Zoho Schools for Graduate Studies

Notes

DAY – 10 (30-07-2025)

Session -1

Rules for Exception Handling with Method Overriding in Java

1. What is Method Overriding and Exception Handling?

- Method Overriding: When a subclass provides a specific implementation of a method that is already defined in its superclass.
- Exception Handling: Mechanism to handle runtime errors using try-catch blocks.
- In Java, when a method is overridden, the overriding method can handle exceptions, but it must follow certain rules, especially for checked exceptions.

2. Rule 1: Superclass method does NOT declare exception

- If the superclass method does not declare any exception, then the overriding method in the subclass cannot declare any checked exception.
- Why? Because that would force callers to handle exceptions that were not originally present in the superclass method.

Example :

```
class Parent {
  void show()
       {
              System.out.println("Parent: No exception");
       }
    }
  class Child extends Parent
       {
              // Compile-time Error: Cannot throw checked exception
              void show() throws Exception {
              System.out.println("Child: Exception? Not allowed here!");
       }
 }
```

3. Rule 2: Superclass method declares a checked exception :

- If the superclass method declares a checked exception, then the overriding method can declare: - The same exception - A subclass of that exception (narrower exception)
- - No exception at all But it CANNOT throw a broader or new checked exception.

Example :

```
class Parent {
    void show() throws IOException
            {
                    System.out.println("Parent: Throws IOException");
            }
      }
 class Child extends Parent
            {
                    void show() throws FileNotFoundException
                            {
System.out.println("Child: Throws subclass exception – Allowed");
                            }
             }
```

4. Rule 3: Superclass method declares an unchecked exception

- If the superclass method declares an unchecked exception (like Runtime Exception), then the overriding method can throw: -
- **Same exception**
- **Subclass of it**
- **No exception**
- **Even new unchecked exceptions**

- There are no restrictions on unchecked exceptions.

- **Example :**

```
 class Parent
     {
    void show() throws ArithmeticException
            {
                    System.out.println("Parent: Runtime exception");
            }
     }
class Child extends Parent
     {
         void show() throws NullPointerException
            {
            System.out.println("Child: Also Runtime exception - Allowed");
            }
     }
```

5. Real-Time Example Scenario :

- Imagine a Bank system.
- Superclass: Bank
- Method: processLoan() - May throw LoanProcessingException
- Subclass: SBI or HDFC - Can override processLoan() and handle a specific type of Loan exception

Example :

```
class Bank
    {
   void processLoan() throws Exception
         {
                System.out.println("Bank: Processing loan...");
         }
     }
class SBI extends Bank
    {
       void processLoan() throws FileNotFoundException
         {
         System.out.println("SBI: Document not found during loan process!");
         }
     }
```

6. Interview Points

- - You CANNOT throw new or broader checked exceptions in the child class
- You CAN throw narrower (subclass) exceptions.
- Unchecked exceptions are NOT restricted in overriding.
- These rules are enforced at compile time.

7. Summary :

- Always be careful with checked exceptions when overriding methods. Breaking the rules will lead to compile-time errors

Session - 2

Collections in Java

- A collection in Java is a group of individual objects that are treated as a single unit. In Java, a separate framework named the "**Collection Framework"** was defined in JDK 1.2, which contains all the Java Collection Classes and interfaces.

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

Needed for a Collection Framework :

- Before the Collection Framework (before JDK 1.2), Java used Arrays, Vectors, and Hashtables to group objects, but they lacked a common interface. Each had a separate implementation, making usage inconsistent and harder for developers to learn and maintain.

- Let's understand this with an example of adding an element to a hashtable and a vector.

Example:

```
import java.io.*;

import java.util.*;


class CollectionDemo {

    public static void main(String[] args)

    {

        // Creating instances of the array, vector and hashtable

        int arr[] = new int[] { 1, 2, 3, 4 };

        Vector<Integer> v = new Vector();

        Hashtable<Integer, String> h = new Hashtable();


        // Adding the elements into the vector

        v.addElement(1);

        v.addElement(2);


        // Adding the element into the hashtable

        h.put(1, "geeks");
        h.put(2, "4geeks");

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

    }
}
```

Output

```
1
1
geeks
```

Advantages of the Java Collection Framework :

Since the lack of a collection framework gave rise to the above set of disadvantages, the following are the advantages of the collection framework.

