that implements `Comparator` to define how to compare *objects of some other class*. Useful when you can't modify the class (e.g., `String`) or need multiple sorting criteria. Has methods `compare(Object obj1, Object obj2)` and `equals(Object obj)`.
    - `compare(obj1, obj2)`: Returns negative if `obj1` comes before `obj2`, positive if `obj1` comes after `obj2`, and zero if they are equal, according to the Comparator's logic.

Q) What is the difference between Comparable and Comparator?

Ans:

Feature	**Comparable**	**Comparator**	***Reasoning/Notes***
Package	`java.lang`	`java.util`	
Method(s)	`compareTo(Object)`	`compare(Object,`	Comparator needs two

		`Object), equals(Object)`	objects to compare.
Sorting Type	Implicit (Natural Sorting)	Explicit (Custom Sorting)	Defines the class's *own* order vs. an external order.
Element Requirement	Elements *must* implement it	No element implements it; a separate Comparator class is used.	Where the sorting logic resides.
Predefined Examples	`String`, Wrapper classes, etc.	None (you implement it)	Standard types support natural sorting.

Export to Sheets

TreeSet Sorting Logic: TreeSet uses the comparison result (`compareTo` or `compare`) to build a balanced binary search tree.

- -ve result: Current element goes to the left subtree.
- +ve result: Current element goes to the right subtree.
- 0 result: Elements are considered equal; the current element is a duplicate and discarded.

In-order traversal (Left -> Root -> Right) of the balanced tree yields elements in sorted order.

Using Comparable with TreeSet: Elements added to a `TreeSet` (created with the no-arg constructor) must implement `java.lang.Comparable`.

Java

```
import java.util.*;

// Employee class must implement Comparable to be stored in a default
TreeSet
class Employee implements Comparable<Employee> {
   private int eno;
   private String ename;
   private float esal;
   private String eaddr;

   public Employee(int eno, String ename, float esal, String eaddr) {
       this.eno = eno; this.ename = ename; this.esal = esal; this.eaddr =
eaddr;
   }

   @Override
   public String toString() { // Provides a readable string representation
       return "Employee{" + "eno=" + eno + ", ename='" + ename + '\'' + ",
esal=" + esal + ", eaddr='" + eaddr + '\'' + '}';
   }

   @Override
   // Example compareTo implementation (sorting by address in reverse
order)
   public int compareTo(Employee emp) {
       int val = this.eaddr.compareTo(emp.eaddr); // Compare addresses
(natural String order)
       return -val; // Return -val for reverse order
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      Employee emp1 = new Employee(111,"Durga", 50000, "Hyderabad");
      Employee emp2 = new Employee(222,"Anil", 60000, "Chennai");
      Employee emp3 = new Employee(333,"Ramesh", 70000, "Pune");
      Employee emp4 = new Employee(444,"Venkat", 80000, "Delhi");
      Employee emp5 = new Employee(555,"Ramana", 90000, "Mumbai");

      TreeSet treeSet = new TreeSet(); // Uses natural sorting (based on
Employee.compareTo)
      treeSet.add(emp1); // Calls emp1.compareTo(...)
      treeSet.add(emp2); // Calls emp2.compareTo(emp1), etc.
      treeSet.add(emp3);
      treeSet.add(emp4);
      treeSet.add(emp5);
      System.out.println(treeSet);
   }
}
```

Output Example: (Sorted by address in reverse alphabetical order)

```
[Employee{eno=333, ename='Ramesh', esal=70000.0, eaddr='Pune'},
Employee{eno=555, ename='Ramana', esal=90000.0, eaddr='Mumbai'},
Employee{eno=111, ename='Durga', esal=50000.0, eaddr='Hyderabad'},
Employee{eno=444, ename='Venkat', esal=80000.0, eaddr='Delhi'},
Employee{eno=222, ename='Anil', esal=60000.0, eaddr='Chennai'}]
```

Using Comparator with TreeSet: Create a separate class implementing `Comparator` and pass an instance to the `TreeSet` constructor.

Java

```
import java.util.*;

// Employee class does NOT need to implement Comparable here
class Employee {
   private int eno;
   private String ename;
   private float esal;
   private String eaddr;

   public Employee(int eno, String ename, float esal, String eaddr) {
       this.eno = eno; this.ename = ename; this.esal = esal; this.eaddr =
eaddr;
   }

   // Getters needed for the Comparator to access private fields
   public int getEno() { return eno; }
   public String getEname() { return ename; }
   public float getEsal() { return esal; }
   public String getEaddr() { return eaddr; }

   @Override
   public String toString() {
       return "Employee{" + "eno=" + eno + ", ename='" + ename + '\'' + ",
esal=" + esal + ", eaddr='" + eaddr + '\'' + '}';
   }
}
```

```
// Separate class implements Comparator
class EmployeeComparator implements Comparator<Employee> {
   @Override
   // compare method defines the sorting logic (e.g., by address reverse
order)
   public int compare(Employee emp1, Employee emp2) {
       // int val = emp1.getEno() - emp2.getEno(); // Compare by eno
       // int val = emp1.getEname().compareTo(emp2.getEname()); // Compare
by ename
       // int val = (int)(emp1.getEsal() - emp2.getEsal()); // Compare by
esal (handle float diff carefully)
       int val = emp1.getEaddr().compareTo(emp2.getEaddr()); // Compare by
eaddr
       return -val; // Reverse order
   }
   // Note: equals(Object) method inherited from Object is usually
sufficient for Comparators
}

public class Main {
   public static void main(String[] args) {
      Employee emp1 = new Employee(111,"Durga", 50000, "Hyderabad");
      Employee emp2 = new Employee(222,"Anil", 60000, "Chennai");
      Employee emp3 = new Employee(333,"Ramesh", 70000, "Pune");
      Employee emp4 = new Employee(444,"Venkat", 80000, "Delhi");
      Employee emp5 = new Employee(555,"Ramana", 90000, "Mumbai");

      EmployeeComparator employeeComparator = new EmployeeComparator();
      TreeSet treeSet = new TreeSet(employeeComparator); // Pass Comparator
instance to constructor
      treeSet.add(emp1); // Uses employeeComparator.compare(emp1, existing)
      treeSet.add(emp2); // Uses employeeComparator.compare(emp2, existing)
      treeSet.add(emp3);
      treeSet.add(emp4);
      treeSet.add(emp5);
      System.out.println(treeSet);
   }
}
```

Output Example: (Sorted by address in reverse alphabetical order)

```
[Employee{eno=333, ename='Ramesh', esal=70000.0, eaddr='Pune'},
Employee{eno=555, ename='Ramana', esal=90000.0, eaddr='Mumbai'},
Employee{eno=111, ename='Durga', esal=50000.0, eaddr='Hyderabad'},
Employee{eno=444, ename='Venkat', esal=80000.0, eaddr='Delhi'},
Employee{eno=222, ename='Anil', esal=60000.0, eaddr='Chennai'}]
```

Q) If we provide both implicit Sorting order through Comparable and the explicit Sorting order through the Comparator to the TreeSet then which Sorting logic would be used by the TreeSet?

