Collections Framework

Collections framework

1. *Collections overview,*
2. *Collections Interfaces-*
3. *List Interface.*

Collections Class

1. *ArrayList class*
2. *. Accessing a Collection via an Iterator*

Collections Framework in Java - Overview

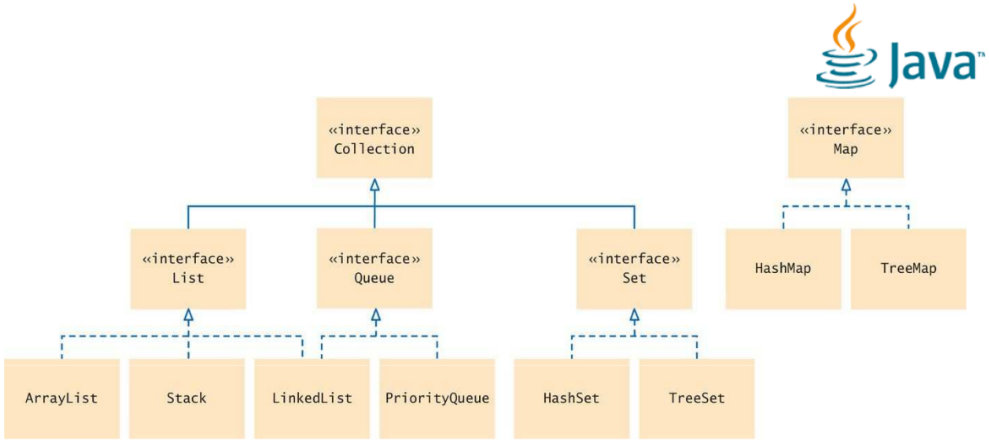

Figure 1 Interfaces and Classes in the Java Collections Framework

Introduction

1. The Java platform includes a collections framework.
2. A collection is an object that represents a group of objects (such as the classic Vector class).
3. A collections framework is a unified architecture for representing and manipulating collections, enabling collections to be manipulated independently of implementation details.

*"Collection Framework is a **group of classes** and **interfaces**, that implements commonly reusable data structures and algorithms to manipulate them."*

Advantages – or Goals of Collection Framework

The primary advantages of a collections framework are that it:

1. **Flexibility**: Collections offer a wide range of data structures suitable for various scenarios.
2. **Ease of Use**: They provide a consistent and easy-to-use interface for manipulating collections of objects.
3. **Interoperability**: Collections allow different collection types to work together using common interfaces.
4. **Efficiency**: The framework includes optimized implementations for different data structures and algorithms, ensuring efficient performance.

5. **Software reuse:** by providing a standard interface for collections and algorithms with which to manipulate them.

Key Components of Collection Framework:

Key components are

1. Interface.
2. Classes
3. Algorithms

1. **Interfaces:**
   - Interfaces like `Collection`, `List`, `Set`, `Map`, `Queue`, etc., define behaviors and contracts that different collection types must adhere to.
   - These interfaces provide a unified way to work with collections, allowing similar operations to be performed on different collection types.
2. **Classes:**
   - Classes like `ArrayList`, `LinkedList`, `HashSet`, `HashMap`, `PriorityQueue`, etc., are implementations of the collection interfaces. These classes provide different data structures with various behaviors and performance characteristics.
3. **Algorithms:**
   - The Collections Framework includes algorithms that operate on collections, such as sorting, searching, shuffling, etc. These algorithms can be used with different collection types, providing standardized methods for common operations.

   implementations that are commonly used such as sorting, searching etc.
   – void sort(List list)
   – int binarySearch(List list, Object value)

Collection Interfaces

1. The `Collection` interface in Java is the root interface of the Java Collections Framework.
2. It sits at the top of the collection hierarchy, defining the fundamental behavior and methods that all collection classes must support.
3. This interface serves as a foundation for various types of collections in Java.

