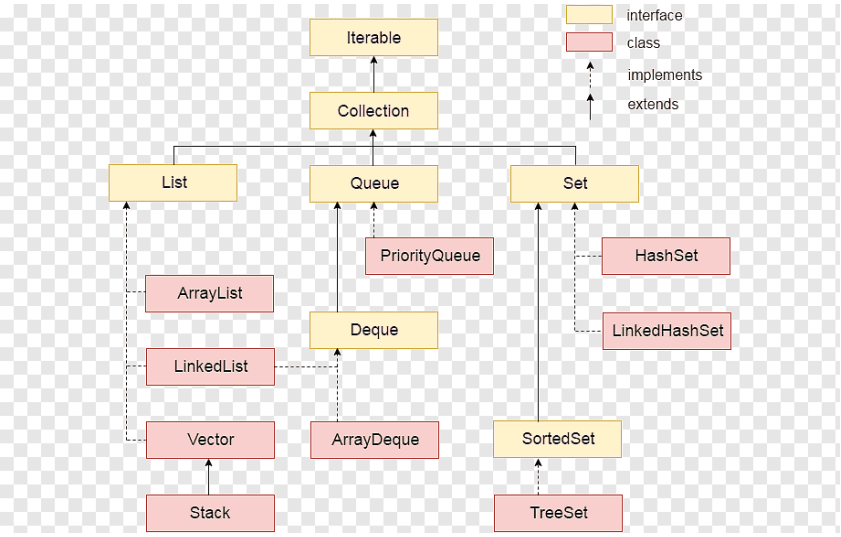
ADVANCED JAVA

Priyanshu Routh

Java Collection FrameWork

Introduction to the Java Collections Framework

The Java Collections Framework (JCF) is a powerful and essential part of the Java Standard Edition (Java SE) that provides a unified architecture for representing, manipulating, and managing collections of objects. Introduced in Java 2 (JDK 1.2), the framework offers a well-defined set of interfaces and classes to handle groups of objects, such as lists, sets, queues, and maps, making it easier to work with data structures and algorithms in Java applications.



Key Features of the Java Collections Framework

1. **Unified Architecture**: The JCF provides a consistent and standard way to work with collections, ensuring that various data structures can be used interchangeably and are easy to understand and implement.
2. **Type Safety with Generics**: Introduced in Java 5, generics allow collections to be parameterized with types, enhancing type safety and reducing runtime errors by catching potential issues at compile time.
3. **Enhanced Performance**: The JCF includes efficient data structures and algorithms, optimized for various operations such as insertion, deletion, traversal, and search, ensuring high performance in different scenarios.
4. **Extensive Utility Methods**: The `Collections` and `Arrays` utility classes provide a wide range of static methods for manipulating collections and arrays, including sorting, searching, shuffling, reversing, and more.
5. **Thread-Safety and Concurrency**: The JCF offers synchronized versions of collections and concurrent collections, enabling safe usage in multi-threaded environments and improving performance in concurrent applications.

Core Interfaces of the Java Collections Framework

1. **Collection**: The root interface representing a group of objects. It defines basic methods for adding, removing, and querying elements.
2. **List**: An ordered collection that allows duplicate elements. Common implementations include `ArrayList`, `LinkedList`, `Vector`, and `Stack`.
3. **Set**: A collection that does not allow duplicate elements. Common implementations include `HashSet`, `LinkedHashSet`, and `TreeSet`.
4. **Queue**: A collection that supports First-In-First-Out (FIFO) operations. Common implementations include `LinkedList` and `PriorityQueue`.
5. **Deque**: A double-ended queue supporting element insertion and removal at both ends. Common implementations include `ArrayDeque` and `LinkedList`.
6. **Map**: A collection that maps keys to values, ensuring unique keys. Common implementations include `HashMap`, `LinkedHashMap`, and `TreeMap`.

Implementations of the Java Collections Framework

1. **ArrayList**: A resizable array implementation of the `List` interface, providing fast random access and efficient iteration.
2. **LinkedList**: A doubly linked list implementation of the `List` and `Deque` interfaces, offering efficient insertion and deletion operations.
3. **HashSet**: A hash table-based implementation of the `Set` interface, providing constant-time performance for basic operations.
4. **TreeSet**: A Red-Black tree-based implementation of the `SortedSet` interface, maintaining elements in sorted order.
5. **HashMap**: A hash table-based implementation of the `Map` interface, offering constant-time performance for basic operations.
6. **TreeMap**: A Red-Black tree-based implementation of the `SortedMap` interface, maintaining keys in sorted order.
7. **PriorityQueue**: A priority heap-based implementation of the `Queue` interface, retrieving elements based on their natural ordering or a custom comparator.

Utility Classes

1. **Collections**: Provides static methods for common operations on collections, such as sorting, searching, shuffling, and more.
2. **Arrays**: Provides static methods for common operations on arrays, including sorting, searching, and conversion between arrays and collections.

List and Its Implementations in Java

The `List` interface is a part of the Java Collections Framework and represents an ordered collection of elements. It allows duplicate entries and provides precise control over where elements are inserted. The `List` interface extends the `Collection` interface and includes operations for positional access, search, iteration, and range-view manipulation.

Key Features of the List Interface

1. **Ordered Collection**: Elements are maintained in the order they are inserted.
2. **Allows Duplicates**: Multiple occurrences of the same element are allowed.
3. **Indexed Access**: Elements can be accessed by their position (index) in the list.
4. **Flexibility**: Various implementations provide flexibility in terms of performance and usage.

Common Methods in the List Interface

1. **add(E e)**: 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)**: Returns the element at the specified position 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. **indexOf(Object o)**: Returns the index of the first occurrence of the specified element in the list.
7. **lastIndexOf(Object o)**: Returns the index of the last occurrence of the specified element in the list.
8. **subList(int fromIndex, int toIndex)**: Returns a view of the portion of the list between the specified fromIndex, inclusive, and toIndex, exclusive.
9. **iterator()**: Returns an iterator over the elements in the list.
10. **listIterator()**: Returns a list iterator over the elements in the list.

```java
package Java_Collection_Framework;

import java.util.LinkedList;
import java.util.Iterator;

public class LinkedListExample {
    public static void main(String[] args) {
        // Create a LinkedList
        LinkedList<String> linkedList = new LinkedList<>();

        // Add elements
        linkedList.add("Java");
        linkedList.add("Python");
        linkedList.add("C++");

        // Add elements at the beginning and end
        linkedList.addFirst("HTML");
        linkedList.addLast("CSS");

        // Access elements
        System.out.println("First element: " + linkedList.getFirst());
        System.out.println("Last element: " + linkedList.getLast());
        System.out.println("Element at index 1: " +
linkedList.get(1));

        // Remove elements
        linkedList.removeFirst();
        linkedList.removeLast();
        System.out.println("After removals: " + linkedList);

        // Iterate over elements
        System.out.println("Iterating over LinkedList:");
        Iterator<String> iterator = linkedList.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // Check if an element exists
        System.out.println("Contains 'Python': " +
linkedList.contains("Python"));

        // Get the size of the LinkedList
        System.out.println("Size of LinkedList: " +
linkedList.size());
    }
}
```

Feature	Array	ArrayList	List Interface
Definition	Fixed-size, ordered collection of elements.	Resizable-array implementation of List.	An interface that represents an ordered collection.
Type	Built-in data structure in Java.	Part of Java Collections Framework.	Interface in the Java Collections Framework.
Size	Fixed at the time of creation.	Dynamic, can grow and shrink as needed.	Size can vary depending on the implementation.
Indexing	Supports direct indexing via bracket notation (`[]`).	Supports indexing through `get()` method.	Supports indexing through `get()` method.
Performance	Fast random access (`O(1)` time complexity).	Fast random access (`O(1)` time complexity).	Depends on the implementation (e.g., `ArrayList` or `LinkedList`).
Memory Usage	Memory is allocated at the time of creation, which can be wasteful if the array is not fully utilized.	Allocates memory dynamically as needed.	Depends on the implementation.
Type Safety	Can hold primitive types and objects.	Can only hold objects. Supports generics.	Can only hold objects. Supports generics.
Flexibility	Less flexible, as size is fixed.	More flexible due to dynamic resizing.	Very flexible, as various implementations are available.
Ease of Use	Requires manual handling of resizing and type casting.	Easier to use with built-in methods for resizing and type safety through generics.	Depends on the implementation.
Iterating	Can use for-loop, enhanced for-loop.	Can use for-loop, enhanced for-loop, iterators.	Can use for-loop, enhanced for-loop, iterators.
Thread Safety	Not thread-safe by default.	Not thread-safe by default. Can use `Collections.synchronizedList()` for synchronization.	Not thread-safe by default. Depends on the implementation.

