Introduction

April 18, 2024

1 Java Collections

In Java, both interfaces and classes are two different types of entities that allow you to define fields and methods. However, they have some key differences:

1. **Definition:**
   - **Interface:** An interface is a reference type in Java, similar to a class, that is defined by the keyword `interface`. It is a collection of abstract methods (methods without bodies) and static constants. In Java 8 and later, interfaces can contain default methods (methods with bodies) and static methods.
   - **Class:** A class is a blueprint or template for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member methods or functions).
2. **Implementation:**
   - **Interface:** An interface cannot be instantiated. It needs a class for the implementation of its methods. A class can implement multiple interfaces.
   - **Class:** A class can be instantiated. The `new` keyword followed by the class name is used to create an instance of the class.
3. **Inheritance:**
   - **Interface:** An interface can extend multiple interfaces but cannot extend a class.
   - **Class:** A class can extend only one class but can implement multiple interfaces.
4. **Access Modifiers:**
   - **Interface:** In an interface, all variables are automatically `public`, `static`, and `final` and all methods are `public`.
   - **Class:** A class can have any access modifier. It can contain `public`, `private`, `protected` or default variables and methods.

Here's an example of an interface and a class in Java:

[ ]:
```java
// Interface
public interface Animal {
    void eat();
    void sleep();
}

// Class
public class Dog implements Animal {
    @Override
    public void eat() {
```

```
        System.out.println("Dog eats");
    }

    @Override
    public void sleep() {
        System.out.println("Dog sleeps");
    }
}
```

In this example, `Dog` is a class that implements the `Animal` interface. The `Dog` class provides the implementation for the `eat` and `sleep` methods.

2 What is Java Collections?

The Java Collections Framework is a set of classes and interfaces that implement commonly reusable collection data structures. It is located in the `java.util` package and includes interfaces like `List`, `Set`, `Queue`, `Deque`, and `Map`, and classes like `ArrayList`, `LinkedList`, `HashSet`, `LinkedHashSet`, `HashMap`, and `TreeMap`, among others.

The Collections Framework provides several benefits:

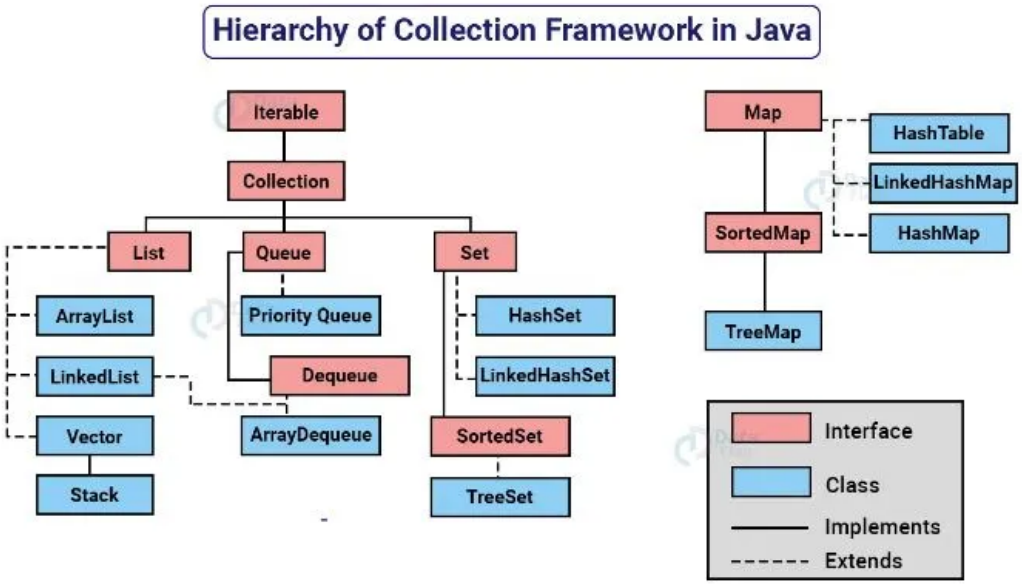
1. **Reduced Development Effort:** By using core collection classes provided by Java, developers can focus on the important parts of their application rather than writing data structures from scratch.
2. **Quality:** The provided classes are high-quality, industry-tested classes that have been optimized for performance.
3. **Consistency:** All collections have a unified architecture, which makes them easy to use and understand.
4. **Interoperability:** As many methods (like sorting and binary search) are polymorphic (that is, the same method can be used on many different implementations of a similar interface), different collections can work in a similar manner.
5. **Extensibility:** The abstract classes and interfaces in the Collections Framework allow you to make your own collections.

For more details, you can refer to the Introduction.ipynb file in your workspace.