4. *The collection interfaces are divided **into two groups**.*

   1. *The most basic interface, java.util.Collection*
      (Collections store data as List, Queue or Sets)
      - Top level sub-interfaces are
      a) ***java.util.Set <E>***
      b) ***java.util.Queue <E>***
      c) ***java.util.List <E>***

   2. *The other collection interfaces are based on java.util.Map*
      (Maps store data in Key-value pair)
      - Top level sub-interfaces are
      a) ***java.util.SortedMap<Key, Val>***
      b) ***java.util.ConcurrentMap<K,V>***

Sub Interfaces of java.util.Collection & java.util.Map

1. java.util.Collection *has the following descendants:*

public interface Collection<E>

All interfaces ***extends Collection***

java.util.Set

def: public interface Set<E> extends **Collection**<E>

java.util.SortedSet

def: public interface SortedSet<E> extends **Set**<E>

java.util.NavigableSet

def: public interface NavigableSet<E> extends **SortedSet**<E>

java.util.Queue:

def : public interface Queue<E> extends **Collection**<E>

java.util.concurrent.BlockingQueue

def: public interface BlockingQueue<E> extends **Queue**<E>

java.util.concurrent.TransferQueue

def: public interface TransferQueue<E> extends **BlockingQueue**<E>

java.util.Deque

def: public interface Deque<E> extends **Queue**<E>

java.util.concurrent.BlockingDeque

def: public interface BlockingDeque<E> extends **Deque**<E>

java.util.List

def: public interface List<E> extends **Collection**<E>

2. java.util.Map *has the following descendants:*

public interface Map<K,V>

All interfaces ***extends Map***

java.util.**SortedMap**

def: public interface SortedMap<K,V> extends **Map**<K,V>

java.util.NavigableMap

def: public interface NavigableMap<K,V> extends **SortedMap**<K,V>

java.util.concurrent.ConcurrentMap

def: public interface ConcurrentMap<K,V> extends **NavigableMap**<K,V>

java.util.concurrent.ConcurrentNavigableMap

def: public interface ConcurrentNavigableMap<K,V> extends

ConcurrentMap<K,V>

Abstract Funcions Defined in Collection Interface

boolean	*add(E e)*
boolean	*addAll(Collection<? extends E> c)*
boolean	*remove(Object o)*
boolean	*removeAll(Collection<?> c)*
boolean	*contains(Object o)*
boolean	*containsAll(Collection<?> c)*
boolean	*equals(Object o)*
boolean	*isEmpty()*
Object[]	***toArray()***

Collection Classes

These classes are the ***implementation of above interfaces***

Implementing ***classes of*** ***java.util.List:***

All classes ***implements List***

AbstractList, AbstractSequentialList, ***ArrayList*** *, AttributeList,*
CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList
Stack. Vector

Implementing ***classes of*** ***java.util.Map***

All classes ***implements Map***

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap,
ConcurrentSkipListMap, EnumMap, ***HashMap****,****Hashtable***
IdentityHashMap, LinkedHashMap, PrinterStateReasons
Properties, Provider, RenderingHints, SimpleBindings
TabularDataSupport, TreeMap, UIDefaults, WeakHashMap

Implementing ***classes of*** ***java.util.Set***

All classes ***implements Set***

AbstractSet, ConcurrentHashMap, KeySetView, ConcurrentSkipListSet
CopyOnWriteArraySet, EnumSet, ***HashSet****, JobStateReasons*
LinkedHashSet, TreeSet

Implementing ***classes of*** ***java.util.Queue***

All classes ***implements Queue***

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque,
ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque,
LinkedBlockingQueue, ***LinkedList****, LinkedTransferQueue,*
PriorityBlockingQueue, ***PriorityQueue****, SynchronousQueue*

Recent Changes To Collection Interface

Collections Framework underwent a fundamental change that significantly increased its power and streamlined its use.

– The changes were caused by the addition of

- generics
- autoboxing/unboxing, and
- for-each style for loop.

Generics

If a type is defined using **Generics**, it can work with any data type. Generics allow you to create classes, methods, and interfaces that are type-independent, meaning you can specify the data type later when you use or instantiate them

Generics in Java are defined using **angle brackets (<>)**. The angle brackets are used to **specify the type** parameter(s) when defining or using a generic class, interface, or method.
Question 1: what is Collection interface

Generics allow for type-safe code, ***enabling collections*** *to store a specific data type without the need for casting*. Collection Interfaces (e.g., List, Set, Map) are designed with Generics to support any object type, ensuring flexibility while maintaining type safety.