ArrayList in Java

`ArrayList` is a part of the Java Collections Framework and is one of the most commonly used data structures. It implements the `List` interface, providing a dynamic array that can grow as needed. `ArrayList` offers a resizable array, which allows for efficient random access and manipulation of its elements. Here are detailed aspects of `ArrayList`:

Key Characteristics of ArrayList

1. **Resizable Array**: Unlike standard arrays in Java that have a fixed size, `ArrayList` can dynamically resize itself when elements are added or removed.
2. **Indexed Access**: Elements can be accessed directly by their index, providing fast retrieval (`O(1)` time complexity).
3. **Order of Elements**: Maintains the insertion order of elements.
4. **Allows Duplicates**: Elements can be duplicated.
5. **Allows Nulls**: `ArrayList` allows the insertion of null elements.

Common Methods

1. **add(E e)**: 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)**: Returns the element at the specified position 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. **clear()**: Removes all elements from the list.
9. **isEmpty()**: Returns true if the list contains no elements.
10. **contains(Object o)**: Returns true if the list contains the specified element.

Implementing ArrayList

```
package Java_Collection_Framework;

import java.util.ArrayList;

public class ArrayListExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        ArrayList<Integer>lintlist= new ArrayList<>();

        // Adding elements
        list.add("Java");
        list.add("Python");
        list.add("C++");
        //Adding numeric Elements
        lintlist.add(1);
        lintlist.add(5);
        lintlist.get(1);
        // Accessing elements
        System.out.println("Elements in the list:");
        for (String element : list) {
            System.out.println(element);
        }

        // Removing an element
        list.remove("Python");

        // Size of the list
        System.out.println("Size of the list: " +
list.size());
    }
}
```

Advantages of ArrayList

1. **Dynamic Sizing**: The `ArrayList` grows automatically when elements are added beyond its current capacity.
2. **Fast Random Access**: Direct access to elements by their index provides fast retrieval (`O(1)` time complexity).
3. **Ease of Use**: Provides a rich set of methods for manipulating and accessing the elements.

Disadvantages of ArrayList

1. **Slow Insertions/Deletions**: Inserting or deleting elements in the middle of the list requires shifting elements, resulting in slower performance (`O(n)` time complexity).
2. **Memory Overhead**: The `ArrayList` may allocate more space than needed to handle future growth, leading to potential memory overhead.

Conclusion

The `ArrayList` class is a versatile and widely-used implementation of the `List` interface, offering dynamic sizing, fast random access, and a comprehensive set of methods for managing collections of objects. Its ease of use and efficient performance make it a go-to choice for many common programming tasks involving dynamic arrays in Java.

LinkedList in Java Collection Framework

The `LinkedList` class in Java is a part of the Java Collections Framework (JCF) and implements the `List`, `Deque`, and `Queue` interfaces. It provides a doubly-linked list data structure, which is efficient for certain types of operations.

Key Features of LinkedList

1. **Doubly-Linked List**:
    - Each node contains references to both the previous and next nodes, enabling bidirectional traversal.
    - This structure allows for efficient insertion and deletion of elements at both ends of the list and in the middle.
2. **Implements Multiple Interfaces**:
    - Implements the `List` interface, providing ordered and sequential access to elements.
    - Implements the `Deque` interface, supporting operations to insert, remove, and examine elements at both ends.
    - Implements the `Queue` interface, making it suitable for use as a queue with additional methods for queue operations.
3. **Dynamic Size**:
    - The size of the `LinkedList` can grow and shrink dynamically as elements are added or removed.
4. **Efficient Insertions and Deletions**:
    - Insertion and deletion of elements at the beginning, middle, or end of the list are efficient (`O(1)` time complexity for adding/removing elements).
5. **Sequential Access**:
    - Accessing elements by index requires traversal from the beginning or end of the list, which has `O(n)` time complexity.

Common Methods in LinkedList

1. **add(E e)**: 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. **addFirst(E e)**: Inserts the specified element at the beginning of the list.
4. **addLast(E e)**: Appends the specified element to the end of the list.
5. **get(int index)**: Returns the element at the specified position in the list.
6. **getFirst()**: Returns the first element in the list.
7. **getLast()**: Returns the last element in the list.
8. **remove(int index)**: Removes the element at the specified position in the list.
9. **removeFirst()**: Removes and returns the first element in the list.
10. **removeLast()**: Removes and returns the last element in the list.
11. **contains(Object o)**: Returns true if the list contains the specified element.
12. **size()**: Returns the number of elements in the list.
13. **iterator()**: Returns an iterator over the elements in the list.

Example Code Using LinkedList

```
package Java_Collection_Framework;

import java.util.LinkedList;
import java.util.Iterator;

public class LinkedListExample {
    public static void main(String[] args) {
        // Create a LinkedList
        LinkedList<String> linkedList = new LinkedList<>();

        // Add elements
        linkedList.add("Java");
        linkedList.add("Python");
        linkedList.add("C++");

        // Add elements at the beginning and end
        linkedList.addFirst("HTML");
        linkedList.addLast("CSS");

        // Access elements
        System.out.println("First element: " + linkedList.getFirst());
        System.out.println("Last element: " + linkedList.getLast());
        System.out.println("Element at index 1: " +
linkedList.get(1));

        // Remove elements
        linkedList.removeFirst();
        linkedList.removeLast();
        System.out.println("After removals: " + linkedList);

        // Iterate over elements
        System.out.println("Iterating over LinkedList:");
        Iterator<String> iterator = linkedList.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // Check if an element exists
        System.out.println("Contains 'Python': " +
linkedList.contains("Python"));

        // Get the size of the LinkedList
        System.out.println("Size of LinkedList: " +
linkedList.size());
    }
}
```

Vector in Java Collection Framework

`Vector` is a part of the Java Collections Framework (JCF) and implements the `List` interface. It is a resizable array that provides thread-safe operations. Introduced in Java 1.0, `Vector` has been part of the Java API for a long time and was retrofitted to implement the `List` interface with the introduction of the Collections Framework in Java 2 (Java 1.2).

Key Features of Vector

1. **Resizable Array**:
    - Like `ArrayList`, `Vector` can dynamically resize itself as elements are added or removed.
    - It grows its size by a specified amount, or by doubling its size, to accommodate additional elements when the current capacity is exceeded.
2. **Thread-Safety**:
    - All methods of `Vector` are synchronized, making it thread-safe.
    - This synchronization means that only one thread can access the `Vector` at a time, which can be useful in multi-threaded environments but may lead to performance overhead compared to non-synchronized collections.
3. **Legacy Class**:
    - `Vector` is considered a legacy class, and its use is generally discouraged in favor of newer, unsynchronized collections like `ArrayList`, unless thread safety is specifically required.

Common Methods in Vector

1. **add(E e)**: Appends the specified element to the end of the vector.
2. **add(int index, E element)**: Inserts the specified element at the specified position in the vector.
3. **get(int index)**: Returns the element at the specified position in the vector.
4. **get(int index)**: Returns the element at the specified position in the vector.
5. **set(int index, E element)**: Replaces the element at the specified position in the vector with the specified element.
6. **remove(int index)**: Removes the element at the specified position in the vector.
7. **remove(Object o)**: Removes the first occurrence of the specified element from the vector.
8. **size()**: Returns the number of elements in the vector.
9. **contains(Object o)**: Returns `true` if the vector contains the specified element.
10. **isEmpty()**: Returns `true` if the vector contains no elements.
11. **elementAt(int index)**: Returns the element at the specified position in the vector (legacy method).

Vector Implemantion

```java
package Java_Collection_Framework;

import java.util.Vector;
import java.util.Iterator;

public class VectorExample {
    public static void main(String[] args) {
        // Create a Vector
        Vector<String> vector = new Vector<>();

        // Add elements
        vector.add("Java");
        vector.add("Python");
        vector.add("C++");

        // Add element at a specific position
        vector.add(1, "JavaScript");

        // Access elements
        System.out.println("Element at index 0: " + vector.get(0));
        System.out.println("Element at index 1: " + vector.get(1));

        // Replace element at a specific position
        vector.set(1, "TypeScript");
        System.out.println("Element at index 1 after replacement: " +
vector.get(1));

        // Remove elements
        vector.remove(0);
        vector.remove("C++");
        System.out.println("After removals: " + vector);

        // Check if vector contains an element
        System.out.println("Contains 'Python': " +
vector.contains("Python"));

        // Check size of vector
        System.out.println("Size of vector: " + vector.size());

        // Iterate over elements
        System.out.println("Iterating over vector:");
        Iterator<String> iterator = vector.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Stack in Java Collection Framework

The `Stack` class in Java is a part of the Java Collections Framework (JCF) and represents a last-in, first-out (LIFO) stack of objects. It extends the `Vector` class with five operations that allow a vector to be treated as a stack. Although `Stack` is considered a legacy class and newer implementations like `Deque` should be preferred for stack operations, it is still widely used and serves as an important example of LIFO data structures.

Key Features of Stack

1. **LIFO (Last-In, First-Out) Structure**:
    - `Stack` operates on a LIFO principle, meaning the last element added is the first one to be removed.
2. **Extends Vector**:
    - As `Stack` extends `Vector`, it inherits all the methods and properties of `Vector`.
    - This means `Stack` is also synchronized and thread-safe.
3. **Basic Stack Operations**:
    - `push(E item)`: Adds an item to the top of the stack.
    - `pop()`: Removes and returns the item at the top of the stack.
    - `peek()`: Returns the item at the top of the stack without removing it.
    - `empty()`: Tests if the stack is empty.
    - `search(Object o)`: Returns the 1-based position from the top of the stack where the object is located, or -1 if the object is not in the stack.

Common Methods in Stack

1. **push(E item)**: Pushes an item onto the top of the stack.
2. **pop()**: Removes the object at the top of the stack and returns it.
3. **peek()**: Looks at the object at the top of the stack without removing it.
4. **empty()**: Tests if the stack is empty.
5. **search(Object o)**: Returns the 1-based position from the top of the stack where the object is located.

Stack Implementation