Ans: If a `TreeSet` is created using a `Comparator` (i.e., `new TreeSet(comparator)`), it will *always* use the `Comparator`'s `compare()` method for sorting and checking equality, even if the elements themselves implement `Comparable`. The explicit `Comparator` overrides the implicit `Comparable`.

Java

```
import java.util.*;
```

```
// Employee implements Comparable (sorts by name)
class Employee implements Comparable<Employee> {
   private int eno; private String ename; private float esal; private
String eaddr;
   public Employee(int eno, String ename, float esal, String eaddr) {
       this.eno = eno; this.ename = ename; this.esal = esal; this.eaddr =
eaddr;
   }
   public int getEno() { return eno; } public String getEname() { return
ename; }
   public float getEsal() { return esal; } public String getEaddr()
{ return eaddr; }
   @Override public String toString() { return "Employee{" + "eno=" + eno +
", ename='" + ename + '\'' + ", esal=" + esal + ", eaddr='" + eaddr + '\''
+ '}'; }

   // Implements Comparable - Natural sort by name (ascending)
   @Override public int compareTo(Employee emp) { int val =
this.ename.compareTo(emp.ename); return val; }
}

// EmployeeComparator implements Comparator (sorts by address ascending)
class EmployeeComparator implements Comparator<Employee> {
   @Override // Explicit sort by address (ascending)
   public int compare(Employee emp1, Employee emp2) { int val =
emp1.getEaddr().compareTo(emp2.getEaddr()); return val; }
}

public class Main {
   public static void main(String[] args) {
      Employee emp1 = new Employee(111,"Durga", 50000, "Hyderabad");
      Employee emp2 = new Employee(222,"Anil", 60000, "Chennai");
      Employee emp3 = new Employee(333,"Ramesh", 70000, "Pune");
      Employee emp4 = new Employee(444,"Venkat", 80000, "Delhi");
      Employee emp5 = new Employee(555,"Ramana", 90000, "Mumbai");

      EmployeeComparator employeeComparator = new EmployeeComparator();
      // Create TreeSet using the Comparator -> explicit sorting takes
priority
      TreeSet treeSet = new TreeSet(employeeComparator);
      treeSet.add(emp1); treeSet.add(emp2); treeSet.add(emp3);
treeSet.add(emp4); treeSet.add(emp5);
      System.out.println(treeSet); // Output will be sorted by Address
(Comparator logic)
   }
}
```

Output Example: (Sorted by Address ascending: Chennai, Delhi, Hyderabad, Mumbai, Pune)

```
[Employee{eno=222, ename='Anil', esal=60000.0, eaddr='Chennai'},
Employee{eno=444, ename='Venkat', esal=80000.0, eaddr='Delhi'},
Employee{eno=111, ename='Durga', esal=50000.0, eaddr='Hyderabad'},
Employee{eno=555, ename='Ramana', esal=90000.0, eaddr='Mumbai'},
Employee{eno=333, ename='Ramesh', esal=70000.0, eaddr='Pune'}]
```

TreeSet Constructors:

1. `public TreeSet()`: Creates an empty TreeSet that uses the elements' natural sorting order (elements must implement `Comparable`).

   Java

   ```
   import java.util.TreeSet;
   // Requires elements added to implement Comparable
   // TreeSet ts = new TreeSet();
   ```

2. `public TreeSet(Comparator c)`: Creates an empty TreeSet that uses the provided `Comparator` for sorting.

   Java

   ```
   import java.util.TreeSet;
   import java.util.Comparator;
   // Requires a Comparator implementation
   // class MyComparator implements Comparator { ... }
   // TreeSet ts = new TreeSet(new MyComparator());
   ```

3. `public TreeSet(Collection c)`: Creates a TreeSet containing all elements from the collection, sorted using their natural order (elements must be `Comparable`).
   - *Reasoning:* Converts other Collection types to a naturally sorted TreeSet.

   Java

   ```
   import java.util.*;

   public class Main {
      public static void main(String[] args) {
         List list = new ArrayList();
         list.add("AAA"); list.add("EEE"); list.add("BBB");
         list.add("FFF"); list.add("CCC"); list.add("DDD");
         System.out.println(list); // Insertion order

         TreeSet treeSet1 = new TreeSet(list); // Converts & sorts by
   natural order (String)
         System.out.println(treeSet1); // Sorted

         Set set = new HashSet();
         set.add(10); set.add(20); set.add(30); set.add(40);
   set.add(50);
         System.out.println(set); // No guaranteed order

         TreeSet treeSet2 = new TreeSet(set); // Converts & sorts by
   natural order (Integer)
         System.out.println(treeSet2); // Sorted

         Queue queue = new PriorityQueue();
         queue.add(1000); queue.add(3000); queue.add(2000);
   queue.add(4000); queue.add(6000); queue.add(5000);
         System.out.println(queue); // Priority order

         TreeSet treeSet3 = new TreeSet(queue); // Converts & sorts by
   natural order (Integer)
         System.out.println(treeSet3); // Sorted
      }
   ```

```
}
```

Output:

```
[AAA, EEE, BBB, FFF, CCC, DDD]
[AAA, BBB, CCC, DDD, EEE, FFF]

[20, 40, 10, 30, 50]
[10, 20, 30, 40, 50]

[1000, 3000, 2000, 4000, 6000, 5000]
[1000, 2000, 3000, 4000, 5000, 6000]
```

4. `public TreeSet(SortedSet ss)`: Creates a TreeSet containing all elements from the SortedSet. Uses the same sorting order as the provided SortedSet.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      SortedSet sortedSet = new TreeSet(); // Create a source
SortedSet (TreeSet)
      sortedSet.add("AAA"); sortedSet.add("BBB");
sortedSet.add("CCC"); sortedSet.add("DDD");
      System.out.println(sortedSet); // [AAA, BBB, CCC, DDD]

      TreeSet treeSet = new TreeSet(sortedSet); // Creates a TreeSet
with the same elements and order
      System.out.println(treeSet); // [AAA, BBB, CCC, DDD]
   }
}
```

   Output:

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
```

TreeSet Exception Examples:

Java

```
import java.util.*;
class A { } // Class A does not implement Comparable

public class Main {
   public static void main(String[] args) {
      TreeSet treeSet = new TreeSet();
      treeSet.add("AAA"); treeSet.add("EEE"); treeSet.add("BBB");
      treeSet.add("DDD"); treeSet.add("CCC");
      System.out.println(treeSet); // [AAA, BBB, CCC, DDD, EEE] (Sorted)

      treeSet.add("BBB"); // Duplicate, ignored
      System.out.println(treeSet); // [AAA, BBB, CCC, DDD, EEE] (No change)

      // treeSet.add(10); // --> ClassCastException (mixing String and
Integer - heterogeneous non-Comparable)
```

```
      // treeSet.add(null); // --> NullPointerException (TreeSet does not
allow nulls)
      // treeSet.add(new A()); // --> ClassCastException (Class A does not
implement Comparable)
      System.out.println(treeSet);
   }
}
```

Output Example: (Output depends on which commented line is uncommented and run)

```
[AAA, BBB, CCC, DDD, EEE]
[AAA, BBB, CCC, DDD, EEE]
[AAA, BBB, CCC, DDD, EEE]
```

Map Interface

- Interface provided in JDK 1.2.
- **Not a child of the `Collection` interface.**
- Stores elements as **Key-Value pairs**. Keys and values are objects.
- **Keys must be unique**. Values can be duplicated.
- Allows heterogeneous elements for both keys and values.
- Allows at most one null key.
- Allows any number of null values.