Example:

```
class GenericType <T> {
        T obj;
        GenericType(T obj) {
                this.obj = obj;
        }
        public T getObject () {
                return this.obj;
        }
}
public class GenericTypeTest {
        public static void main (String s[]) {
                GenericType<Integer> gInt = new GenericType<Integer>(10);
                System.out.println("Type = "+ gInt.getObject().getClass());
                System.out.println("Data = "+ gInt.getObject());
                GenericType<String> gString = new GenericType<String>("ASIET");
                System.out.println("Type = "+ gString.getObject().getClass());
                System.out.println("Data = "+ gString.getObject());
        }
}
```

Output

```
Type = class java.lang.Integer
Data = 10
Type = class java.lang.String
Data = ASIET
```

Autoboxing/unboxing

Add the ability to store **primitive types** in collections.

• IN THE PAST, if we wanted to store a primitive value, such as an int, in a collection, we had to manually box it into its type wrapper.
• When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type.

```
ArrayList<Integer> arrList = new ArrayList<Integer>();
int intData = 10;
// to add data - autoboxing
arrList.add(new Integer(intData);
// and to recover data - unboxing
int iData = ((Integer) arrList.get(0)).intValue();
```

This is called boxing/unbpxing.
Curerently this practie in not neededd...! Collections can work on primitive data types as well.

– Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types.

for-each style for loop.

The for-each loop (also known as the **enhanced for loop**) in Java is a simplified version of the traditional for loop, used to iterate over arrays and collections like List, Set, or Map.

The for-each loop makes the code more readable and eliminates the need to manage loop counters or access array/collection elements using indices.

Prototype

```
for (Type element : collection) {
   // Code to be executed for each element
}
```

Example 1:

Parse a traditional array

```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
   // print each elements
   System.out.println(num);
}
```

Example 2:

Parse an ArayList

```
ArrayList<String> list = new ArrayList<>();
list.add("Alice");
list.add("Bob");
list.add("Cindy");
for (String s: list) {
        // print each element
        System.out.println(s);
}
```

List Interface

Def:

public interface List<E> extends Collection<E>

1. **List** is an interface defined in Collection famework.
2. Inherits Collection Interface and introduces additional behaviors specific to lists
3. It represents ordered collection of elements starting from zero index.
4. It uses **Generics** to provide type safety and the type of elements in the list is determined by the generic type parameter `<E>`.
5. Generics allows a `List` to store any type of object while maintaining type safety, and `E` is a placeholder for that type.

Key characteristics and features of the `List` interface:

Ordered Collection:

Lists maintain the order of elements based on their insertion sequence.

Elements can be accessed and retrieved by their index, allowing **random access to elements**.

Allows Duplicates:

Unlike sets (e.g., Set implementations), lists permit duplicate elements. The same element can appear multiple times in a list.

Index-Based Access:

Elements can be inserted or accessed by their position in the list, using zero-based index.

Inherited functions from Collection Interface

boolean	*add(E e) : Adds an element of type E*
boolean	*addAll(Collection<? extends E> c) : add another collection of same type from given index.*
boolean	*remove(Object o) : remove given element*
boolean	*removeAll(Collection<?> c)*
boolean	*contains(Object o)*
boolean	*containsAll(Collection<?> c)*
boolean	*equals(Object o)*
boolean	*isEmpty()*
Object[]	***toArray()***

functions defined in List interface

<E> get(int index)

<E> set(int index, E element)

<E> remove(int index)

int indexOf(Object o)

int size()

Some of the implementing classes of List interface are

ArrayList , Stack, Vector,

AbstractList, AbstractSequentialList, AttributeList,

CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList,

ArrayList Class

Definition

public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
public abstract class AbstractList<E>
extends AbstractCollection<E>
implements List<E>
public abstract class AbstractCollection<E>
extends Object
implements Collection<E>

1. The `ArrayList` class in Java is part of the Java Collections Framework and is an implementation of the `List` interface.
2. It provides a dynamic array-like data structure that allows for flexible resizing of the underlying array to accommodate the addition or removal of elements.

Key characteristics and features of the ArrayList Class:

Dynamic Resizing:
ArrayList internally uses an array to store elements.
It dynamically increases its capacity as elements are added beyond the current capacity of the underlying array. This resizing is done automatically when needed.

Ordered Collection:
Maintains the order of elements based on their insertion sequence.
Elements can be accessed and retrieved by their index, allowing for random access to elements.