```
package Java_Collection_Framework;

import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        // Create a Stack
        Stack<String> stack = new Stack<>();

        // Push elements onto the stack
        stack.push("Java");
        stack.push("Python");
        stack.push("C++");

        // Access the element at the top of the stack
        System.out.println("Element at the top: " +
stack.peek());

        // Pop the top element
        System.out.println("Popped element: " + stack.pop());

        // Check if the stack is empty
        System.out.println("Is stack empty? " + stack.empty());

        // Search for an element
        System.out.println("Position of 'Java': " +
stack.search("Java"));

        // Pop all elements
        while (!stack.empty()) {
            System.out.println("Popped: " + stack.pop());
        }

        // Check the stack is empty
        System.out.println("Is stack empty? " + stack.empty());
    }
}
```

Sets in Java Collection Framework

In the Java Collections Framework, a `Set` is a collection that does not allow duplicate elements. It models the mathematical set abstraction and provides several implementations, each with its unique properties and use cases. The primary `Set` implementations are `HashSet`, `LinkedHashSet`, and `TreeSet`.

Key Features of Sets

1. **No Duplicates**: Sets do not allow duplicate elements. If you attempt to add a duplicate element, the set remains unchanged.
2. **Unordered Collection**: By default, sets do not maintain any order of elements (though some implementations like `LinkedHashSet` and `TreeSet` do).
3. **Efficient Membership Testing**: Sets provide efficient methods for testing membership (checking if an element is in the set).

Common Methods in Set

1. **add(E e)**: Adds the specified element to the set if it is not already present
2. **remove(Object o)**: Removes the specified element from the set if it is present.
3. **contains(Object o)**: Returns true if the set contains the specified element.
4. **size()**: Returns the number of elements in the set.
5. **isEmpty()**: Returns true if the set contains no elements.
6. **clear()**: Removes all the elements from the set.
7. **iterator()**: Returns an iterator over the elements in the set.

Set Implementation

```
8. package Java_Collection_Framework;

   import java.util.HashSet;
   import java.util.Set;

   public class SetExample {
       public static void main(String[] args) {
           // Create a Set using HashSet
           Set<String> set = new HashSet<>();

           // Add elements to the Set
           set.add("Java");
           set.add("Python");
           set.add("C++");
           set.add("Java"); // Duplicate element, will not be
   added

           // Display the Set
           System.out.println("Set: " + set);

           // Check for membership
           System.out.println("Contains 'Python': " +
   set.contains("Python"));
           System.out.println("Contains 'Ruby': " +
   set.contains("Ruby"));

           // Remove an element
           set.remove("C++");
           System.out.println("After removal: " + set);

           // Get the size of the Set
           System.out.println("Size of set: " + set.size());

           // Iterate through the Set
           System.out.println("Iterating through the set:");
           for (String element : set) {
               System.out.println(element);
           }

           // Clear the Set
           set.clear();
           System.out.println("Is set empty after clearing? "
   + set.isEmpty());
       }
   }
```

HashSet in Java Collection Framework

`HashSet` is one of the most popular implementations of the `Set` interface in the Java Collections Framework. It is part of the `java.util` package and provides a collection that does not allow duplicate elements. `HashSet` is backed by a hash table (actually a `HashMap` instance), which ensures that the elements are unique and provides efficient operations.

Key Features of HashSet

1. **No Duplicates**: Ensures that no duplicate elements are present.
2. **Unordered**: Does not maintain any order of elements. The elements are stored in a way that is optimized for quick searches.
3. **Allows Null**: Can contain at most one null element.
4. **Constant-Time Performance**: Basic operations like add, remove, contains, and size have constant-time performance, on average.

Common Methods

1. **add(E e)**: Adds the specified element to the set if it is not already present.
2. **remove(Object o)**: Removes the specified element from the set if it is present.
3. **contains(Object o)**: Returns true if the set contains the specified element.
4. **size()**: Returns the number of elements in the set.
5. **isEmpty()**: Returns true if the set contains no elements.
6. **clear()**: Removes all the elements from the set.
7. **iterator()**: Returns an iterator over the elements in the set.

Internal Working

- **Hashing**: HashSet uses hashing to store its elements. The hash code of each element is used to determine its bucket location in the hash table. The effectiveness of HashSet largely depends on the quality of the hash function of the elements. A good hash function minimizes collisions (when two elements hash to the same bucket).
- **Load Factor and Capacity**:
    - **Load Factor**: A measure that determines when to increase the capacity of the hash table. It is a threshold for rehashing (increasing the size of the hash table).
    - **Capacity**: The number of buckets in the hash table. The default capacity is 16, and the default load factor is 0.75.

Performance Considerations

- **Collision Handling**: HashSet uses separate chaining (linked lists) to handle hash collisions. When two elements hash to the same bucket, they are stored in a linked list within that bucket.
- **Rehashing**: When the number of elements in the HashSet exceeds the product of its capacity and load factor, the hash table is resized (rehashing) to accommodate more

HashSet Implementation

```
package Java_Collection_Framework;

import java.util.HashSet;

public class HashSetExample {
    public static void main(String[] args) {
         HashSet<String> set = new HashSet<>();
         HashSet<Integer>set2=new HashSet<>();

        // Adding elements
        set.add("Apple");
        set.add("Banana");
        set.add("Orange");
        set.add("Orange");
        set2.add(1);
        set2.add(1);
        // Checking if an element exists
        System.out.println("Contains 'Banana': " +
set.contains("Banana"));

        // Iterating over elements
        System.out.println("Elements in the set:");
        for (String element : set) {
            System.out.println(element);
        }
        for(int i: set2)
        {
            System.out.println(set2);
        }

        // Removing an element
        set.remove("Orange");

        // Size of the set
        System.out.println("Size of the set: " + set.size());
    }
}
```

LinkedHashSet in Java Collection Framework

`LinkedHashSet` is a variant of `HashSet` that maintains the insertion order of elements. It combines the properties of `HashSet` with a linked list to preserve the order in which elements were added. This makes `LinkedHashSet` a useful choice when you need to maintain the order of insertion while benefiting from the performance characteristics of a hash-based collection.

Key Features of LinkedHashSet

1. **Ordered**:
    - `LinkedHashSet` maintains the order in which elements were added. This means that when iterating over a `LinkedHashSet`, elements are returned in their insertion order.
2. **Fast Access**:
    - It provides constant-time performance (O(1)) for basic operations such as `add`, `remove`, `contains`, and `size`, similar to `HashSet`.
3. **Allows Null**:
    - `LinkedHashSet` can contain at most one null element, just like `HashSet`.
4. **Consistent Order**:
    - The insertion order is preserved, which is useful in scenarios where you need to maintain the sequence of elements.

Common Methods

1. **`add(E e)`**: Adds the specified element to the set if it is not already present.
2. **remove(Object o)**: Removes the specified element from the set if it is present.
3. **contains(Object o)**: returns true if the set contains the specified element.
4. **size()**:Returns the number of elements in the set
5. **isEmpty()**:Returns true if the set contains no elements.
6. **clear()**:Removes all elements from the set.
7. **iterator()**:Returns an iterator over the elements in the set.

Internal Working

- **Hashing and Linked List**:
    - LinkedHashSet uses a hash table to provide fast access to elements and a doubly-linked list to maintain the order of insertion. Each element is stored in a node that contains pointers to the previous and next nodes, allowing for efficient ordering and traversal.
- **Load Factor and Capacity**:
    - Similar to HashSet, LinkedHashSet has a default initial capacity (16) and load factor (0.75). These parameters affect when the internal hash table will be resized.

Linked Hashset

```
package Java_Collection_Framework;

import java.util.Iterator;
import java.util.LinkedHashSet;

public class Linked_HashSet {
    public static void main(String[] args) {
        // Create a LinkedHashSet
        LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();

        // Add elements
        linkedHashSet.add("Java");
        linkedHashSet.add("Python");
        linkedHashSet.add("C++");
        linkedHashSet.add("Java"); // Duplicate element, will not be
added

        // Display the LinkedHashSet
        System.out.println("LinkedHashSet: " + linkedHashSet);

        // Check for membership
        System.out.println("Contains 'Python': " +
linkedHashSet.contains("Python"));

        // Remove an element
        linkedHashSet.remove("C++");
        System.out.println("After removal: " + linkedHashSet);

        // Get the size of the LinkedHashSet
        System.out.println("Size of LinkedHashSet: " +
linkedHashSet.size());