Methods:

1. `public V put(K key, V value)`: Adds a key-value pair. If the key already exists, replaces the old value and returns the old value; otherwise, returns null.

   Java

   ```
   import java.util.*; // Imports Map and HashMap

   public class Main {
      public static void main(String[] args) {
         Map map = new HashMap(); // HashMap is a common implementation
         map.put(111, "Durga");
         map.put(112, "Anil");
         map.put(113, "Ramesh");
         map.put(114, "Venkat");
         map.put(115, "Rahul");
         System.out.println(map); // Order not guaranteed for HashMap
      }
   }
   ```

   Output Example: (Order varies)

   ```
   {111=Durga, 112=Anil, 113=Ramesh, 114=Venkat, 115=Rahul}
   ```

2. `public void putAll(Map<? extends K, ? extends V> m)`: Copies all key-value pairs from the specified map to this map.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Map map1 = new HashMap();
      map1.put(1,"AAA"); map1.put(2,"BBB"); map1.put(3,"CCC");
map1.put(4,"DDD");
      System.out.println(map1);

      Map map2 = new HashMap();
      map2.putAll(map1); // Copies all from map1 to map2
      System.out.println(map2);
   }
}
```

Output Example: (Order varies)

```
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
```

3. `public V get(Object key)`: Returns the value associated with the key, or null if the key is not found.

   Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Map map = new HashMap();
      map.put(111, "Durga"); map.put(112, "Anil"); map.put(113,
"Ramesh");
      map.put(114, "Venkat"); map.put(115, "Rahul");
      System.out.println(map);

      System.out.println("111 : " + map.get(111)); // Get value for
key 111
      System.out.println("112 : " + map.get(112));
      System.out.println("113 : " + map.get(113));
      System.out.println("114 : " + map.get(114));
      System.out.println("115 : " + map.get(115));
      System.out.println("999 : " + map.get(999)); // Key not found,
returns null
   }
}
```

   Output Example: (Order varies)

```
{111=Durga, 112=Anil, 113=Ramesh, 114=Venkat, 115=Rahul}
111 : Durga
112 : Anil
113 : Ramesh
114 : Venkat
115 : Rahul
999 : null
```

4. `public V remove(Object key)`: Removes the key-value pair for the specified key. Returns the value associated with the key, or null if the key was not found.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Map map = new HashMap();
      map.put(111, "Durga"); map.put(112, "Anil"); map.put(113,
"Ramesh");
      map.put(114, "Venkat"); map.put(115, "Rahul");
      System.out.println(map);

      System.out.println("Removing 112: " + map.remove(112)); //
Remove by key, print removed value
      System.out.println(map);

      System.out.println("Removing 113: " + map.remove(113));
      System.out.println(map);

       System.out.println("Removing 999: " + map.remove(999)); // Key
not found, returns null
   }
}
```

Output Example: (Order varies)

```
{111=Durga, 112=Anil, 113=Ramesh, 114=Venkat, 115=Rahul}
Removing 112: Anil
{111=Durga, 113=Ramesh, 114=Venkat, 115=Rahul}
Removing 113: Ramesh
{111=Durga, 114=Venkat, 115=Rahul}
Removing 999: null
```

5. `public int size()`: Returns the number of key-value pairs.

   Java

   ```
   import java.util.*;

   public class Main {
      public static void main(String[] args) {
         Map map1 = new HashMap();
         map1.put(1,"AAA"); map1.put(2,"BBB"); map1.put(3,"CCC");
   map1.put(4,"DDD");
         System.out.println(map1);
         System.out.println(map1.size()); // 4
      }
   }
   ```

   Output Example:

   ```
   {1=AAA, 2=BBB, 3=CCC, 4=DDD}
   4
   ```

6. `public boolean isEmpty()`: Returns `true` if the map contains no key-value pairs.
7. `public void clear()`: Removes all key-value pairs.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Map map1 = new HashMap();
      map1.put(1,"AAA"); map1.put(2,"BBB"); map1.put(3,"CCC");
map1.put(4,"DDD");
      System.out.println(map1);
      System.out.println(map1.isEmpty()); // false
      map1.clear(); // Remove all pairs
      System.out.println(map1.isEmpty()); // true
   }
}
```

Output Example:

```
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
false
true
```

8. `public boolean containsKey(Object key)`: Returns `true` if the map contains the key.
9. `public boolean containsValue(Object value)`: Returns `true` if the map contains the value.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Map map1 = new HashMap();
      map1.put(1,"AAA"); map1.put(2,"BBB"); map1.put(3,"CCC");
map1.put(4,"DDD");
      System.out.println(map1);

      System.out.println(map1.containsKey(3)); // true
      System.out.println(map1.containsKey(5)); // false
      System.out.println(map1.containsValue("BBB")); // true
      System.out.println(map1.containsValue("ZZZ")); // false
   }
}
```

Output Example:

```
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
true
false
true
false
```

10. `public Collection<V> values()`: Returns a `Collection` view of the values.
11. `public Set<K> keySet()`: Returns a `Set` view of the keys.

Java

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Map map1 = new HashMap();
      map1.put(1,"AAA"); map1.put(2,"BBB"); map1.put(3,"CCC");
map1.put(4,"DDD");
      System.out.println(map1);

      Collection valuesCollection = map1.values(); // Get values as a
Collection
      System.out.println(valuesCollection); // [AAA, BBB, CCC, DDD]
(Order may vary)

      Set keysSet = map1.keySet(); // Get keys as a Set
      System.out.println(keysSet); // [1, 2, 3, 4] (Order may vary)
   }
}
```

Output Example: (Order varies)

```
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
[AAA, BBB, CCC, DDD]
[1, 2, 3, 4]
```

Java Collections: Map and Queue Implementations

This section covers various implementations of the `Map` interface, the `Properties` class, the `Queue` and `Deque` interfaces, and their common implementations.