Allows Duplicates:
ArrayList permits duplicate elements.
The same element can appear multiple times in the list.

Resizable Capacity:
While ArrayList can dynamically grow in size as elements are added,
it can also shrink its capacity when elements are removed, freeing up memory.

Indexed Access:
Provides efficient indexed access, insertion, removal, and manipulation of elements at specific positions within the list.

Methods like get(int index), *add(int index, E element), remove(int index), set(int index, E element)*, etc., facilitate these operations.

Algorithm Support:
ArrayList supports algorithms like **sort(), iterator(), subList() etc**

Supported Consructors

1. **ArrayList()**
   Constructs an empty list with an initial capacity of ten.
2. **ArrayList(int initialCapacity)**
   Constructs an empty list with the specified initial capacity.

Supported Functions

1. **add(E element):** Appends the specified element to the end of the list.
2. **add(int index, E element):** Inserts the specified element at the specified position in the list.
3. **get(int index)**: Retrieves the element at the specified index in the list.
4. **set(int index, E element)**: Replaces the element at the specified position in the list with the specified element.
5. **remove(int index)**: Removes the element at the specified position in the list.
6. **remove(Object o)**: Removes the first occurrence of the specified element from the list.
7. **size()**: Returns the number of elements in the list.
8. **isEmpty()**: Returns true if the list is empty; otherwise, returns false.
9. **clear()**: Removes all the elements from the list.
10. **contains(Object o)**: Returns true if the list contains the specified element
11. **indexOf(Object o)**: Returns the index of the first occurrence of the specified element
12. **lastIndexOf(Object o)**: Returns the index of the last occurrence of the specified element

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

    /*-------------------list of String-----------------*/
        ArrayList<String> list = new ArrayList<>();
        list.add("Alice");
        list.add("Bob");
        list.add("Cindy");

        System.out.println("List size = " + list.size());

        // prase list
        for (int i=0; i<list.size();i++) {
            System.out.println("Strng at index["+i+"] =" + list.get(i));
        }

        System.out.println("Index of Alice = " + list.indexOf("Alice"));
        System.out.println("Index of Bob = " + list.indexOf("Bob"));
        System.out.println("Index of Cindy = " + list.indexOf("Cindy"));

        // add at an index
        System.out.println("Adding Sam at index 0");
        list.add(0,"Sam");

        // parse again
        // prase list
        for (int i=0; i<list.size();i++) {
            System.out.println("Strng at index["+i+"] =" + list.get(i));
        }

        // remove at index
        System.out.println("Removing at index 1, that is Alice");
        list.remove(1);

        // prase list
        for (int i=0; i<list.size();i++) {
            System.out.println("Strng at index["+i+"] =" + list.get(i));
```

```
        }
        // remove All
        System.out.println("Clearing list");
        list.clear();

        // print size
        System.out.println("List size = " + list.size());
```

Output

```
List size = 3
Strng at index[0] =Alice
Strng at index[1] =Bob
Strng at index[2] =Cindy
Index of Alice = 0
Index of Bob = 1
Index of Cindy = 2
Adding Sam at index 0
Strng at index[0] =Sam
Strng at index[1] =Alice
Strng at index[2] =Bob
Strng at index[3] =Cindy
Removing index 1, that is Alice
Strng at index[0] =Sam
Strng at index[1] =Bob
Strng at index[2] =Cindy
Clearing list
List size = 0
```

Iterator & ListIterator

public interface Iterator<E>
public interface ListIterator<E> extends Iterator<E>

1. An iterator in programming refers to an object that allows sequential access to elements within a collection, such as lists, sets, maps, arrays, and more.

2. It provides a way to traverse through the elements of a collection one by one and perform various operations like retrieving, removing, or iterating over the elements.

3. It offers a uniform way to access elements irrespective of the type of collection being used.

The **Iterator** provides three main methods:

boolean hasNext() :
Checks if there are more elements available in the collection

Object next() :
Retrieves the next element in the collection

The **ListIerator** provides some more APIs like

void remove() , boolean hasPrevious( ), E previous( ) etc

The Methods Defined by Iterator

1. **boolean hasNext( ):** Returns true if there are more elements. Otherwise, returns false.
2. **E next( ) :** Returns the next element.Throws NoSuchElementException if there is not a next element.
3. **void remove( ):** Removes the current element.Throws IllegalStateException if an attempt is made to call remove( ) that is not preceded by a call to next( ).

Method Defined by ListIterator

1. **void add(E obj):** Inserts obj into the list in front of the element that will be returned by the next call to next( ).
2. **boolean hasNext( ):** Returns true if there is a next element. Otherwise, returns false.
3. **boolean hasPrevious( ):** Returns true if there is a previous element. Otherwise, returns false.
4. **E next( ):** Returns the next element. NoSuchElementException is thrown if there is not a next element.
5. **int nextIndex( ):** Returns the index of the next element. If there is not a next element, returns the size of the list.
6. **E previous( ):** Returns the previous element. NoSuchElementException is thrown if there is not a previous element.
7. **int previousIndex( ):** Returns the index of the previous element. If there is not a previous element, returns −1
8. **void remove( ):** Removes the current element from the list. An IllegalStateException is thrown if remove( ) is called before next( ) or previous( ) is invoked.
9. **void set(E obj):** Assigns obj to the current element. This is the element last returned by a call to either next( ) or previous( ).

Example : Iterator