        // Iterate through the LinkedHashSet
        System.out.println("Iterating through the LinkedHashSet:");
        Iterator<String> iterator = linkedHashSet.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // Clear the LinkedHashSet
        linkedHashSet.clear();
        System.out.println("Is LinkedHashSet empty after clearing? " +
linkedHashSet.isEmpty());
    }
}
```

TreeSet in Java Collection Framework

`TreeSet` is an implementation of the `SortedSet` interface in the Java Collections Framework. It is part of the `java.util` package and is based on a NavigableMap, specifically a Red-Black Tree. `TreeSet` provides a collection of elements that are stored in a sorted, ascending order.

Key Features of TreeSet

1. **Sorted Order**:
    - `TreeSet` maintains the elements in a sorted order, based on their natural ordering or by a comparator provided at set creation time. The elements are stored in ascending order.
2. **NavigableSet**:
    - Implements the `NavigableSet` interface, which provides methods for navigating through the set (e.g., `higher`, `lower`, `ceiling`, `floor`).
3. **No Duplicates**:
    - `TreeSet` does not allow duplicate elements. If an attempt is made to add a duplicate element, the set remains unchanged.
4. **Performance**:
    - Basic operations such as `add`, `remove`, and `contains` are performed in logarithmic time (O(log n)) due to the underlying Red-Black Tree structure.
5. **Null Elements**:
    - `TreeSet` does not allow null elements. Attempting to add a null element will result in a `NullPointerException`.

Common Methods

1. **`add(E e)`**: Adds the specified element to the set if it is not already present. The element is added in the correct position to maintain the sorted order.
2. **remove(Object o)**: Removes the specified element from the set if it is present.
3. **contains(Object o)**: Returns true if the set contains the specified element.
4. **size()**:Returns the number of elements in the set.
5. **isEmpty()**:Returns true if the set contains no elements.
6. **clear()**:Removes all elements from the set.
7. **iterator()**:Returns an iterator over the elements in the set.
8. **first()**: Returns the first (lowest) element currently in the set.
9. **last()**:Returns the last (highest) element currently in the set.
10. **higher(E e)**: Returns the least element in the set that is strictly greater than the specified element, or null if there is no such element.
11. **lower(E e)**: Returns the greatest element in the set that is strictly less than the specified element, or null if there is no such element.

Treeset Example

```
package Java_Collection_Framework;

import java.util.TreeSet;
import java.util.Iterator;

public class TreeSetExample {
    public static void main(String[] args) {
        // Create a TreeSet
        TreeSet<String> treeSet = new TreeSet<>();

        // Add elements
        treeSet.add("Java");
        treeSet.add("Python");
        treeSet.add("C++");
        treeSet.add("Java"); // Duplicate element, will not be added

        // Display the TreeSet
        System.out.println("TreeSet: " + treeSet);

        // Check for membership
        System.out.println("Contains 'Python': " +
treeSet.contains("Python"));

        // Remove an element
        treeSet.remove("C++");
        System.out.println("After removal: " + treeSet);

        // Get the size of the TreeSet
        System.out.println("Size of TreeSet: " + treeSet.size());

        // Iterate through the TreeSet
        System.out.println("Iterating through the TreeSet:");
        Iterator<String> iterator = treeSet.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // Get first and last elements
        System.out.println("First element: " + treeSet.first());
        System.out.println("Last element: " + treeSet.last());

        // Get higher and lower elements
        System.out.println("Higher than 'Java': " + treeSet.higher("Java"));
        System.out.println("Lower than 'Python': " +
treeSet.lower("Python"));

        // Clear the TreeSet
        treeSet.clear();
        System.out.println("Is TreeSet empty after clearing? " +
treeSet.isEmpty());
    }
}
```

Introduction to Java Collections: Maps and Their Types

The Java Collections Framework offers a variety of data structures to handle collections of objects. One of the most important interfaces in this framework is the `Map` interface. A `Map` represents a collection of key-value pairs, where each key is associated with exactly one value. Unlike other collections, Maps do not allow duplicate keys, and they provide efficient methods for accessing, inserting, and deleting entries based on keys.

Java provides several implementations of the `Map` interface, each optimized for different use cases and performance characteristics. This guide will cover the main types of Maps in Java, their constructors, and their methods.

1. `HashMap`

`HashMap` is the most commonly used implementation of the `Map` interface. It provides constant-time performance for basic operations (like add, remove, and contains) assuming the hash function disperses elements properly.

Key Characteristics:

- Allows null values and one null key.
- Unordered; the order of elements is not guaranteed.

Constructors:

- `HashMap()`: Constructs an empty `HashMap` with the default initial capacity (16) and load factor (0.75).
- `HashMap(int initialCapacity)`: Constructs an empty `HashMap` with the specified initial capacity.
- `HashMap(int initialCapacity, float loadFactor)`: Constructs an empty `HashMap` with the specified initial capacity and load factor.
- `HashMap(Map<? extends K, ? extends V> m)`: Constructs a `HashMap` with the same mappings as the specified `Map`.

Methods:

- `V put(K key, V value)`: Associates the specified value with the specified key.
- `V get(Object key)`: Returns the value associated with the specified key.
- `V remove(Object key)`: Removes the mapping for the specified key if present.
- `boolean containsKey(Object key)`: Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`: Returns true if this map maps one or more keys to the specified value.
- `Set<K> keySet()`: Returns a set view of the keys contained in this map.
- `Collection<V> values()`: Returns a collection view of the values contained in this map.
- `Set<Map.Entry<K, V>> entrySet()`: Returns a set view of the mappings contained in this map.

Implementation

```
package Java_Collection_Framework;

import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>();

        // Adding key-value pairs
        map.put(1, "One");
        map.put(2, "Two");
        map.put(3, "Three");

        // Accessing values by keys
        System.out.println("Value for key 2: " + map.get(2));

        // Iterating over entries
        System.out.println("Entries in the map:");
        for (Map.Entry<Integer, String> entry : map.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ",
Value: " + entry.getValue());
        }

        // Removing a key-value pair
        map.remove(3);

        // Size of the map
        System.out.println("Size of the map: " + map.size());
    }
}
```

2. `LinkedHashMap`

`LinkedHashMap` is an implementation of the `Map` interface that maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).

Key Characteristics:

- Allows null values and one null key.
- Maintains insertion order.

Constructors:

- `LinkedHashMap()`: Constructs an empty insertion-ordered `LinkedHashMap` with the default initial capacity (16) and load factor (0.75).
- `LinkedHashMap(int initialCapacity)`: Constructs an empty, insertion-ordered `LinkedHashMap` with the specified initial capacity.
- `LinkedHashMap(int initialCapacity, float loadFactor)`: Constructs an empty, insertion-ordered `LinkedHashMap` with the specified initial capacity and load factor.
- `LinkedHashMap(Map<? extends K, ? extends V> m)`: Constructs an insertion-ordered `LinkedHashMap` with the same mappings as the specified map.
- `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`: Constructs an empty `LinkedHashMap` with the specified initial capacity, load factor, and ordering mode.

Methods:

- Inherits all methods from `HashMap` with the added benefit of maintaining insertion order.

```java
package Java_Collection_Framework;

import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashmapExample {
    public static void main(String[] args) {
        // Creating a LinkedHashMap with a capacity of 3 and a
load factor of 0.75
        LinkedHashMap<Integer, String> linkedHashMap = new
LinkedHashMap<>(3, 0.75f);

        // Adding key-value pairs
        linkedHashMap.put(3, "Three");
        linkedHashMap.put(1, "One");
        linkedHashMap.put(2, "Two");

        // Accessing values by keys
        System.out.println("Value for key 2: " +
linkedHashMap.get(2));

        // Iterating over entries
        System.out.println("Entries in the LinkedHashMap:");
        for (Map.Entry<Integer, String> entry :
linkedHashMap.entrySet()) {
            System.out.println("Key: " + entry.getKey() + ",
Value: " + entry.getValue());
        }

        // Removing a key-value pair
        linkedHashMap.remove(1);

        // Size of the LinkedHashMap
        System.out.println("Size of the LinkedHashMap: " +
linkedHashMap.size());
    }
}
```

3. `TreeMap`

`TreeMap` is a Red-Black tree-based implementation of the `Map` interface. It guarantees that the map will be in ascending key order, sorted according to the natural ordering of its keys or by a comparator provided at map creation time.

Key Characteristics:

- Does not allow null keys.
- Maintains a sorted order of keys.

Constructors:

- `TreeMap()`: Constructs an empty tree map, sorted according to the natural ordering of its keys.
- `TreeMap(Comparator<? super K> comparator)`: Constructs an empty tree map, sorted according to the specified comparator.
- `TreeMap(Map<? extends K, ? extends V> m)`: Constructs a new tree map containing the same mappings as the given map, sorted according to the natural ordering of its keys.
- `TreeMap(SortedMap<K, ? extends V> m)`: Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

Methods:

- `V put(K key, V value)`: Associates the specified value with the specified key.
- `V get(Object key)`: Returns the value associated with the specified key.
- `V remove(Object key)`: Removes the mapping for the specified key if present.
- `boolean containsKey(Object key)`: Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`: Returns true if this map maps one or more keys to the specified value.
- `K firstKey()`: Returns the first (lowest) key currently in this map.
- `K lastKey()`: Returns the last (highest) key currently in this map.
- `SortedMap<K, V> headMap(K toKey)`: Returns a view of the portion of this map whose keys are strictly less than `toKey`.
- `SortedMap<K, V> tailMap(K fromKey)`: Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
- `SortedMap<K, V> subMap(K fromKey, K toKey)`: Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.

IMPLEMENTATION OF TREE MAPS