HashMap

- Provided in JDK 1.2.
- Not a legacy Map.
- Implements `Map` interface.
- Stores data as Key-Value pairs. Keys and values are objects.
- Keys must be unique; values can be duplicated.
- Allows heterogeneous elements for keys and values.
- Allows one null key; allows any number of null values.
- Does not follow insertion order.
- Does not follow sorting order.
- Internal data structure: Hash table (array of buckets, potentially trees).
- Default initial capacity: 16.
- Default load factor: 0.75 (75%).
- Not synchronized.
- Not thread-safe. Generally offers high performance.

Constructors:

1. `public HashMap()`: Creates an empty HashMap with default initial capacity (16) and load factor (0.75).

   Java

   ```
   import java.util.*;
   public class Main {
      public static void main(String[] args) {
         HashMap hashMap = new HashMap();
         System.out.println(hashMap); // {}
      }
   }
   ```

   Output:

   ```
   {}
   ```

2. `public HashMap(int capacity)`: Creates an empty HashMap with the specified capacity and default load factor (0.75).

   Java

   ```
   import java.util.*;
   public class Main {
      public static void main(String[] args) {
         HashMap hashMap = new HashMap(20);
         System.out.println(hashMap); // {}
      }
   }
   ```

   Output:

   ```
   {}
   ```

3. `public HashMap(int capacity, float loadFactor)`: Creates an empty HashMap with the specified capacity and load factor.

   Java

   ```
   import java.util.*;
   public class Main {
      public static void main(String[] args) {
         HashMap hashMap = new HashMap(20, 0.85f);
         System.out.println(hashMap); // {}
      }
   }
   ```

   Output:

   ```
   {}
   ```

4. `public HashMap(Map<? extends K, ? extends V> m)`: Creates a HashMap with all key-value pairs of the provided Map.

Java

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
      HashMap hashMap = new HashMap();
      hashMap.put(1, "AAA"); hashMap.put(2, "BBB"); hashMap.put(3,
"CCC"); hashMap.put(4, "DDD");
      System.out.println(hashMap); // {1=AAA, ...}

      HashMap hashMap1 = new HashMap(hashMap); // Copies elements
      System.out.println(hashMap1); // {1=AAA, ...}
   }
}
```

Output Example: (Order varies)

```
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
```

HashMap Example (General Use):

Java

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
      HashMap hashMap = new HashMap();
      hashMap.put(1, "AAA");
      hashMap.put(2, "BBB");
      hashMap.put(3, "CCC");
      hashMap.put(4, "DDD");
      System.out.println(hashMap); // {1=AAA, 2=BBB, 3=CCC, 4=DDD}
      hashMap.put(2, "EEE"); // Key 2 exists, updates value
      System.out.println(hashMap); // {1=AAA, 2=EEE, 3=CCC, 4=DDD}
      hashMap.put(5, "CCC"); // Key 5 is new, value "CCC" is duplicate
(allowed)
      System.out.println(hashMap); // {1=AAA, 2=EEE, 3=CCC, 4=DDD, 5=CCC}
      hashMap.put(null, null); // Add null key and null value
      hashMap.put(6, null); // Add null value with a non-null key
      System.out.println(hashMap); // {null=null, 1=AAA, 2=EEE, 3=CCC,
4=DDD, 5=CCC, 6=null}
      hashMap.put("A", 222); // Add heterogeneous types
      hashMap.put(22.222f, 33.3333d);
      System.out.println(hashMap); // {null=null, 1=AAA, A=222, 2=EEE,
3=CCC, 4=DDD, 5=CCC, 6=null, 22.222=33.3333}
   }
}
```

Output Example: (Order varies)

```
{1=AAA, 2=BBB, 3=CCC, 4=DDD}
{1=AAA, 2=EEE, 3=CCC, 4=DDD}
{1=AAA, 2=EEE, 3=CCC, 4=DDD, 5=CCC}
{null=null, 1=AAA, 2=EEE, 3=CCC, 4=DDD, 5=CCC, 6=null}
{null=null, 1=AAA, A=222, 2=EEE, 3=CCC, 4=DDD, 5=CCC, 6=null,
22.222=33.3333}
```

LinkedHashMap

- Provided in JDK 1.4.
- Implements `Map`.
- Maintains insertion order (or access order).
- Internal data structure: Hash table and Doubly-linked list.

Q) What are the differences between HashMap and LinkedHashMap?

Ans:

Feature	HashMap	LinkedHashMap	*Reasoning/Notes*
Introduced in	JDK 1.2	JDK 1.4	LinkedHashMap is newer.
Insertion Order	Does not follow insertion order	**Follows insertion order**	LinkedHashMap uses a linked list for iteration order.
Internal DS	HashMap (array of buckets)	HashMap + Doubly-linked list	LinkedHashMap adds linked list overhead for order.

Export to Sheets

Java

```
import java.util.*;
public class Main {
   public static void main(String[] args) {
      HashMap hashMap = new HashMap(); // Order not guaranteed
      hashMap.put(111, "AAA"); hashMap.put(222, "BBB"); hashMap.put(333,
"CCC"); hashMap.put(444, "DDD");
      System.out.println(hashMap);

      LinkedHashMap linkedHashMap = new LinkedHashMap(); // Maintains
insertion order
      linkedHashMap.put(111, "AAA"); linkedHashMap.put(222, "BBB");
linkedHashMap.put(333, "CCC"); linkedHashMap.put(444, "DDD");
      System.out.println(linkedHashMap);
   }
}
```

Output Example: (HashMap order may vary, LinkedHashMap order matches insertion)

```
{444=DDD, 333=CCC, 222=BBB, 111=AAA} // HashMap order might vary
{111=AAA, 222=BBB, 333=CCC, 444=DDD} // LinkedHashMap order
```

IdentityHashMap

- Provided in JDK 1.4.
- Implements `Map`.
- Uses reference equality (`==`) instead of object equality (`equals()`) to compare keys.

`==` vs `equals()`:

- `==` operator: Compares primitive values directly or object references (checks if two references point to the exact same object in memory).

- `equals()` method (from `Object`): Default implementation also checks object references. Many classes (like `String`, Wrapper classes) override `equals()` to compare object *contents* instead of references.

Java
```
class A{ } // Custom class, inherits Object's == and equals()

public class Main {
   public static void main(String[] args) {
      int a = 10; int b = 20; // Primitives
      A a1 = new A(); A a2 = new A(); // Different objects, different
references
      String str1 = new String("abc"); String str2 = new String("abc"); //
Different objects, different references, same content

      System.out.println(a == b)); // false (values differ)
      System.out.println(a1 == a2)); // false (references differ)
      System.out.println(str1 == str2)); // false (references differ)
      System.out.println();

      System.out.println(a1.equals(a2)); // false (references differ,
default equals)
      System.out.println(str1.equals(str2)); // true (contents are equal,
overridden equals)
   }
}
```

Output:

```
false
false
false

false
true
```

Q) What is the difference between HashMap and IdentityHashMap?