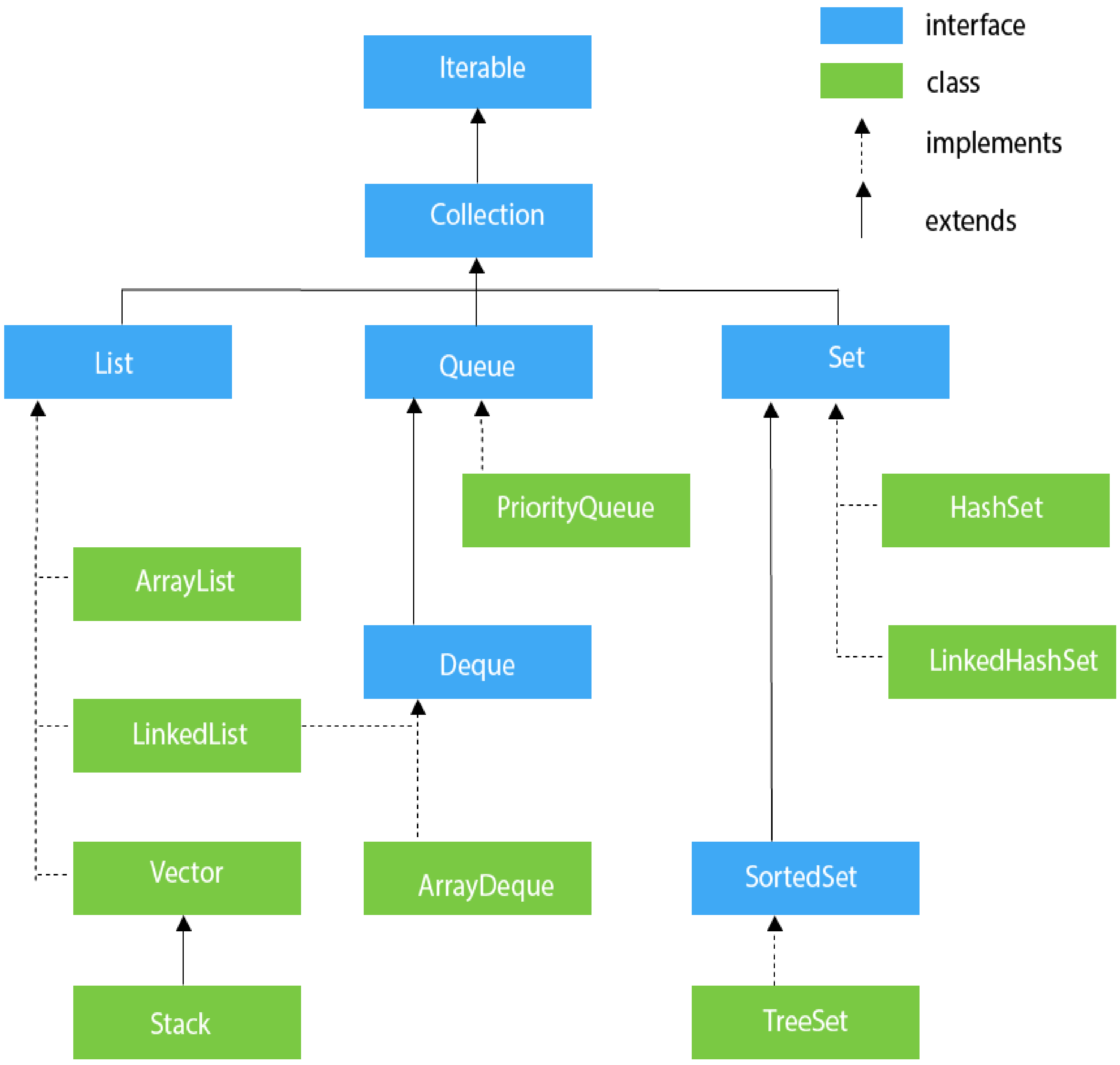
1.**Consistent API:** Interfaces like List, Set, and Map have common methods across classes (ArrayList, LinkedList, etc.)**.**

2.**Less Coding Effort:** Developers focus on usage, not designing data structures—supports OOP abstraction.

3.**Better Performance:** Offers fast, reliable implementations of data structures, improving speed and quality of code.

Hierarchy of the Collection Framework in Java :

The Collection interface extends Iterable and serves as the root, defining common methods inherited by all collection classes.



Interfaces that Extend the Java Collections Interface :

The collection framework contains multiple interfaces where every interface is used to store a specific type of data. The following are the interfaces present in the framework.

1. Iterable Interface

Iterable interface is the root of the Collection Framework. It is extended by the Collection interface, making all collections inherently iterable. Its primary purpose is to provide an Iterator to traverse elements, defined by its single abstract method iterator().

```
Iterator iterator();
```

2. Collection Interface :

Collection interface extends Iterable and serves as the foundation of the Collection Framework. It defines common methods like add(), remove(), and clear(), ensuring consistency and reusability across all collection implementations.