```
      package Java_Collection_Framework;

import java.util.Map;
import java.util.TreeMap;

public class TreemmapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> treeMap = new TreeMap<>();

        // Adding key-value pairs
        treeMap.put("John", 30);
        treeMap.put("Alice", 25);
        treeMap.put("Bob", 35);

        // Accessing values by keys
        System.out.println("Age of Bob: " + treeMap.get("Bob"));

        // Iterating over entries
        System.out.println("Entries in the TreeMap:");
        for (Map.Entry<String, Integer> entry :
treeMap.entrySet()) {
            System.out.println("Name: " + entry.getKey() + ",
Age: " + entry.getValue());
        }

        // Removing a key-value pair
        treeMap.remove("Alice");

        // Size of the TreeMap
        System.out.println("Size of the TreeMap: " +
treeMap.size());
    }
}
```

4. `Hashtable`

`Hashtable` is a legacy implementation of the `Map` interface. Unlike `HashMap`, it is synchronized and does not allow null keys or values.

Key Characteristics:

- Synchronized; thread-safe.
- Does not allow null keys or values.

Constructors:

- `Hashtable()`: Constructs a new, empty hashtable with a default initial capacity (11) and load factor (0.75).
- `Hashtable(int initialCapacity)`: Constructs a new, empty hashtable with the specified initial capacity.
- `Hashtable(int initialCapacity, float loadFactor)`: Constructs a new, empty hashtable with the specified initial capacity and load factor.
- `Hashtable(Map<? extends K, ? extends V> t)`: Constructs a new hashtable with the same mappings as the given map.

Methods:

- `V put(K key, V value)`: Associates the specified value with the specified key.
- `V get(Object key)`: Returns the value associated with the specified key.
- `V remove(Object key)`: Removes the mapping for the specified key if present.
- `boolean containsKey(Object key)`: Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`: Returns true if this map maps one or more keys to the specified value.
- `boolean isEmpty()`: Returns true if this map contains no key-value mappings.
- `int size()`: Returns the number of key-value mappings in this map.
- `Enumeration<K> keys()`: Returns an enumeration of the keys in this hashtable.
- `Enumeration<V> elements()`: Returns an enumeration of the values in this hashtable.

IMPLEMENTATION OF HASHTABLES

```
package Java_Collection_Framework;

import java.util.Hashtable;
import java.util.Map;

public class HashtableExampe {
    public static void main(String[] args) {
        Hashtable<Integer, String> hashtable = new
Hashtable<>();

        // Adding key-value pairs
        hashtable.put(1, "One");
        hashtable.put(2, "Two");
        hashtable.put(3, "Three");

        // Accessing values by keys
        System.out.println("Value for key 2: " +
hashtable.get(2));

        // Iterating over entries
        System.out.println("Entries in the
Hashtable:");
        for (Map.Entry<Integer, String> entry :
hashtable.entrySet()) {
            System.out.println("Key: " + entry.getKey()
+ ", Value: " + entry.getValue());
        }

        // Removing a key-value pair
        hashtable.remove(3);

        // Size of the Hashtable
        System.out.println("Size of the Hashtable: " +
hashtable.size());
    }
}
```

5. ConcurrentHashMap

`ConcurrentHashMap` is a thread-safe implementation of the `Map` interface that allows concurrent access to the map by multiple threads without locking the entire map.

Key Characteristics:

- Thread-safe.
- High concurrency with better performance in a multi-threaded environment compared to synchronized collections.

Constructors:

- `ConcurrentHashMap()`: Constructs a new, empty map with a default initial capacity (16), load factor (0.75), and concurrency level (16).
- `ConcurrentHashMap(int initialCapacity)`: Constructs a new, empty map with the specified initial capacity, and default load factor (0.75) and concurrency level (16).
- `ConcurrentHashMap(int initialCapacity, float loadFactor)`: Constructs a new, empty map with the specified initial capacity and load factor, and default concurrency level (16).
- `ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)`: Constructs a new, empty map with the specified initial capacity, load factor, and concurrency level.
- `ConcurrentHashMap(Map<? extends K, ? extends V> m)`: Constructs a new map with the same mappings as the given map.

Methods:

- `V put(K key, V value)`: Associates the specified value with the specified key.
- `V get(Object key)`: Returns the value associated with the specified key.
- `V remove(Object key)`: Removes the mapping for the specified key if present.
- `boolean containsKey(Object key)`: Returns true if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`: Returns true if this map maps one or more keys to the specified value.
- `boolean isEmpty()`: Returns true if this map contains no key-value mappings.
- `int size()`: Returns the number of key-value mappings in this map.
- `V putIfAbsent(K key, V value)`: If the specified key is not already associated with a value, associate it with the given value.
- `boolean remove(Object key, Object value)`: Removes the entry for a key only if it is currently mapped to a given value.
- `boolean replace(K key, V oldValue, V newValue)`: Replaces the entry for a key only if currently mapped to a given value.

- `V replace(K key, V value)`: Replaces the entry for a key only if currently mapped to some value.

IMPLEMENTING CONCURRENT HASHMAPS

```
package Java_Collection_Framework;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        // Create a ConcurrentHashMap
        ConcurrentHashMap<String, Integer> concurrentMap =
new ConcurrentHashMap<>();

        // Add some initial entries
        concurrentMap.put("Apple", 10);
        concurrentMap.put("Banana", 20);
        concurrentMap.put("Cherry", 30);

        // Create a thread pool with 3 threads
        ExecutorService executor =
Executors.newFixedThreadPool(3);

        // Task 1: Update values in the map
        Runnable task1 = () -> {
            concurrentMap.put("Apple", 40);
            System.out.println("Task 1 - Updated Apple: " +
concurrentMap.get("Apple"));
        };

        // Task 2: Add a new entry to the map
        Runnable task2 = () -> {
            concurrentMap.put("Date", 50);
            System.out.println("Task 2 - Added Date: " +
concurrentMap.get("Date"));
        };

         // Task 3: Iterate over the map entries
         Runnable task3 = () -> {
             for (Map.Entry<String, Integer> entry :
concurrentMap.entrySet()) {
                 System.out.println("Task 3 - Key: "
```

```
 + entry.getKey() + ", Value: " + entry.getValue());
                }
            };

            // Execute the tasks
            executor.execute(task1);
            executor.execute(task2);
            executor.execute(task3);

            // Shutdown the executor
            executor.shutdown();
        }
    }
```

Each of these `Map` implementations provides a distinct set of features and performance trade-offs, allowing you to choose the most suitable one based on your specific requirements.
Understanding these different types of Maps and their methods will enable you to make the most out of the Java Collections Framework

1. Queue Interface

The `Queue` interface is a part of the `java.util` package and extends the `Collection` interface. It provides several methods for queue operations. Below are the primary methods:

- **`boolean add(E e)`**: Inserts the specified element into the queue. If the element is successfully added, it returns `true`; otherwise, it throws an `IllegalStateException` if the capacity restrictions are violated.
- **`boolean offer(E e)`**: Inserts the specified element into the queue. If the element is added successfully, it returns `true`; otherwise, it returns `false` if the queue is full.
- **`E remove()`**: Retrieves and removes the head of the queue. It throws `NoSuchElementException` if the queue is empty.
- **`E poll()`**: Retrieves and removes the head of the queue, or returns `null` if the queue is empty.
- **`E element()`**: Retrieves, but does not remove, the head of the queue. It throws `NoSuchElementException` if the queue is empty.
- **`E peek()`**: Retrieves, but does not remove, the head of the queue, or returns `null` if the queue is empty.

2. Implementations of Queue

Here are a few popular classes that implement the `Queue` interface:

- **`LinkedList`**: A class that implements both the `List` and `Queue` interfaces. It provides a doubly-linked list data structure.
- **`PriorityQueue`**: A class that implements a priority queue based on a priority heap.
- **`ArrayDeque`**: A class that implements a resizable array and can be used as both a queue and a deque.

3. Constructors of Queue Implementations

LinkedList Constructor

```
Queue<String> queue = new LinkedList<>();
```

- Creates an empty linked list that can be used as a queue.

PriorityQueue Constructor