```
ArrayList<String> list = new ArrayList<>();
list.add("Alice");
list.add("Bob");
list.add("Cindy");

// iterator
System.out.println("parsing using Itrator");
Iterator it = list.iterator();
while(it.hasNext()) {
        String name = (String)it.next();
        System.out.println(name);
}
```

Output:

```
parsing using Itrator
Alice
Bob
Cindy
```

Example : ListIterator

It has function remove()

```
ArrayList<String> list = new ArrayList<>();
list.add("Alice");
list.add("Bob");
list.add("Cindy");

System.out.println("parsing using ListItrator to remove");
ListIterator lit = list.listIterator();
while(lit.hasNext()) {
        String name = (String)lit.next();
        if (name.equals("Bob") == true) {
                System.out.println("removing Bob");
                lit.remove();
        }
}

System.out.println("parsing using ListItrator");
while(lit.hasNext()) {
        String name = (String)lit.next();
        System.out.println(name);
}
System.out.println("List size = " + list.size());
```

Output:

```
parsing using ListItrator to remove
removing Bob
parsing using ListItrator
List size = 2
Strng at index[0] =Alice
Strng at index[1] =Cindy
```

Another Example

```
ArrayList <Book> bookShelf = new ArrayList <Book>();

Book book1 = new Book(1,"100 Years Of Solitude", "Gabriel G Marquez");
Book book2 = new Book(2,"Harry Poter", "J K Rawling");
Book book3 = new Book(3,"Godfather", "Mario Puzo");

public static void parseBookShelf(ArrayList<Book> shelf) {
    Iterator<Book> iterator = shelf.iterator();
    System.out.println("Books in the BookShelf:");
    while (iterator.hasNext()) {
       Book b = iterator.next();
       System.out.println(b.id + " " + b.title + "-" + b.author);
    }
  }
```

Output