3. List Interface :

List interface extends the Collection interface and represents an ordered collection that allows duplicate elements. It is implemented by classes like ArrayList, Vector, and Stack. Since all these classes implement List, a list object can be instantiated using any of them

For example:

```
List <T> al = new ArrayList<> ();
List <T> ll = new LinkedList<> ();
List <T> v = new Vector<> ();
Where T is the type of the object
```

The classes which implement the List interface are as follows:

ArrayList :

ArrayList provides a dynamic array in Java that resizes automatically as elements are added or removed. Although slower than standard arrays, it is efficient for frequent modifications. It supports random access but cannot store primitive types directly—wrapper classes like Integer or Character are required

Let's understand the ArrayList with the following example:

Example :

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

class GFG {

    // Main Method
    public static void main(String[] args)
    {

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

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

```
            al.add(i);

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

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

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

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

Output :

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Methods that are specific to the ArrayList :

1. ensureCapacity(int minCapacity): Increases the capacity of the ArrayList to ensure it can hold at least the specified number of elements.

2. trimToSize(): Trims the capacity of the ArrayList to its current size, minimizing storage overhead.

3. get(int index): Retrieves the element at the specified position in the list.
4. set(int index, E element): Replaces the element at the specified position with the provided element.

5. add(int index, E element): Inserts the specified element at the specified position in the list.

6. remove(int index): Removes the element at the specified position in the list.

7. int indexOf(Object o): Returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.

8. int lastIndexOf(Object o): Returns the index of the last occurrence of the specified element in the list, or -1 if the list does not contain the element.

9. ListIterator<E> listIterator(): Returns a list iterator over the elements in the list (in proper sequence).

10. ListIterator<E> listIterator(int index): Returns a list iterator over the elements in the list, starting at the specified position in the list.

11. List<E> subList(int fromIndex, int toIndex): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

LinkedList :

LinkedList is a linear data structure where elements (nodes) are stored non-contiguously. Each node contains data and a reference to the next (and optionally previous) node, forming a chain of elements linked by pointers.
Let's understand the LinkedList with the following example:

Example :

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

class GFG {

    // Main Method
    public static void main(String[] args)
    {

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

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

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

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

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

Output :

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Methods specific to the LinkedList class :

1.addFirst(E e): Inserts the specified element at the beginning of the list.

2.addLast(E e): Appends the specified element to the end of the list.

3.removeFirst(): Removes and returns the first element of the list.

4.removeLast(): Removes and returns the last element of the list.

5.getFirst(): Retrieves, but does not remove, the first element of the list.

6.getLast(): Retrieves, but does not remove, the last element of the list.

7.peekFirst(): Retrieves, but does not remove, the first element, or returns null if the list is empty.

8.peekLast(): Retrieves, but does not remove, the last element, or returns null if the list is empty.

9.pollFirst(): Retrieves and removes the first element, or returns null if the list is empty.

10.pollLast(): Retrieves and removes the last element, or returns null if the list is empty.

Vector :

Vector provides a dynamic array in Java, similar to ArrayList, but with synchronized methods for thread safety. While slower due to synchronization overhead, it is useful in multi-threaded environments. Like ArrayList, it resizes automatically during element manipulation.
Let's understand the Vector with an example:

Example :

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

class GFG {

  // Main Method
  public static void main(String[] args)
  {

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

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

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

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

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

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

Output :

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

some of the methods unique to Vector :

•**addElement(E obj):** Adds the specified component to the end of the vector, increasing its size by one. This method is similar to add(E e), but it is unique to Vector.

•**elementAt(int index):** Returns the element at the specified position in the vector. It is similar to get(int
index), but it is specific to Vector.

•**firstElement():** Returns the first component (element) of the vector.

•**lastElement():** Returns the last component (element) of the vector.

•**removeElement(Object obj):** Removes the first occurrence of the specified element from the vector. This is
somewhat similar to remove(Object o), but is specific to Vector.

•**removeElementAt(int index):** Removes the element at the specified position in the vector. This is similar to
remove(int index), but specific to Vector.

•**setSize(int newSize):** Sets the size of the vector. If the new size is greater than the current size, new
elements are added with null values. If the new size is smaller, elements are truncated.

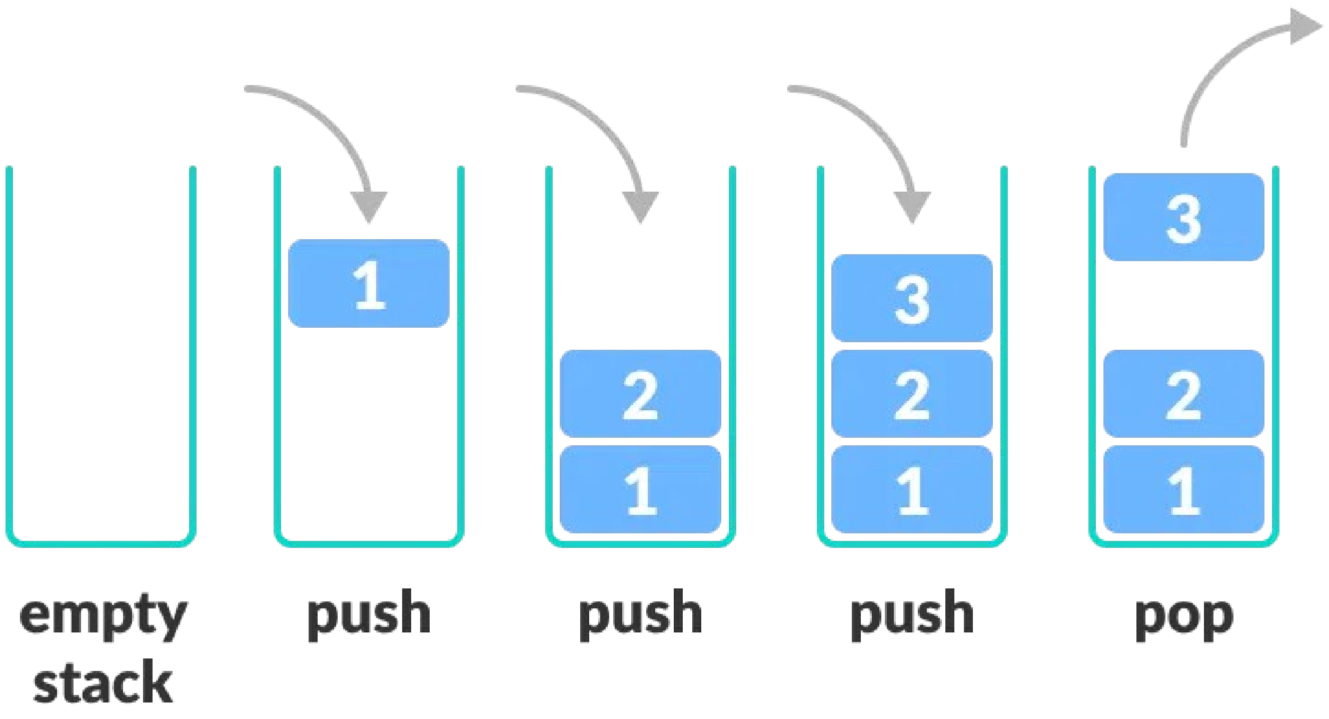
•**ensureCapacity(int minCapacity):** Ensures that the vector has at least the specified number of elements,
expanding its capacity if necessary. This is used to prevent frequent resizing.

•**capacity():** Returns the current capacity of the vector, which is the size of the internal array used to store the
elements.

•**trimToSize():** Trims the capacity of the vector to be the vector's current size, which can help to reduce
memory usage.

Stack :

Stack class implements the LIFO (last-in-first-out) data structure. It supports core operations like push() and pop(), along with peek(), empty(), and search(). Stack is a subclass of Vector and inherits its properties.
Let's understand the stack with an example:



Example :

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

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

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

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

        System.out.println();

        stack.pop();

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

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

Output :