```
Queue<String> queue = new PriorityQueue<>();
```

ArrayDeque Constructor

```
Queue<String> queue = new ArrayDeque<>();
```

Queue Interface Methods in Java

1. `boolean add(E e)`

- **Description**: Inserts the specified element into the queue. If the element is successfully added, it returns `true`. If the queue is full and cannot accommodate the new element, it throws an `IllegalStateException`.
- **Usage**: Use when you are sure that the queue can accept more elements.

2. **`boolean offer(E e)`**

- **Description**: Inserts the specified element into the queue. It returns `true` if the element is successfully added. If the queue is full and cannot accommodate the new element, it returns `false`.
- **Usage**: This is a safer alternative to `add(E e)`, as it does not throw an exception if the queue is full.

3. **`E remove()`**

- **Description**: Retrieves and removes the head of the queue. This method throws a `NoSuchElementException` if the queue is empty.
- **Usage**: Use when you want to retrieve and remove the head of the queue but be aware that it will throw an exception if the queue is empty.

4. **`E poll()`**

- **Description**: Retrieves and removes the head of the queue, or returns `null` if the queue is empty.
- **Usage**: A safer alternative to `remove()`, as it returns `null` instead of throwing an exception when the queue is empty.

5. **`E element()`**

- **Description**: Retrieves, but does not remove, the head of the queue. This method throws a `NoSuchElementException` if the queue is empty.
- **Usage**: Use when you need to access the head of the queue without removing it, but be aware that it throws an exception if the queue is empty.

6. **`E peek()`**

- **Description**: Retrieves, but does not remove, the head of the queue, or returns `null` if the queue is empty.
- **Usage**: A safer alternative to `element()`, as it returns `null` instead of throwing an exception when the queue is empty.

Quee Implementation

```
import java.util.Queue;
import java.util.LinkedList;
```

```java
public class QueueExample {
    public static void main(String[] args) {
        // Create a queue using LinkedList
        Queue<String> queue = new LinkedList<>();

        // Add elements to the queue
        queue.add("Element 1");
        queue.add("Element 2");
        queue.offer("Element 3");

        // Access the head of the queue
        System.out.println("Head of queue: " + queue.peek());

        // Remove elements from the queue
        System.out.println("Removed: " + queue.poll());
        System.out.println("Removed: " + queue.remove());

        // Final state of the queue
        System.out.println("Remaining elements: " + queue);
    }
}
```

PriorityQueue in Java

The `PriorityQueue` class in Java is a part of the Java Collections Framework and extends the `AbstractQueue` class. It represents a priority queue, where elements are ordered according to their natural ordering (for elements that implement `Comparable`) or by a `Comparator` provided at queue construction time.

1. `boolean add(E e)`

Description

- Inserts the specified element into the priority queue.
- Returns `true` if the element is successfully added.
- Throws `ClassCastException` if the element cannot be compared with other elements in the queue, `NullPointerException` if the specified element is `null`, and `IllegalArgumentException` if the element violates capacity restrictions.

Usage

- Use this method when adding an element to the priority queue, ensuring that the element is comparable or a comparator is provided at the time of queue creation.

2. `boolean offer(E e)`

Description

- Inserts the specified element into the priority queue.
- Returns `true` if the element is successfully added.
- Throws `ClassCastException`, `NullPointerException`, and `IllegalArgumentException` under the same conditions as `add()`.

Usage

- Similar to `add(E e)`, but `offer()` is generally used in scenarios where the queue is bounded, and there is a possibility of capacity restrictions.

3. `E remove()`

Description

- Retrieves and removes the head of the priority queue.
- Throws `NoSuchElementException` if the queue is empty.

Usage

- Use this method when you want to remove and retrieve the smallest or highest priority element in the queue, but ensure the queue is not empty.

4. `E poll()`

Description

- Retrieves and removes the head of the priority queue.
- Returns `null` if the queue is empty.

Usage

- A safer alternative to `remove()` because it returns `null` instead of throwing an exception when the queue is empty.

5. `E element()`

Description

- Retrieves, but does not remove, the head of the priority queue.
- Throws `NoSuchElementException` if the queue is empty.

Usage

- Use this method to access the head of the queue without removing it, ensuring the queue is not empty.

Priorty Qeuee Implementation

```java
import java.util.PriorityQueue;

public class PriorityQueueExample {

    public static void main(String[] args) {
        // Create a PriorityQueue of integers
        PriorityQueue<Integer> pq = new PriorityQueue<>();

        // Adding elements to the PriorityQueue
        pq.add(25);
        pq.add(5);
        pq.add(15);
        pq.add(10);
        pq.add(30);

        // Displaying the elements of the PriorityQueue
        System.out.println("PriorityQueue elements: " + pq);

        // Peek at the head of the queue
        System.out.println("Head of the queue (peek): " + pq.peek());

        // Poll elements from the PriorityQueue
        System.out.println("Polling elements from the PriorityQueue:");

        while (!pq.isEmpty()) {
            System.out.println(pq.poll());  // Retrieves and removes the head
        }

        // After polling, the PriorityQueue should be empty
        System.out.println("PriorityQueue after polling: " + pq);
    }
}
```

Code: PriorityQueue with Custom Priority Values

```
import java.util.PriorityQueue;

// Custom class with priority value
class Task implements Comparable<Task> {
    private String name;
    private int priority;

    // Constructor
    public Task(String name, int priority) {
        this.name = name;
        this.priority = priority;
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public int getPriority() {
        return priority;
    }

    // Override compareTo method to compare based on priority
    @Override
    public int compareTo(Task other) {
        // Higher priority tasks come first (lower priority value means higher
priority)
        return Integer.compare(this.priority, other.priority);
    }

    // Override toString method to display task details
    @Override
    public String toString() {
        return "Task{name='" + name + "', priority=" + priority + "}";
    }
}

public class PriorityQueueWithCustomPriority {

    public static void main(String[] args) {
        // Create a PriorityQueue of Task objects
        PriorityQueue<Task> taskQueue = new PriorityQueue<>();
```

```
        // Adding tasks to the PriorityQueue
        taskQueue.add(new Task("Task 1", 3));  // Priority 3
        taskQueue.add(new Task("Task 2", 1));  // Priority 1 (highest)
        taskQueue.add(new Task("Task 3", 4));  // Priority 4
        taskQueue.add(new Task("Task 4", 2));  // Priority 2

        // Displaying the elements of the PriorityQueue
        System.out.println("Tasks in the PriorityQueue (by priority):");

        // Poll tasks from the PriorityQueue according to priority
        while (!taskQueue.isEmpty()) {
            System.out.println(taskQueue.poll());  // Retrieves and removes the
highest priority task
        }
    }
}
```

ArrayDeque Overview

`ArrayDeque` is a part of the Java Collections Framework and implements the `Deque` interface, which allows it to function as both a stack and a queue. It uses a resizable array to store elements and provides efficient operations for adding, removing, and accessing elements from both ends of the deque (double-ended queue).

Key Features of `ArrayDeque`

- **Resizable Array**: It grows dynamically as needed.
- **Fast Operations**: Offers O(1) time complexity for most operations at both ends.
- **No Capacity Restrictions**: Unlike `ArrayBlockingQueue`, it doesn't have a fixed capacity.
- **Null Elements**: It does not allow `null` elements.

Common Methods of `ArrayDeque`

Methods for Adding Elements

- **`addFirst(E e)`**: Inserts the specified element at the front of the deque.
- **`addLast(E e)`**: Inserts the specified element at the end of the deque.
- **`offerFirst(E e)`**: Inserts the specified element at the front of the deque, returning `true` if successful.
- **`offerLast(E e)`**: Inserts the specified element at the end of the deque, returning `true` if successful.

Methods for Removing Elements

- **`removeFirst()`**: Removes and returns the first element of the deque.
- **`removeLast()`**: Removes and returns the last element of the deque.
- **`pollFirst()`**: Removes and returns the first element of the deque, or returns `null` if the deque is empty.
- **`pollLast()`**: Removes and returns the last element of the deque, or returns `null` if the deque is empty.

Methods for Peeking at Elements

- **`getFirst()`**: Retrieves, but does not remove, the first element of the deque.
- **`getLast()`**: Retrieves, but does not remove, the last element of the deque.
- **`peekFirst()`**: Retrieves, but does not remove, the first element of the deque, or returns `null` if the deque is empty.
- **`peekLast()`**: Retrieves, but does not remove, the last element of the deque, or returns `null` if the deque is empty.

Other Methods

- **`size()`**: Returns the number of elements in the deque.
- **`isEmpty()`**: Checks if the deque is empty.
- **`clear()`**: Removes all elements from the deque.
- **`contains(Object o)`**: Checks if the deque contains the specified element.
- **`toArray()`**: Returns an array containing all elements in the deque.
- **`iterator()`**: Returns an iterator over the elements in the deque.

Implemenataion of ArrayDeque

```
import java.util.ArrayDeque;

public class ArrayDequeExample {

    public static void main(String[] args) {
        // Create an ArrayDeque
        ArrayDeque<String> deque = new ArrayDeque<>();

        // Adding elements to the deque
        deque.addFirst("First");
        deque.addLast("Last");
        deque.offerFirst("New First");
        deque.offerLast("New Last");

        // Display the elements
        System.out.println("Deque elements: " + deque);

        // Peeking at elements
        System.out.println("First element (peekFirst): " + deque.peekFirst());
        System.out.println("Last element (peekLast): " + deque.peekLast());

        // Removing elements
        System.out.println("Removed first element (removeFirst): " +
deque.removeFirst());
        System.out.println("Removed last element (removeLast): " +
deque.removeLast());

        // Display the elements after removal
        System.out.println("Deque elements after removals: " + deque);

        // Checking size and emptiness
        System.out.println("Size of deque: " + deque.size());
        System.out.println("Is deque empty? " + deque.isEmpty());

        // Clearing the deque
        deque.clear();
```

```
        System.out.println("Deque elements after clearing: " + deque);
    }
}
```

Java Generics

Generics in Java provide a way to create classes, interfaces, and methods that operate on types specified by the programmer. They allow for type safety and reusability by allowing you to define classes, interfaces, and methods with placeholder types that are replaced with concrete types when the class or method is instantiated or called.

Key Concepts of Generics

1. **Type Parameters**:
    - Generics use type parameters, denoted by angle brackets (<>), to specify types. For example, `List<T>` uses `T` as a placeholder for the type that will be specified later.
2. **Type Safety**:
    - Generics provide compile-time type checking, ensuring that only the correct type of objects are used. This helps prevent `ClassCastException` at runtime.
3. **Reusability**:
    - You can write generic code that works with any data type, allowing you to reuse code without duplication.

Basic Syntax

1. Generic Class

A generic class is defined with a type parameter that can be used throughout the class.

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

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

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

- `T` is a type parameter. It represents the type of object the class will operate on.

2. Generic Method

A generic method can be defined with its own type parameter.