```
Books in the BookShelf:
```

1 100 Years Of Solitude - Gabriel G Marquez
2 Harry Potter - J K Rowling
3 Godfather - Mario Puzo

Question

Question1. What are collection Interfaces ?
Ans:

The `Collection` interface in Java is the root interface of the Java Collections Framework. It sits at the top of the collection hierarchy, defining the fundamental behavior and methods that all collection classes must support. This interface serves as a foundation for various types of collections in Java. *Collection Framework is a **group of classes** and **interfaces**, that implements commonly reusable data structures and algorithms to manipulate them.*

Definition:
public interface Collection<E> extends Iterable<E>

*The collection interfaces are divided **into two groups**.*

3. *The most basic interface, java.util.Collection*
   public interface Collection<E>
4. *The other collection interfaces are based on java.util.Map*
   public interface Map<E>

Top level interfaces are
1. java.util.set : extends Collection
2. java.util.Queue : extends Collection
3. java.util.List : extends Collection
4. java.util.Map :
Coolection classes are defined by implemennting these interfaces.
For eg,

ArrayList(), LinkedList() are defined by implementing interface java.util.List<E>
HashMap(), Hashtable() are defined by implementing java.util.Map<E>
HashSet(), TreeSet() are defined y implemnting java.util.Set<E>
DelayQueue(), PriorityQueue() are defined by implementing java.util.Queue<E>

Question 2: what are the abstract classes defned in Collction interface
Ans: Basic interface functions defined in Collection interface:

boolean	add(E e)
boolean	addAll(Collection<? extends E> c)
boolean	remove(Object o)
boolean	removeAll(Collection<?> c)
boolean	contains(Object o)
boolean	containsAll(Collection<?> c)
boolean	equals(Object o)
boolean	isEmpty()
Object[]	toArray()

Question 3: what is Collection classes
Ans: *Coolection classes are defined by implemennting th interfaces defined in collection framework.*

Top level interfaces in Collection framework are

1. *java.util.set : extends Collection*
2. *java.util.Queue : extends Collection*
3. *java.util.List : extends Collection*
4. *java.util.Map :*

Coolection classes are defined by implemennting these interfaces.
For eg, below are some of the collection classes

ArrayList(), LinkedList(): *are defined by implementing interface java.util.List<E>*
HashMap(), Hashtable() : *are defined by implementing java.util.Map<E>*
HashSet(), TreeSet() : *are defined y implemnting java.util.Set<E>*
DelayQueue(), PriorityQueue() : *are defined by implementing java.util.Queue<E>*

Question 3: Differentiate between Collection Interface and Collections Class.

Ans:

The `Collection` interface in Java is the root interface of the Java Collections Framework.
It sits at the top of the collection hierarchy, defining the fundamental behavior and methods that all collection classes must support.
This interface serves as a foundation for various types of collections in Java.

*The collection interfaces are divided **into two groups**.*

The most basic interface, java.util.Collection
(Collections store data as List, Queue or Sets)
- Top level sub-interfaces are

d) ***java.util.Set <E>***
e) ***java.util.Queue <E>***
f) ***java.util.List <E>***

The other collection interfaces are based on java.util.Map
(Maps store data in Key-value pair)
- Top level sub-interfaces are

c) ***java.util.SortedMap<Key, Val>***
d) ***java.util.ConcurrentMap<K,V>***

`Collection Classes`

*These classes are the **implementation of above interfaces***

Implementing classes of java.util.List:

*Some classes **implements List***

*AbstractList, **ArrayList** , LinkedList, **Stack. Vector***

Implementing classes of java.util.Map

*Some classes **implements Map***

*AbstractMap, **HashMap**,**Hashtable,** TreeMap*

Implementing classes of java.util.Set

Some classes ***implements Set***

AbstractSet, ***HashSet****, LinkedHashSet, TreeSet*

Implementing ***classes of*** ***java.util.Queue***

Some classes ***implements Queue***

AbstractQueue, ArrayDeque DelayQueue, LinkedList, PriorityQueue.

All these classes implement the abstract functions defined in iterfaces along with there own concrete classes and algorithms.

Question 4: for each loop used to find sum of elements in collection

```
import java.util.*;
class ForEachDemo {
        public static void main(String args[]) {
                ArrayList<Integer> vals = new ArrayList<Integer>();
                vals.add(1);
                vals.add(2);
                vals.add(3);
                vals.add(4);
                vals.add(5);
                System.out.print("Original contents of vals: ");
                for(int v : vals)
                        System.out.print(v + " ");
                System.out.println();
                int sum = 0;
                for(int v : vals)
                        sum += v;
                System.out.println("Sum of values: " + sum);
        }
}
```

Question 5: What are the Excptions to be handles in List Interface

UnsupportedOperationException*: if the list cannot b e modified*

ClassCastException*: when one object is incompatible with another*

IndexOutOfBoundsException*: if an invalid index is used*

NullPointerException*: thrown if an attempt is made to store a null object and null elements are not allowed in the list.*

IllegalArgumentException*: if an invalid argument is used.*

Question 6: What are the Excptions to be handles in Collection Interface

UnsupportedOperationException*: throw if a collection cannot be modified.*

ClassCastException*- is generated when one object is incompatible with another.*

NullPointerException*: is thrown if an attempt is made to store a null object and null elements are not allowed in the collection.*

IllegalArgumentException*: is thrown if an invalid argument is used.*

IllegalStateException*: is thrown if an attempt is made to add an element to a fixed-length collection that is full.*