Ans:

Feature	**HashMap**	**IdentityHashMap**	***Reasoning/Notes***
Introduced in	JDK 1.2	JDK 1.4	IdentityHashMap is newer.
Key Comparison	Uses `equals()` method	Uses == operator	Defines how duplicate keys are identified.

Export to Sheets

Java
```
import java.util.HashMap;
import java.util.IdentityHashMap;

public class Main {
   public static void main(String[] args) {
      Integer in1 = new Integer(10); // Creates object 1
      Integer in2 = new Integer(10); // Creates object 2 (different
reference, same value)

      HashMap hashMap = new HashMap();
```

```
        hashMap.put(in1, "abc"); // Key is 10 (value)
        hashMap.put(in2, "def"); // in1.equals(in2) is true, so key 10 is
considered duplicate, value is updated.
        System.out.println(hashMap); // {10=def}

        IdentityHashMap identityHashMap = new IdentityHashMap();
        identityHashMap.put(in1, "abc"); // Key is reference to in1
        identityHashMap.put(in2, "def"); // in1 == in2 is false, keys are
considered different.
        System.out.println(identityHashMap); // {10=abc, 10=def} (shows value
10 twice but uses different keys internally)
    }
}
```

Output:

```
{10=def}
{10=abc, 10=def} // Note: The output string for IdentityHashMap might not
clearly show the distinct keys, but they are treated as separate entries.
```

WeakHashMap

- Implements `Map`.
- Keys are held as **weak references**. This means a key-value pair can be garbage collected if the key is no longer strongly referenced elsewhere in the application.

Garbage Collection (GC) Interaction: GC destroys objects that are no longer reachable by any strong references. You can suggest GC using `System.gc()`, but execution is not guaranteed. The `finalize()` method is called before an object is garbage collected.

Java

```
class A {
    A() { System.out.println("Object Creating...."); }
    @Override protected void finalize() throws Throwable
{ System.out.println("Object Destroying...."); }
    @Override public String toString() { return "AAA"; } // For readable
output
}

public class Main {
    public static void main(String[] args) {
        A a = new A(); // Object A created
        a = null; // Nullify strong reference to A
        System.gc(); // Suggest Garbage Collection
        // Output: Object Creating.... Object Destroying.... (finalize is
called)
    }
}
```

Output Example:

```
Object Creating....
Object Destroying....
```

Q) What is the difference between HashMap and WeakHashMap?

Ans:

Feature	HashMap	WeakHashMap	*Reasoning/Notes*
Key Referencing	Keys are strongly referenced	Keys are weakly referenced	Strong references prevent GC; weak references do not.
GC Interaction	HashMap protects keys from GC	GC can collect keys (and entries)	WeakHashMap allows GC to reclaim memory if keys are only weakly reachable.
Persistence	Entries persist as long as HashMap exists	Entries removed when keys collected	Useful for caches where entries should expire.

Export to Sheets

Java

```
import java.util.HashMap;
import java.util.WeakHashMap;

class A {
   A() { System.out.println("Object Creating...."); }
   @Override protected void finalize() throws Throwable
{ System.out.println("Object Destroying...."); }
   @Override public String toString() { return "AAA"; }
}

public class Main {
   public static void main(String[] args) throws InterruptedException { //
Added InterruptedException for sleep
      A a1 = new A(); // Object A1 created
      HashMap hashMap = new HashMap();
      hashMap.put(a1, "Durga"); // HashMap holds a strong reference to a1

      System.out.println("HashMap before GC    : " + hashMap); //
{AAA=Durga}
      a1 = null; // Nullify the 'a1' reference
      System.gc(); // Suggest GC
      Thread.sleep(100); // Give GC a moment (not guaranteed)
      System.out.println("HashMap after GC     : " + hashMap); //
{AAA=Durga} (a1 not collected because HashMap holds a strong ref)

      System.out.println(); // Separator

      A a2 = new A(); // Object A2 created
      WeakHashMap weakHashMap = new WeakHashMap();
      weakHashMap.put(a2, "Durga"); // WeakHashMap holds a weak reference
to a2

      System.out.println("WeakHashMap before GC  : " + weakHashMap); //
{AAA=Durga}
      a2 = null; // Nullify the 'a2' reference (now only WeakHashMap has a
weak ref)
      System.gc(); // Suggest GC
      Thread.sleep(100); // Give GC a moment
      System.out.println("WeakHashMap after GC   : " + weakHashMap); // {}
(a2 collected, entry removed)
   }
}
```

Output Example: (Order of Creating/Destroying might vary)

```
Object Creating....
HashMap before GC    : {AAA=Durga}
HashMap after GC     : {AAA=Durga}

Object Creating....
WeakHashMap before GC  : {AAA=Durga}
Object Destroying....
WeakHashMap after GC   : {}
```

Hashtable

- Provided in JDK 1.0.
- **Legacy** Map implementation.
- Implements `Map`, `Serializable`, `Cloneable`.
- **Synchronized**.
- **Thread-safe**.

Q) What are the differences between HashMap and Hashtable?

Ans:

Feature	**HashMap**	**Hashtable**	***Reasoning/Notes***
Introduced in	JDK 1.2	JDK 1.0	Hashtable is older, part of original Java.
Legacy Status	No	Yes	
Nulls	One null key, any null values	No null keys, No null values (`NullPointerException`)	Important difference in usage.
Synchronization	Not synchronized	**Synchronized**	Hashtable is thread-safe by default.
Thread Safety	Not thread-safe	**Thread-safe**	Achieved via synchronized methods.
Performance	Faster	Slower	Synchronization adds overhead.
Data Consistency	Not guaranteed (multi-thread)	Guaranteed (multi-thread)	Synchronization ensures only one thread modifies at a time.
Thread Access	Parallel execution	Sequential execution	Due to synchronization.

Export to Sheets

Java

```
import java.util.HashMap;
import java.util.Hashtable;
// import java.util.Comparator; // Not used in this example
// import java.util.TreeMap; // Not used in this example

public class Main {
   public static void main(String[] args) {
      HashMap hashMap = new HashMap();
      hashMap.put("A", 1);
      hashMap.put("B", 2);
      hashMap.put(null, 3); // null key allowed
      hashMap.put("C", null); // null value allowed
```

```
      hashMap.put("D", null); // multiple null values allowed
      System.out.println(hashMap);

      Hashtable hashtable = new Hashtable();
      hashtable.put("A", 1);
      hashtable.put("B", 2);
      // hashtable.put("C", null); // --> NullPointerException (null value
not allowed)
      // hashtable.put(null, "D"); // --> NullPointerException (null key
not allowed)
      System.out.println(hashtable); // {A=1, B=2}
   }
}
```

Output Example: (Order varies for HashMap/Hashtable)

```
{null=3, A=1, B=2, C=null, D=null}
{B=2, A=1} // Hashtable order might vary
```

SortedMap Interface

- Interface provided in JDK 1.2.
- Child interface of `Map`.
- Stores Key-Value pairs, ordered by the **keys**. Keys must be Comparable or use a Comparator.
- Keys must be unique; values can be duplicated.
- Keys must be homogeneous; values can be heterogeneous.
- Does not allow null keys (`NullPointerException`).