```
public class Utils {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.println(element);
        }
    }
}
```

- **`<T>`** before the return type indicates that the method is generic.

3. Generic Interface

A generic interface defines a contract with type parameters.

```
public interface Repository<T> {
    void add(T item);
    T get(int index);
}
```

Example Usage

Using a Generic Class

```
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.set(123);
        System.out.println("Integer value: " + integerBox.get());

        Box<String> stringBox = new Box<>();
        stringBox.set("Hello Generics");
        System.out.println("String value: " + stringBox.get());
    }
}
```

- **`Box<Integer>`** creates a box for integers.
- **`Box<String>`** creates a box for strings.

Using a Generic Method

```
public class Main {
    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3};
        String[] strArray = {"A", "B", "C"};

        Utils.printArray(intArray);
        Utils.printArray(strArray);
    }
}
```

Using a Generic Interface

```
public class ArrayRepository<T> implements Repository<T> {
    private List<T> items = new ArrayList<>();

    @Override
    public void add(T item) {
        items.add(item);
    }

    @Override
    public T get(int index) {
        return items.get(index);
    }
}

public class Main {
    public static void main(String[] args) {
        Repository<String> repo = new ArrayRepository<>();
        repo.add("Item 1");
        repo.add("Item 2");
        System.out.println(repo.get(0));
        System.out.println(repo.get(1));
    }
}
```

Wildcards in Generics

Wildcards are used when you want to specify a range of possible types.

- `?`: Represents an unknown type.

```
public void printList(List<?> list) {
    for (Object item : list) {
        System.out.println(item);
    }
}
```

`<? extends T>`: Represents a type that is a subclass of `T`.

```
public void processNumbers(List<? extends Number> numbers) {
    for (Number number : numbers) {
        System.out.println(number);
    }
}
```

`<? super T>`: Represents a type that is a superclass of `T`.

```
public void addNumbers(List<? super Integer> list) {
    list.add(10);
    list.add(20);
}
```

Collection Interface

The `Collections` interface in Java is a root interface in the Java Collections Framework. It is part of the `java.util` package and provides a unified view of different collections, but it is not intended to be implemented directly. Instead, it is implemented by various classes like `List`, `Set`, and `Queue`.

Key Characteristics

- **Root of Collection Hierarchy**: The `Collections` interface is at the top of the hierarchy, and all collection classes (`List`, `Set`, `Queue`) extend it.
- **Type Parameter**: It uses generics to handle various types of objects.

Major Subinterfaces

- **List**: Ordered collection that allows duplicate elements. Implementations include `ArrayList`, `LinkedList`, and `Vector`.
- **Set**: Collection that does not allow duplicate elements. Implementations include `HashSet`, `LinkedHashSet`, and `TreeSet`.
- **Queue**: Collection used for holding elements prior to processing. Implementations include `PriorityQueue`, `ArrayDeque`, and `LinkedList`.

Core Methods of `Collection` Interface

Here's a summary of the key methods defined in the `Collection` interface:

1. Adding Elements

- **`boolean add(E e)`**: Adds the specified element to the collection.
- **`boolean addAll(Collection<? extends E> c)`**: Adds all elements from the specified collection to this collection.

2. Removing Elements

- **`boolean remove(Object o)`**: Removes a single instance of the specified element from this collection.
- **`boolean removeAll(Collection<?> c)`**: Removes all elements in this collection that are also contained in the specified collection.
- **`boolean retainAll(Collection<?> c)`**: Retains only the elements in this collection that are contained in the specified collection.
- **`void clear()`**: Removes all elements from this collection.

3. Querying Elements

- **`boolean contains(Object o)`**: Checks if the collection contains the specified element.

- **`boolean containsAll(Collection<?> c)`**: Checks if the collection contains all elements of the specified collection.
- **`int size()`**: Returns the number of elements in the collection.
- **`boolean isEmpty()`**: Checks if the collection is empty.

4. Iterating Over Elements

- **`Iterator<E> iterator()`**: Returns an iterator over the elements in the collection.
- **`Object[] toArray()`**: Returns an array containing all elements in the collection.
- **`<T> T[] toArray(T[] a)`**: Returns an array containing all elements in the collection, with the runtime type of the returned array.

Example Code

Here's an example demonstrating some of the `Collection` methods:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionExample {

    public static void main(String[] args) {
        // Create a Collection (using ArrayList as an example)
        Collection<String> collection = new ArrayList<>();

        // Add elements
        collection.add("Apple");
        collection.add("Banana");
        collection.add("Cherry");

        // Print the collection
        System.out.println("Collection: " + collection);

        // Check size
        System.out.println("Size: " + collection.size());

        // Check if it contains a specific element
        System.out.println("Contains Banana? " + collection.contains("Banana"));

        // Remove an element
        collection.remove("Banana");
        System.out.println("Collection after removal: " + collection);

        // Iterate over elements
```

```
        System.out.println("Iterating over collection:");
        Iterator<String> iterator = collection.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }

        // Convert to array
        String[] array = collection.toArray(new String[0]);
        System.out.println("Array: ");
        for (String item : array) {
            System.out.println(item);
        }

        // Clear the collection
        collection.clear();
        System.out.println("Collection after clearing: " + collection);
    }
}
```

Java Collections Class

The `Collections` class in Java is a utility class that provides a range of static methods to operate on or return collections. It is part of the `java.util` package and is designed to offer common algorithms and operations for collections such as sorting, searching, and manipulation. The `Collections` class itself does not implement the `Collection` interface but provides static methods that work with collections.

Key Features

1. **Utility Methods**: Contains static methods for various operations on collections.
2. **Immutable Collections**: Provides methods to create immutable collections.
3. **Algorithm Implementations**: Implements common algorithms like sorting and searching.

Main Methods of `Collections` Class

1. Sorting

- **`public static <T extends Comparable<? super T>> void sort(List<T> list)`**
  - Sorts the specified list into ascending order according to the natural ordering of its elements.
- **`public static <T> void sort(List<T> list, Comparator<? super T> c)`**
  - Sorts the specified list according to the order induced by the specified comparator.

2. Searching

- **`public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`**
  - Searches the specified list for the specified object using binary search.
- **`public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`**
  - Searches the specified list for the specified object using binary search and a comparator.

3. Shuffling

- **`public static void shuffle(List<?> list)`**
  - Randomly permutes the elements in the specified list.
- **`public static void shuffle(List<?> list, Random rnd)`**
  - Randomly permutes the elements in the specified list using the specified source of randomness.

4. Reversing

- **`public static void reverse(List<?> list)`**

- Reverses the order of the elements in the specified list.

5. Rotating

- **`public static void rotate(List<?> list, int distance)`**
  - Rotates the elements in the specified list by the specified distance.

6. Swapping

- **`public static void swap(List<?> list, int i, int j)`**
  - Swaps the elements at the specified positions in the specified list.

7. Unmodifiable Collections

- **`public static <T> List<T> unmodifiableList(List<? extends T> list)`**
  - Returns an unmodifiable view of the specified list.
- **`public static <T> Set<T> unmodifiableSet(Set<? extends T> s)`**
  - Returns an unmodifiable view of the specified set.
- **`public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m)`**
  - Returns an unmodifiable view of the specified map.

8. Empty and Singleton Collections

- **`public static <T> Set<T> emptySet()`**
  - Returns an empty set (immutable).
- **`public static <T> List<T> emptyList()`**
  - Returns an empty list (immutable).
- **`public static <T> Map<T, T> emptyMap()`**
  - Returns an empty map (immutable).
- **`public static <T> Set<T> singleton(T o)`**
  - Returns an immutable set containing only the specified object.
- **`public static <T> List<T> singletonList(T o)`**
  - Returns an immutable list containing only the specified object.
- **`public static <K,V> Map<K,V> singletonMap(K key, V value)`**
  - Returns an immutable map with a single key-value pair.