```
Geeks For Geeks Geeks
Geeks For Geeks
```

Core Methods :

•**push(E item):** Adds an element to the top of the stack.
•**pop():** Removes and returns the element from the top of the stack.
•**peek():** Returns the element at the top of the stack without removing it.
•**empty():** Tests whether the stack is empty (legacy method).
•**search(Object o):** Returns the 1-based position of the object from the top of the stack. Returns -1 if the object is not found.

Note:
Stack is a subclass of Vector and a legacy class. It is thread-safe which might be overhead in an environment where thread safety is not needed. An alternate to Stack is to use ArrayDequeue which is not thread-safe and has faster array implementation.

4. Queue Interface :

The Queue interface follows the FIFO (First-In, First-Out) principle, where elements are processed in the order they are added—similar to a real-world queue (e.g., ticket booking). It is used when order matters. Classes like PriorityQueue and ArrayDeque implement this interface, allowing queue objects to be instantiated accordingly.

For example:

Queue <T> pq = new PriorityQueue<> ();

Queue <T> ad = new ArrayDeque<> ();

Where T is the type of the object.

The most frequently used implementation of the queue interface is the PriorityQueue.

Priority Queue

PriorityQueue processes elements based on their priority rather than insertion order. It uses a priority heap for internal storage. Elements are ordered either by their natural ordering or by a custom Comparator provided at construction.
Let's understand the priority queue with an example:

Example :