Methods:

1. `public SortedMap<K, V> headMap(K toKey)`: Returns a view of the portion whose keys are strictly less than `toKey`.
2. `public SortedMap<K, V> tailMap(K fromKey)`: Returns a view of the portion whose keys are greater than or equal to `fromKey`.
3. `public SortedMap<K, V> subMap(K fromKey, K toKey)`: Returns a view of the portion whose keys range from `fromKey` (inclusive) to `toKey` (exclusive).
4. `public K firstKey()`: Returns the first (lowest) key.
5. `public K lastKey()`: Returns the last (highest) key.

Java

```
import java.util.SortedMap;
import java.util.TreeMap; // TreeMap is a common implementation

public class Main {
   public static void main(String[] args) {
      SortedMap sortedMap = new TreeMap(); // Uses TreeMap to demonstrate
SortedMap methods
      sortedMap.put("F", "FFF"); sortedMap.put("A", "AAA");
sortedMap.put("E", "EEE");
      sortedMap.put("B", "BBB"); sortedMap.put("D", "DDD");
sortedMap.put("C", "CCC");
      System.out.println(sortedMap); // Keys are sorted
```

```
      System.out.println(sortedMap.firstKey()); // A
      System.out.println(sortedMap.lastKey()); // F
      System.out.println(sortedMap.headMap("D")); // {A=AAA, B=BBB, C=CCC}
(Keys < "D")
      System.out.println(sortedMap.tailMap("D")); // {D=DDD, E=EEE, F=FFF}
(Keys >= "D")
      System.out.println(sortedMap.subMap("B", "E")); // {B=BBB, C=CCC,
D=DDD} (Keys >= "B" and < "E")
   }
}
```

Output:

```
{A=AAA, B=BBB, C=CCC, D=DDD, E=EEE, F=FFF}
A
F
{A=AAA, B=BBB, C=CCC}
{D=DDD, E=EEE, F=FFF}
{B=BBB, C=CCC, D=DDD}
```

NavigableMap Interface

- Interface provided in JDK 1.6.
- Child interface to `SortedMap`.
- Provides navigation methods for finding keys based on their relationship to a given key.

Methods:

1. `public NavigableMap<K, V> descendingMap()`: Returns a reverse order view of the map.
2. `public K ceilingKey(K key)`: Returns the least key greater than or equal to `key`.
3. `public K higherKey(K key)`: Returns the least key strictly greater than `key`.
4. `public K floorKey(K key)`: Returns the greatest key less than or equal to `key`.
5. `public K lowerKey(K key)`: Returns the greatest key strictly less than `key`.
6. `public Map.Entry<K, V> pollFirstEntry()`: Retrieves and removes the entry with the lowest key.
7. `public Map.Entry<K, V> pollLastEntry()`: Retrieves and removes the entry with the highest key.

Java

```
import java.util.NavigableMap;
import java.util.SortedMap;
import java.util.TreeMap; // TreeMap implements NavigableMap

public class Main {
   public static void main(String[] args) {
      NavigableMap navigableMap = new TreeMap(); // Uses TreeMap to
demonstrate NavigableMap methods
      navigableMap.put("A", "AAA"); navigableMap.put("D", "DDD");
navigableMap.put("B", "BBB");
      navigableMap.put("E", "EEE"); navigableMap.put("C", "CCC");
navigableMap.put("F", "FFF");
      System.out.println(navigableMap); // {A=AAA, B=BBB, C=CCC, D=DDD,
E=EEE, F=FFF}
```

```
      System.out.println(navigableMap.descendingMap()); // {F=FFF, E=EEE,
D=DDD, C=CCC, B=BBB, A=AAA} (Reverse order view)
      System.out.println(navigableMap.ceilingKey("D")); // D (>= D)
      System.out.println(navigableMap.higherKey("D")); // E (> D)
      System.out.println(navigableMap.floorKey("D")); // D (<= D)
      System.out.println(navigableMap.lowerKey("D")); // C (< D)

      System.out.println(navigableMap.pollFirstEntry()); // A=AAA (Removed
entry)
      System.out.println(navigableMap.pollLastEntry()); // F=FFF (Removed
entry)
      System.out.println(navigableMap); // {B=BBB, C=CCC, D=DDD, E=EEE}
(Remaining entries)
   }
}
```

Output:

```
{A=AAA, B=BBB, C=CCC, D=DDD, E=EEE, F=FFF}
{F=FFF, E=EEE, D=DDD, C=CCC, B=BBB, A=AAA}
D
E
D
C
A=AAA
F=FFF
{B=BBB, C=CCC, D=DDD, E=EEE}
```

TreeMap

- Class provided in JDK 1.2.
- Not a legacy Map.
- Implements `NavigableMap`, `SortedMap`, and `Map`.
- Stores Key-Value pairs in **key sorting order**.
- Does not allow duplicate keys.
- Does not follow insertion order.
- Keys must be **Homogeneous** and **Comparable** or use a **Comparator**.
- Does not allow null keys (`NullPointerException`).
- Not synchronized, not thread-safe.

Java

```
import java.util.TreeMap;
import java.util.Comparator; // Needed for custom sorting example

class A{} // Class A does not implement Comparable

public class Main {
   public static void main(String[] args) {
      TreeMap treeMap = new TreeMap(); // Uses natural sorting of keys
(keys must be Comparable)
      treeMap.put("Nagoor", 123456);
      treeMap.put("Durga", 4567890);
      treeMap.put("Sarah", 987654321);
      treeMap.put("Alex", 987654321); // Alex vs Sarah, Alex comes first,
new entry
      treeMap.put("Rakesh", 83735292);
```

```
        System.out.println(treeMap); // Keys are sorted alphabetically

        treeMap.put("Nagoor", 972383); // Key "Nagoor" exists, value is
updated
        System.out.println(treeMap); // {Alex=987654321, Durga=4567890,
Nagoor=972383, Rakesh=83735292, Sarah=987654321}

        // treeMap.put(null, 987654321); // --> NullPointerException (TreeMap
does not allow null keys)
        // treeMap.put(111, 987654321); // --> ClassCastException (mixing
String keys with Integer keys)
        // A a = new A();
        // TreeMap treeMap1 = new TreeMap();
        // treeMap1.put(a, 972383); // --> ClassCastException (Class A is not
Comparable)
    }
}
```

Output Example: (Keys are sorted)

```
{Alex=987654321, Durga=4567890, Nagoor=123456, Rakesh=83735292,
Sarah=987654321}
{Alex=987654321, Durga=4567890, Nagoor=972383, Rakesh=83735292,
Sarah=987654321}
```