Example Usage

Here's a brief example demonstrating some of the methods from the `Collections` class:

```
import java.util.*;

public class CollectionsExample {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("Banana", "Apple",
"Cherry", "Date"));

        // Sort the list
        Collections.sort(list);
        System.out.println("Sorted list: " + list);

        // Shuffle the list
        Collections.shuffle(list);
        System.out.println("Shuffled list: " + list);

        // Reverse the list
        Collections.reverse(list);
        System.out.println("Reversed list: " + list);

        // Rotate the list
        Collections.rotate(list, 2);
        System.out.println("Rotated list: " + list);

        // Find index of an element
        int index = Collections.binarySearch(list, "Apple");
        System.out.println("Index of 'Apple': " + index);

        // Create an unmodifiable list
        List<String> unmodifiableList = Collections.unmodifiableList(new
ArrayList<>(list));
        System.out.println("Unmodifiable list: " + unmodifiableList);

        // Attempting to modify the unmodifiable list will throw
UnsupportedOperationException
        // unmodifiableList.add("Orange"); // Uncommenting this will throw an
exception
    }
}
```

`Iterator` Interface

The `Iterator` interface is part of the `java.util` package and provides a way to traverse the elements in a collection, one element at a time. It is a core part of the Java Collections Framework and is often used in loops to iterate through collections such as `List`, `Set`, and `Map`.

Key Methods of `Iterator`

- **`boolean hasNext()`**: Returns `true` if the iteration has more elements.
- **`E next()`**: Returns the next element in the iteration.
- **`void remove()`**: Removes from the underlying collection the last element returned by this iterator (optional operation).

Implementation of Iterator

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class IteratorExample {

    public static void main(String[] args) {
        // Create an ArrayList of Strings
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Date");
        fruits.add("Elderberry");

        // 1. Iterating through the ArrayList using Iterator
        System.out.println("Iterating over ArrayList elements:");
        Iterator<String> iterator = fruits.iterator();
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            System.out.println(fruit);
        }

        // 2. Removing an element conditionally using Iterator
        System.out.println("\nRemoving elements that start with 'A':");
        iterator = fruits.iterator();  // Re-initializing the iterator
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            if (fruit.startsWith("A")) {
                iterator.remove();  // Removes "Apple"
```

```
            }
        }
        System.out.println("ArrayList after removal: " + fruits);

        // 3. Adding elements back to demonstrate another removal
        fruits.add("Apricot");

        // 4. Removing all elements that start with a vowel using Iterator
        System.out.println("\nRemoving elements that start with a vowel:");
        iterator = fruits.iterator();  // Re-initializing the iterator
        while (iterator.hasNext()) {
            String fruit = iterator.next();
            if (startsWithVowel(fruit)) {
                iterator.remove();  // Removes "Apricot", "Elderberry"
            }
        }
        System.out.println("ArrayList after vowel removal: " + fruits);

        // 5. Displaying the final state of the ArrayList
        System.out.println("\nFinal ArrayList elements:");
        iterator = fruits.iterator();  // Re-initializing the iterator
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }

    // Helper method to check if a string starts with a vowel
    private static boolean startsWithVowel(String str) {
        return str.toLowerCase().matches("^[aeiou].*");
    }
}
```

`Stream` Interface

The `Stream` interface is part of the `java.util.stream` package and represents a sequence of elements supporting sequential and parallel aggregate operations. Streams are not a data structure but a higher-level abstraction for processing sequences of elements.

Streams in Java 8 and later provide a functional programming approach to processing collections of objects. They allow operations such as filtering, mapping, and reducing.

Key Methods of `Stream`

- **`filter(Predicate<? super T> predicate)`**: Returns a stream consisting of the elements of this stream that match the given predicate.
- **`map(Function<? super T, ? extends R> mapper)`**: Returns a stream consisting of the results of applying the given function to the elements of this stream.
- **`collect(Collector<? super T, A, R> collector)`**: Performs a mutable reduction operation on the elements of this stream using a `Collector`.
- **`forEach(Consumer<? super T> action)`**: Performs an action for each element of this stream.
- **`reduce(BinaryOperator<T> accumulator)`**: Performs a reduction on the elements of this stream, using an associative accumulation function, and returns an `Optional`.

Implemenation of Stream()

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {

    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        fruits.add("Apricot");

        // Filtering and collecting the results using Stream
        List<String> filteredFruits = fruits.stream()
                                            .filter(fruit ->
fruit.startsWith("A"))
                                            .collect(Collectors.toList());

        System.out.println("Filtered fruits: " + filteredFruits);
```

```
        // Mapping and collecting the results using Stream
        List<String> upperCaseFruits = fruits.stream()
                                             .map(String::toUpperCase)
                                             .collect(Collectors.toList());

        System.out.println("Uppercase fruits: " + upperCaseFruits);

        // Finding max using Stream
        String maxFruit = fruits.stream()
                                .max(String::compareTo)
                                .orElse("No fruits");

        System.out.println("Max fruit: " + maxFruit);
    }
}
```

Lambda Functions

Java Lambda Functions, introduced in Java 8, are a key feature that brings functional programming capabilities to Java. A lambda expression is essentially a short block of code which takes in parameters and returns a value. Lambda expressions are a clear and concise way to represent one method interface using an expression.

Key Features of Lambda Functions

1. **Conciseness**: They allow you to express instances of single-method interfaces (functional interfaces) more compactly.
2. **Readability**: They can make your code more readable, especially when used with functional interfaces like `Runnable`, `Comparator`, and others.
3. **Functional Programming**: Lambdas bring functional programming constructs to Java, making it easier to work with collections, streams, and asynchronous programming.

Syntax of Lambda Expressions

The syntax of a lambda expression in Java is:

```
(parameters) -> expression
```

Or, if the body contains more than one statement:

```
(parameters) -> { statements; }
```

Example Syntax Breakdown

- **No Parameters**: `() -> System.out.println("Hello World");`
- **One Parameter**: `x -> x * x`
- **Multiple Parameters**: `(a, b) -> a + b`
- **With Body**: `(int x, int y) -> { return x + y; }`

Functional Interfaces

Lambda expressions can be used only with functional interfaces. A functional interface is an interface that contains exactly one abstract method. The `@FunctionalInterface` annotation is optional but it's a good practice to use it to make the intention clear.

Example of a Functional Interface:

```
@FunctionalInterface
interface MyFunctionalInterface {
    void myMethod();
}
```

Example Usage of Lambda Expressions

1. Using Lambda with Runnable

```
public class LambdaExample {

    public static void main(String[] args) {
        // Traditional way using an anonymous class
        Runnable r1 = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello from Runnable");
            }
        };

        // Using Lambda Expression
        Runnable r2 = () -> System.out.println("Hello from Lambda");

        r1.run();
        r2.run();
    }
}
```

2. Using Lambda with Comparator

```
import java.util.Arrays;
import java.util.Comparator;

public class ComparatorLambdaExample {

    public static void main(String[] args) {
        String[] names = {"Steve", "Anna", "Mike", "Xenia"};

        // Traditional way using an anonymous class
        Arrays.sort(names, new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
```

```
                return s1.compareTo(s2);
            }
        });

        System.out.println("Sorted using anonymous class: " +
Arrays.toString(names));

        // Using Lambda Expression
        Arrays.sort(names, (s1, s2) -> s1.compareTo(s2));

        System.out.println("Sorted using Lambda: " + Arrays.toString(names));
    }
}
```

4. **Using Lambda with List and Streams**

```
import java.util.ArrayList;
import java.util.List;

public class StreamLambdaExample {

    public static void main(String[] args) {
        List<String> names = new ArrayList<>();
        names.add("Steve");
        names.add("Anna");
        names.add("Mike");
        names.add("Xenia");

        // Using forEach with Lambda
        names.forEach(name -> System.out.println(name));

        // Using Stream and Lambda for filtering and sorting
        names.stream()
             .filter(name -> name.startsWith("S"))
             .sorted()
             .forEach(System.out::println);
    }
}
```

Benefits of Using Lambda Expressions

1. **Code Reduction**: Lambdas can significantly reduce the amount of boilerplate code, especially when creating simple instances of functional interfaces.
2. **Improved Readability**: The code becomes more readable and expressive.
3. **Functional Programming**: They make it easier to apply functional programming techniques, especially when working with Streams API.
4. **Parallel Processing**: Lambdas, when used with the Streams API, can be easily used to parallelize operations, improving performance.

Important Notes

- **Scope and Variable Capture**: Lambda expressions can capture variables from their enclosing scope (both `final` and effectively `final` variables).
- **Type Inference**: The compiler can often infer the types of the parameters, making the syntax more concise.
- **Functional Interface Requirement**: A lambda can only be used where a functional interface is expected (an interface with a single abstract method).

Conclusion

Java Lambda Expressions offer a powerful way to write concise and readable code, especially when used with the Streams API or when creating instances of functional interfaces. They make Java a more versatile language by bringing in functional programming features, which are especially useful for processing collections, implementing callbacks, and handling concurrency.