```
import java.util.*;

class GfG {

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

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

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

      // Printing the top element and removing it
      // from the PriorityQueue container
```

```
        System.out.println(pQueue.poll());

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

Output :

```
10
10
15
```

5. Deque Interface :

Deque interface :

extends Queue and allows insertion and removal of elements from both ends. It is implemented by classes like ArrayDeque, which can be used to instantiate a Deque object.

For example:

Deque<T> ad = new ArrayDeque<> ();
Where T is the type of the object.

The class which implements the deque interface is ArrayDeque.
ArrayDeque

ArrayDeque :

class implements a resizable, double-ended queue that allows insertion and removal from both ends. It has no capacity restrictions and grows automatically as needed.
Let's understand ArrayDeque with an example:

Examle :

```
import java.util.*;
public class ArrayDequeDemo {
    public static void main(String[] args)
    {
        // Initializing an deque
        ArrayDeque<Integer> de_que
            = new ArrayDeque<Integer>(10);

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

        System.out.println(de_que);

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

        // addFirst() method to insert the
        // elements at the head
        de_que.addFirst(564);
        de_que.addFirst(291);
        // addLast() method to insert the
        // elements at the tail
        de_que.addLast(24);
        de_que.addLast(14);

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

Output :

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

Methods of Deque Interface:

- **add():** The add() method adds an element to the end of the deque. This method returns true if the element is added to the deque and throws an exception if the element cannot be added to the deque.
- **AddFirst():** The addFirst() method adds elements to the beginning of deque. This method returns true if the element is added to the deque and throws an exception if the element cannot be added to the deque.
- **AddLast():** The addLast() method adds elements to the end of the deque. This method returns true if the element is added to the deque and throws an exception if the element cannot be added to the deque. This method is equivalent to the add() method.
- **Offer():** The offer() method adds an element to the end of the deque. This method returns true if the element is added to the deque and false if the element cannot be added to the deque.

- **OfferFirst():** The offerFirst() method adds an element to the end of the deque. This method returns true if the element is added to the deque and false if the element cannot be added to the deque.
- **OfferLast():** The offerLast() method adds an element to the end of the deque. This method returns true if the element is added to the end of the deque and false if the element cannot be added to the deque.
- **Push() :** The push() method adds elements to the beginning of deque. This method doesn't return any value if the element is added to the deque and throws an exception when the element cannot be added to the deque.
- **Pop():**   pop() method removes and returns the first element of the deque. This method throws an exception if the deque is empty. This method internally called the removeFirst() bmethod, which we will see shortly.
- **Remove():** The remove() method removes and returns the first element of the deque. This method throws an exception if the deque is empty. This method is internally called the removeFirst() method.
- **RemoveFirst() :** The removeFirst() method removes and returns the first element of the deque. This method throws an exception if the deque is empty.
- **RemoveLast():** The removeLast() method removes and returns the last element of the deque. This method throws an exception if the deque is empty.
- **Poll() :** The poll() method removes and returns the first element of the deque. This method returns null if the deque is empty.

- **PollFirst():** The pollFirst() method removes and returns the first element of the deque. This method returns null if the deque is empty.
- **pollLast():** The pollLast() method removes and returns the last element of the deque. This method returns null if the deque is empty
- **peek() :**The peak() method returns the first element of the deque without removing it. This method returns null if the deque is empty.
- **PeekFirst():** The peakFirst() method returns the first element of the deque without removing it. It is equivalent to the peak() method. This method returns null if the deque is empty.
- **PeekLast():** The peakLast() method returns the last element of the deque without removing it. This method returns null if the deque is empty.
- **GetFirst():** The getFirst() method returns the first element of the deque without removing it. This method throws an exception if the deque is empty.
- **GetLast():** The getLast() method returns the last element of the deque without removing it. This method throws an exception if the deque is empty.
- **pollLast():** The pollLast() method removes and returns the last element of the deque. This method returns null if the deque is empty.

6. Set Interface

Set :

interface represents an unordered collection that stores only unique elements (no duplicates). It's implemented by classes like HashSet, TreeSet, and LinkedHashSet, and can be instantiated using any of these

For example:

Set<T> hs = new HashSet<> ();

Set<T> lhs = new LinkedHashSet<> ();

Set<T> ts = new TreeSet<> ();

Where T is the type of the object.

Methods :

- **add(element):** This method is used to add a specific element to the set. The function adds the element only if the specified element is not already present in the set else the function returns False if the element is already present in the Set.
- **addAll(collection):** This method is used to append all of the elements from the mentioned collection to the existing set. The elements are added randomly without following any specific order.
- **clear():**This method is used to remove all the elements from the set but not delete the set. The reference for the set still exists.
- **contains(element):**This method is used to check whether the set contains all the elements present in the given collection or not. This method returns true if the set contains all the elements and returns false if any of the elements are missing.
- **HashCode():** This method is used to get the hashCode value for this instance of the Set. It returns an integer value which is the hashCode value for this instance of the Set.
- **IsEmpty():** This method is used to check whether the set is empty or not.
- **Iterator():** This method is used to return the iterator of the set. The elements from the set are returned in a random order.
- **remove(element):** This method is used to remove the given element from the set. This method returns True if the specified element is present in the Set otherwise it returns False.
- **removeAll(collection):** This method is used to remove all the elements from the collection which are present in the set. This method returns true if this set
- changed as a result of the call.

- **retainAll(collection):** This method is used to retain all the elements from the set which are mentioned in the given collection. This method returns true if this set changed as a result of the call.
- **Size():** This method is used to get the size of the set. This returns an integer value which signifies the number of elements.
- **ToArray():** This method is used to form an array of the same elements as that of the Set.

The following are the classes that implement the Set interface:

HashSet

HashSet class implements a hash table and stores elements based on their hash codes. It does not guarantee insertion order and allows one null element.

Example:

```
import java.util.*;

public class HashSetDemo {

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

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

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

```
        }
    }
}
```

Output :

```
Very helpful
Geeks
For
Is
```

Methods in HashSet:

- **add(Element):** Used to add the specified element if it is not present, if it is present then return false.
- **Clear():** Used to remove all the elements from the set
- **contains(Object o):** Used to return true if an element is present in a set.
- **remove(Object o) :** Used to remove the element if it is present in set.
- **iterator():** Used to return an iterator over the element in the set.
- **IsEmpty():** Used to check whether the set is empty or not. Returns true for empty and false for a non-empty condition for set.
- **Size():** Used to return the size of the set.
- **Clone():** Used to create a shallow copy of the set.

LinkedHashSet :

LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.
Let's understand the LinkedHashSet with an example:

Example :

```
import java.util.*;

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

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

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

Output :

```
Geeks
For
Is
Very helpful
```

Methods :

- **add(Element):** nserts the specified element into the set.
- **AddAll:** dds all elements from the specified collection to the set.
- **ceiling(Element):** eturns the last element in the set greater than or equal to the element or null if there is no such element.
- **Clear():** emoves all elements from the set.
- **contains(Object o):**Checks if the set contains the specified element.

descendingIterator(): Returns an iterator over the elements in descending order.
first(): Returns the first (lowest) element in the set.
headSet(E toElement): Returns a view of the portion of the set strictly less than the givenelement.
isEmpty(): Checks if the set is empty.
iterator(): Returns an iterator over the elements in ascending order.
last(): Returns the last (highest) element of the set.
pollFirst(): Retrieves and removes the first element of the set.
pollLast(): Retrieves and removes the last element of the set

7. Sorted Set Interface :

Sorted Set interface extends Set and maintains elements in sorted order. It includes additional methods for range views and ordering. It is implemented by the TreeSet class.

For example :

SortedSet<T> ts = new TreeSet<> ();
Where T is the type of the object.
The class which implements the sorted set interface is TreeSet.

TreeSet :

TreeSet uses a self-balancing tree (Red-Black Tree) to store elements in sorted order. It maintains natural ordering or uses a custom Comparator if provided during creation. Ordering must be consistent with equals to ensure proper Set behavior.
Let's understand TreeSet with an example:

Example :

```
import java.util.*;

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

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

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

Output :