TreeMap with Comparator:

Java

```
import java.util.Comparator;
import java.util.TreeMap;

// Custom Comparator for String keys (reverse alphabetical order)
class MyComparator implements Comparator<String> {
    @Override
    public int compare(String str1, String str2) {
        return -str1.compareTo(str2); // '-' reverses natural String order
    }
}

public class Main {
    public static void main(String[] args) {
        // Create TreeMap with the custom Comparator
        TreeMap treeMap = new TreeMap(new MyComparator());
        treeMap.put("****", "Bahubali");
        treeMap.put("*", "Shakthi");
        treeMap.put("***", "Pushpa");
        treeMap.put("*****", "Avathar");
        treeMap.put("**", "Sahu");
        System.out.println(treeMap); // Keys sorted by MyComparator logic
    }
}
```

Output Example: (Keys sorted by reverse alphabetical order)

```
{*****=Avathar, ****=Bahubali, ***=Pushpa, **=Sahu, *=Shakthi}
```

Properties

- Class in `java.util`. Extends `Hashtable`.
- Used to manage key-value pairs, specifically for reading from and writing to `.properties` files. Keys and values are typically `String`.
- Thread-safe (inherited from `Hashtable`).

Common Use Cases:

- Managing frequently changing data (configurations).
- Storing database connection parameters (JDBC).
- Managing labels for GUI components.
- Localization/Internationalization messages.
- Exception or validation messages.

Procedure to Store Data to a Properties File:

1. Create `FileOutputStream` for the target `.properties` file.
2. Create `java.util.Properties` object.
3. Set key-value pairs using `setProperty(String key, String value).`
4. Store data to the file using `store(OutputStream out, String comments).`
5. Close the stream.

Java

```
import java.io.FileOutputStream;
import java.util.Properties;

public class Main {
   public static void main(String[] args) throws Exception { // throws
Exception for simplicity
      FileOutputStream fos = new
FileOutputStream("/Users/nagoorn/Documents/docs/db.properties");
      Properties prop = new Properties(); // Create Properties object

      prop.setProperty("driverClassName", "com.mysql.cj.jdbc.Driver"); //
Set properties
      prop.setProperty("url", "jdbc:mysql://localhost:3306/durgadb");
      prop.setProperty("user", "root");
      prop.setProperty("password", "root");

      prop.store(fos, "Jdbc Parameters"); // Store to file with comments
      System.out.println("Jdbc parameters Sent to
/Users/nagoorn/Documents/docs/db.properties");
      fos.close();
   }
}
```

Output:

```
Jdbc parameters Sent to /Users/nagoorn/Documents/docs/db.properties
```

Procedure to Load Data from a Properties File:

1. Create `FileInputStream` for the `.properties` file.
2. Create `java.util.Properties` object.
3. Load data from the file using `load(InputStream in).`

4. Get property values using `getProperty(String key)`.
5. Close the stream.

Java

```
import java.io.FileInputStream;
import java.util.Properties;

public class Main {
   public static void main(String[] args) throws Exception { // throws
Exception for simplicity
      FileInputStream fis = new
FileInputStream("/Users/nagoorn/Documents/docs/db.properties");
      Properties prop = new Properties(); // Create Properties object

      prop.load(fis); // Load from file

      System.out.println("JDBC Parameters Details");
      System.out.println("--------------------------------");
      System.out.println("Driver Class Name : " +
prop.getProperty("driverClassName"));
      System.out.println("Driver URL        : " + prop.getProperty("url"));
      System.out.println("Database User Name: " +
prop.getProperty("user"));
      System.out.println("Database Password : " +
prop.getProperty("password"));

      fis.close();
   }
}
```

Output:

```
JDBC Parameters Details
--------------------------------
Driver Class Name : com.mysql.cj.jdbc.Driver
Driver URL        : jdbc:mysql://localhost:3306/durgadb
Database User Name: root
Database Password : root
```

Queue Interface

- Interface provided in JDK 1.5.
- Child interface to `Collection`.
- Follows **FIFO** (First-In, First-Out) principle for processing elements.
- Allows duplicate elements.
- Does not follow insertion order (for processing).
- Does not follow sorting order (for processing).
- Allows homogeneous elements (can be restricted with Generics).
- May or may not allow null elements depending on implementation (`PriorityQueue` and `ArrayDeque` do not).

Methods (designed for FIFO):

Operation	For Success	For Failure (Empty Queue)	Description
Add	`boolean`	`IllegalStateException` (if	Adds element; returns

	`offer(E e)`	capacity limited)	true/false. Preferred over `add()`.
Remove	`E poll()`	`null`	Removes and returns head; returns null if empty.
	`E remove()`	`NoSuchElementException`	Removes and returns head; throws exception if empty.
Examine	`E peek()`	`null`	Returns head without removing; returns null if empty.
	`E element()`	`NoSuchElementException`	Returns head without removing; throws exception if empty.

Export to Sheets

Java

```
import java.util.PriorityQueue; // PriorityQueue implements Queue
import java.util.Queue;

public class Main {
   public static void main(String[] args) {
      Queue queue = new PriorityQueue(); // Elements sorted by natural
order/Comparator
      queue.offer("AAA"); // Add elements
      queue.offer("BBB");
      queue.offer("CCC");
      queue.offer("DDD");
      queue.offer("EEE");
      System.out.println(queue); // [AAA, DDD, BBB, EEE, CCC] (Order
reflects priority heap, not insertion)

      System.out.println(queue.poll()); // AAA (Removes and returns the
head/highest priority)
      System.out.println(queue); // [BBB, DDD, CCC, EEE] (AAA removed, heap
reordered)

      System.out.println(queue.remove()); // BBB (Removes and returns the
head)
      System.out.println(queue); // [CCC, DDD, EEE] (BBB removed)

      Queue queue1 = new PriorityQueue();
      System.out.println(queue1.poll()); // null (poll on empty queue)
      // System.out.println(queue1.remove()); // --> NoSuchElementException
(remove on empty queue)

      System.out.println(queue.peek()); // CCC (Examines head)
      System.out.println(queue.element()); // CCC (Examines head)

      System.out.println(queue1.peek()); // null (peek on empty queue)
      // System.out.println(queue1.element()); // -->
NoSuchElementException (element on empty queue)
   }
}
```

Output Example: (Order might vary for PriorityQueue printout but poll/remove will get the 'head')

```
[AAA, DDD, BBB, EEE, CCC]
```

```
AAA
[BBB, DDD, CCC, EEE]
BBB
[CCC, DDD, EEE]
null
CCC
CCC
null
```

PriorityQueue

- Class provided in JDK 1.5.
- Not a legacy Collection.
- Implements `Queue`.
- Elements are ordered based on their **natural sorting order** or by a `Comparator` provided at construction.
- Retrieval operations (`poll`, `remove`, `peek`, `element`) access the head element, which is the element with the highest priority (lowest value according to the sorting order).
- Not index-based.
- Allows duplicate elements.
- Does not follow insertion order.
- Follows priority-based ordering (not standard sorting order like `TreeSet`).
- Allows only **Homogeneous** elements.
- Does not allow null elements (`NullPointerException`).
- Default initial capacity: 11.
- Not synchronized, not thread-safe.