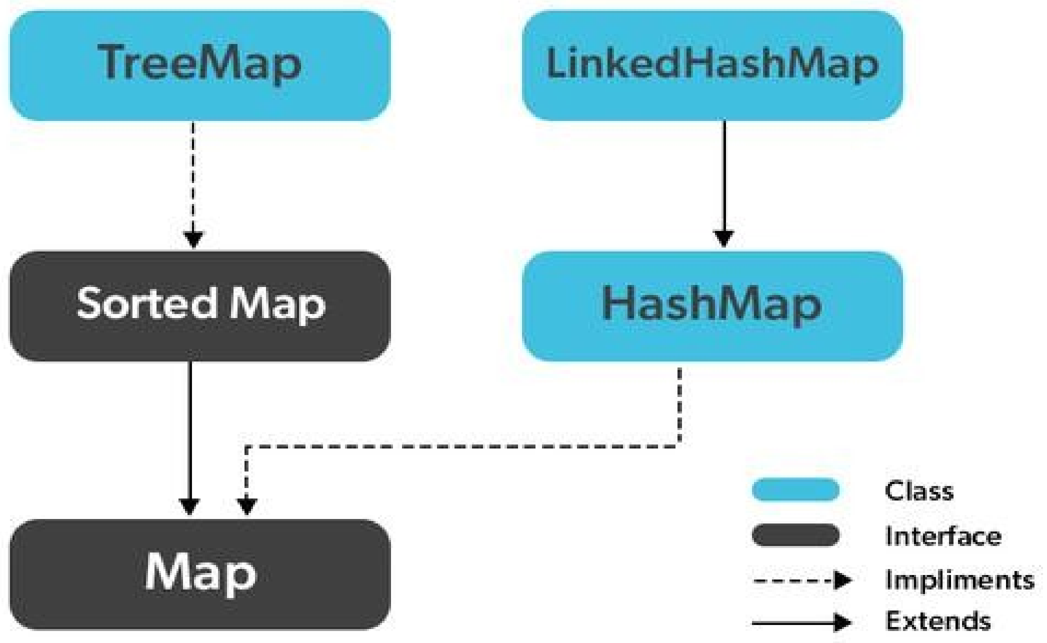
```
For
Geeks
Is
Very helpful
```

Java Collections Comparison Table :

Name	ArrayList 1.2	Vector 1.0	LinkedList1.2	PriorityQueue1.5	HashSet1.2	LinkedHashSet1.4	TreeSet1.2	HashMap1.2	LinkedHashMap1.4	TreeMap1.2
Parent	List	List	List	Queue	Set	Set	Set	Map	Map	Map
Operation	get()	get()	Insertion & Deletion	FCFS	Searching	Cache-based	Sorting	Searching	Cache-based	Sorting
Properities								**KEY VALUE**	**KEY VALUE**	**KEY VALUE**
1. Heterogeneous	YES	YES	YES	NO	YES	YES	NO	YES YES	YES YES	NO YES
2. Duplicate	YES	YES	YES	YES	NO	NO	NO	NO YES	NO YES	NO YES
3. Null Values	YES	YES	YES	NO	YES	YES	NO	YES YES	YES YES	NO YES
4. Insertion Order	YES	YES	YES	NO	NO	YES	NO	NO	YES	NO
5. Default Capacity	10	10		11	16	16	16	16	16	16
RandomAccess	YES	YES								
Synchronized	No	YES	NO	NO	NO	NO	NO	NO	NO	NO

Map Interface :

Map is a data structure that supports the key-value pair for mapping the data. This interface doesn't support duplicate keys because the same key cannot have multiple mappings, however, it allows duplicate values in different keys. A map is useful if there is data and we wish to perform operations on the basis of the key. This map interface is implemented by various classes like **HashMap**, **TreeMap**, etc. Since all the subclasses implement the map, we can instantiate a map object with any of these classes.



For example:

Map<T> hm = new HashMap<> ();
Map<T> tm = new TreeMap<> ();

Where T is the type of the object.

The frequently used implementation of a Map interface is a HashMap.

Methods in Map Interface:

- **put(Key, Value):** It is used to insert an entry in the map.
- **get(Key):** This method returns the object that contains the value associated with the key.
- **containsKey(Key):**If a key matching the key exists in the map, this method return true; otherwise, it returns false.
- **containsValue(Value):** If a value matching the value exists in the map, this method
- **returns true;** otherwise, it returns false.
- **isEmpty():** When the map has no keys, this method returns true; when there are keys, it return false.
- **Clear():** This method is used to clear all the elements in a map
- **remove(Key):** You can use it to remove an entry for the given key.
- **remove(key, value):** The given values and their corresponding specified keys are
- removed from the map.
- **size():** The number of key/value pairs in the map is returned by this method.
- **equals(Object):** The given Object and the Map are compared using this method.
- **EntrySet():** It returns the Set view, which includes each key and value.
- **HashCode():** It returns the Map's hash code value.
- **KeySet(): I**t returns the Set view with all the keys contained in it.
- **putAll(Map):** It is used to put the specified map into the map.
- r**eplace(key, value):** For a given key, it swaps out the given value.
- **replace(key, oldValue, newValue):** For a given key, it swaps out the old value with
- the new one.
- **Values():** The values contained in the map are returned as a collection view.
- The getOrDefault(key, defaultValue) method of
- Properties class is used to get the value mapped to this key, passed as the parameter,
- in this Properties object. This method will fetch the corresponding value to this key,
- if present, and return it. If there is no such mapping, then it returns the defaultValue.

HashMap :

HashMap is a basic implementation of the Map interface that stores data as key-value pairs. It uses hashing for fast access, converting keys into hash codes to index values efficiently. To retrieve a value, the corresponding key is required. Internally, HashSet is also backed by a HashMap.

Let's understand the HashMap with an example:

Example :

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

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

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

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

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

Output :

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

Methods of the Collection Interface :

This interface contains various methods which can be directly used by all the collections which implement this interface. They are:

Method	Description
add(Object)	This method is used to add an object to the collection.
addAll(Collection c)	This method adds all the elements in the given collection to this collection.
clear()	This method removes all of the elements from this collection.
contains(Object o)	This method returns true if the collection contains the specified element.
containsAll(Collection c)	This method returns true if the collection contains all of the elements in the given collection.
equals(Object o)	This method compares the specified object with this collection for equality.
hashCode()	This method is used to return the hash code value for this collection.
isEmpty()	This method returns true if this collection contains no elements.
iterator()	This method returns an iterator over the elements in this collection.
parallelStream()	This method returns a parallel Stream with this collection as its source.
remove(Object o)	This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object.

Method	Description
removeAll(Collection c)	This method is used to remove all the objects mentioned in the given collection from the collection.
removeIf(Predicate filter)	This method is used to remove all the elements of this collection that satisfy the given predicate.
retainAll(Collection c)	This method is used to retain only the elements in this collection that are contained in the specified collection.
size()	This method is used to return the number of elements in the collection.
spliterator()	This method is used to create a Spliterator over the elements in this collection.
stream()	This method is used to return a sequential Stream with this collection as its source.
toArray()	This method is used to return an array containing all of the elements in this collection.

1. Comparable vs Comparator :

Comparable :

- Part of java.lang
- Natural ordering
- compareTo(Object o)

Comparator :

- Part of java.util
- Custom ordering
- compare(Object o1, Object o2)
- Example: class Student implements Comparable

Example :

```
class Student implements
   Comparable<Student> { public int
   compareTo(Student s) {
      return this.age - s.age;
   }
}


class AgeComparator implements
   Comparator<Student> { public int
   compare(Student s1, Student s2) {
      return s2.age - s1.age;
   }  }
```

Fail-Fast vs Fail-Safe :

Fail-Fast:

- Throws ConcurrentModificationException
- Not thread-safe
- Examples: ArrayList, HashMap

Fail-Safe:

No exception if modified
Thread-safe
Examples: CopyOnWriteArrayList, ConcurrentHashMap

Example:
for (String s : list)

```
{
list.add("C");   // Fail-Fast: Exception
 }
```

3. Synchronized vs Concurrent Collections :

Synchronized:

Collections.synchronizedList()

Thread-safe, less efficient

Concurrent:

CopyOnWriteArrayList, ConcurrentHashMap

Thread-safe, more efficient

Example:

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
CopyOnWriteArrayList<String> cowList = new CopyOnWriteArrayList<>();
```

WeakHashMap vs HashMap vs IdentityHashMap :

HashMap:

Uses .equals() for key comparison

Keys are not GC'd

WeakHashMap:

Uses .equals()

Keys eligible for GC

IdentityHashMap:

Uses == for key comparison

Example:

Map<Object, String> map = new

WeakHashMap<>(); Object key = new Object();

map.put(key,

"value"); key = null;

System.gc(); // key may be removed

Knowledge – Check :

1) What is Collection and why we need it?
2) What is Collection framework?
3) What is List and when to use it?
4) How ArrayList works internally?
5) How LinkedList works internally?
6) When to use ArrayList and When to Use LinkedList?
7) What is Cursor and How many cursors available?
8) What is Set?
9) How HashSet works internally?
10) What is TreeSet and when to use it?
11) What is Comparable?
12) What is Comparator?
13) How to sort Objects?
14) What is Map and when to use it?
15) What is Hash Map?
16) How HashMap works internally?

17) How to iterate a Map?

18) What is Weak HashMap?

19) What is Identity HashMap?

20) What is Collections?

21) What is Properties Class?

22) What are Fail-fast collection and Fail-Safe Collection?.

Referance :

https://www.geeksforgeeks.org/java/collections-in-java-2/
https://www.w3schools.com/java/java_collections.asp
https://www.programiz.com/java-programming/collection-interface