Constructors:

1. `public PriorityQueue()`: Creates a PriorityQueue using the elements' natural sorting order. Elements must be Comparable.
2. `public PriorityQueue(Comparator<? super E> comparator)`: Creates a PriorityQueue using the specified Comparator.
3. `public PriorityQueue(int initialCapacity)`: Creates a PriorityQueue with the specified initial capacity, using natural sorting.
4. `public PriorityQueue(PriorityQueue<? extends E> c)`: Creates a PriorityQueue with elements from another PriorityQueue.
5. `public PriorityQueue(SortedSet<? extends E> c)`: Creates a PriorityQueue with elements from a SortedSet.
6. `public PriorityQueue(Collection<? extends E> c)`: Creates a PriorityQueue with elements from a Collection. Elements must be Comparable or the queue must use a Comparator.

Java

```
import java.util.PriorityQueue;
import java.util.Queue;
// import java.util.Comparator; // For Comparator constructor example

public class Main {
   public static void main(String[] args) {
      PriorityQueue pq = new PriorityQueue(); // Uses natural sorting
(String)
```

```
      pq.add("AAA"); pq.add("BBB"); pq.add("CCC"); pq.add("DDD");
pq.add("EEE"); pq.add("FFF");
      System.out.println(pq); // [AAA, DDD, BBB, EEE, CCC, FFF] (Order in
printout is internal heap order)

      pq.add("BBB"); // Add duplicate (allowed)
      System.out.println(pq); // [AAA, DDD, BBB, EEE, CCC, FFF, BBB]

      // pq.add(null); // --> NullPointerException (does not allow nulls)
      // pq.add(10); // --> ClassCastException (does not allow
heterogeneous unless using Comparator with a common supertype)
      System.out.println(pq); // [AAA, DDD, BBB, EEE, CCC, FFF, BBB]
   }
}
```

Output Example: (Order might vary)

```
[AAA, DDD, BBB, EEE, CCC, FFF]
[AAA, DDD, BBB, EEE, CCC, FFF, BBB]
[AAA, DDD, BBB, EEE, CCC, FFF, BBB]
```

Deque Interface

- Interface provided in JDK 1.6.
- Stands for **Double Ended Queue**.
- Child interface to `Queue`.
- Allows element insertion, removal, and examination at **both ends**.

Methods (common ones):

Operation	First End	Last End	Description
Insert	`void addFirst(E e)`	`void addLast(E e)`	Adds element; throws exception if capacity limited.
	`boolean offerFirst(E e)`	`boolean offerLast(E e)`	Adds element; returns true/false.
Remove	`E removeFirst()`	`E removeLast()`	Removes and returns element; throws exception if empty.
	`E pollFirst()`	`E pollLast()`	Removes and returns element; returns null if empty.
Examine	`E getFirst()`	`E getLast()`	Returns element without removing; throws exception if empty.
	`E peekFirst()`	`E peekLast()`	Returns element without removing; returns null if empty.

Export to Sheets

Java

```
import java.util.ArrayDeque; // ArrayDeque is a common implementation
import java.util.Deque;
// import java.util.PriorityQueue; // Not used in this example
// import java.util.Queue; // Not used in this example

public class Main {
   public static void main(String[] args) {
      Deque deque = new ArrayDeque(); // Uses ArrayDeque to demonstrate
Deque methods
```

```
        deque.add("AAA");
        deque.add("BBB");
        deque.add("CCC");
        deque.add("DDD");
        System.out.println(deque); // [AAA, BBB, CCC, DDD]

        deque.addFirst("XXX"); // Add to front
        deque.addLast("YYY"); // Add to end
        System.out.println(deque); // [XXX, AAA, BBB, CCC, DDD, YYY]

        System.out.println(deque.getFirst()); // XXX (Examine front)
        System.out.println(deque.getLast()); // YYY (Examine end)

        System.out.println(deque.removeFirst()); // XXX (Remove from front)
        System.out.println(deque.removeLast()); // YYY (Remove from end)
        System.out.println(deque); // [AAA, BBB, CCC, DDD] (Remaining
elements)
    }
}
```

Output:

```
[AAA, BBB, CCC, DDD]
[XXX, AAA, BBB, CCC, DDD, YYY]
XXX
YYY
XXX
YYY
[AAA, BBB, CCC, DDD]
```

ArrayDeque

- Class provided in JDK 1.6.
- Not a legacy Collection.
- Implements `Deque`.
- Internal data structure: Resizable Array.
- Not index-based.
- Allows duplicate elements.
- **Follows insertion order** for iteration.
- Allows heterogeneous elements (can be restricted with Generics).
- **Does not allow null elements** (`NullPointerException`).
- Default initial capacity: 16.
- Not synchronized, not thread-safe.

Constructors:

1. `public ArrayDeque()`: Creates an empty ArrayDeque with default initial capacity (16).
2. `public ArrayDeque(int capacity)`: Creates an empty ArrayDeque with the specified initial capacity.
3. `public ArrayDeque(Collection<? extends E> c)`: Creates an ArrayDeque with all elements from the provided Collection.

Java

```
import java.util.ArrayDeque;
import java.util.Deque;
// import java.util.PriorityQueue; // Not used in this example
// import java.util.Queue; // Not used in this example

public class Main {
   public static void main(String[] args) {
      ArrayDeque deque = new ArrayDeque(); // Uses ArrayDeque
      deque.add("AAA"); deque.add("FFF"); deque.add("BBB");
      deque.add("EEE"); deque.add("CCC"); deque.add("DDD");
      System.out.println(deque); // [AAA, FFF, BBB, EEE, CCC, DDD]
(Maintains insertion order)

      deque.add("BBB"); // Add duplicate
      System.out.println(deque); // [AAA, FFF, BBB, EEE, CCC, DDD, BBB]

      // deque.add(null); // --> NullPointerException (does not allow
nulls)

      System.out.println(deque); // [AAA, FFF, BBB, EEE, CCC, DDD, BBB]
      deque.add(10); // Add heterogeneous
      System.out.println(deque); // [AAA, FFF, BBB, EEE, CCC, DDD, BBB, 10]
   }
}
```

Output:

```
[AAA, FFF, BBB, EEE, CCC, DDD]
[AAA, FFF, BBB, EEE, CCC, DDD, BBB]
[AAA, FFF, BBB, EEE, CCC, DDD, BBB]
[AAA, FFF, BBB, EEE, CCC, DDD, BBB, 10]
```