3 Hierarchy of Collection Framework

[ ]:
```
from IPython.display import Image
Image("JC.jpg")
```

[ ]:



4 What is Difference between Interface and Class?

In Java, both interfaces and classes are two different types of entities that allow you to define fields and methods. However, they have some key differences:

1. **Definition:**
   - **Interface:** An interface is a reference type in Java, similar to a class, that is defined by the keyword `interface`. It is a collection of abstract methods (methods without bodies) and static constants. In Java 8 and later, interfaces can contain default methods (methods with bodies) and static methods.
   - **Class:** A class is a blueprint or template for creating objects (a particular data structure), providing initial values for state (member variables or attributes), and implementations of behavior (member methods or functions).
2. **Implementation:**
   - **Interface:** An interface cannot be instantiated. It needs a class for the implementation of its methods. A class can implement multiple interfaces.
   - **Class:** A class can be instantiated. The `new` keyword followed by the class name is used to create an instance of the class.
3. **Inheritance:**
   - **Interface:** An interface can extend multiple interfaces but cannot extend a class.
   - **Class:** A class can extend only one class but can implement multiple interfaces.
4. **Access Modifiers:**
   - **Interface:** In an interface, all variables are automatically `public`, `static`, and `final` and all methods are `public`.
   - **Class:** A class can have any access modifier. It can contain `public`, `private`, `protected` or default variables and methods.

Here's an example of an interface and a class in Java:

[ ]:
```java
// Interface
public interface Animal {
    void eat();
    void sleep();
}

// Class
public class Dog implements Animal {
    @Override
    public void eat() {
        System.out.println("Dog eats");
    }

    @Override
    public void sleep() {
        System.out.println("Dog sleeps");
    }
}
```

In this example, `Dog` is a class that implements the `Animal` interface. The `Dog` class provides the implementation for the `eat` and `sleep` methods.

5 Explains Methods of Collection interface in details?

The `Collection` interface in Java is the root interface in the collection hierarchy. Here are the key methods defined in the `Collection` interface:

1. **`boolean add(E e)`:** This method ensures that this collection contains the specified element. Returns `true` if this collection changed as a result of the call.
2. **`boolean addAll(Collection<? extends E> c)`:** This method adds all of the elements in the specified collection to this collection. Returns `true` if this collection changed as a result of the call.
3. **`void clear()`:** This method removes all of the elements from this collection.
4. **`boolean contains(Object o)`:** This method returns `true` if this collection contains the specified element.
5. **`boolean containsAll(Collection<?> c)`:** This method returns `true` if this collection contains all of the elements in the specified collection.
6. **`boolean equals(Object o)`:** This method compares the specified object with this collection for equality.
7. **`int hashCode()`:** This method returns the hash code value for this collection.
8. **`boolean isEmpty()`:** This method returns `true` if this collection contains no elements.

9. **`Iterator<E> iterator()`:** This method returns an iterator over the elements in this collection.
10. **`boolean remove(Object o)`:** This method removes a single instance of the specified element from this collection, if it is present.
11. **`boolean removeAll(Collection<?> c)`:** This method removes all of this collection's elements that are also contained in the specified collection.
12. **`boolean retainAll(Collection<?> c)`:** This method retains only the elements in this collection that are contained in the specified collection.
13. **`int size()`:** This method returns the number of elements in this collection.
14. **`Object[] toArray()`:** This method returns an array containing all of the elements in this collection.
15. **`<T> T[] toArray(T[] a)`:** This method returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

These methods provide the basic operations that are required for a collection, such as adding elements, removing elements, checking if an element exists, and iterating over elements.

6 Method in Iterator interface?

The `Iterator` interface in Java provides methods to iterate over any type of collection. It has three main methods:

1. **`boolean hasNext()`:** This method returns `true` if the iteration has more elements. It can be used to check if there are more elements to iterate over.
2. **`E next()`:** This method returns the next element in the iteration. It throws `NoSuchElementException` if the iteration has no more elements.
3. **`void remove()`:** This method removes from the underlying collection the last element returned by this iterator. This method can be called only once per call to `next()`. It throws `IllegalStateException` if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method.

Here's an example of using an Iterator:

[ ]:
```
List<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Cherry");

Iterator<String> iterator = list.iterator();

while(iterator.hasNext()) {
    String fruit = iterator.next();
    System.out.println(fruit);
}
```

```
Apple
Banana
Cherry
```

In this example, the `hasNext` method is used to check if there are more elements in the list, and the `next` method is used to get each element. The `remove` method is not used in this example, but if it were called after `next`, it would remove the current element from the list.

7 Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

8 List Interface

The `List` interface in Java is part of the Java Collections Framework and extends the `Collection` interface. It represents an ordered collection of elements, also known as a sequence. The user of a `List` has precise control over where in the list each element is inserted and can access elements by their integer index (position in the list).

Here are some key methods provided by the `List` interface (in addition to those methods inherited from `Collection`):

1. **`void add(int index, E element):`** Inserts the specified element at the specified position in this list.
2. **`boolean addAll(int index, Collection<? extends E> c):`** Inserts all of the elements in the specified collection into this list at the specified position.
3. **`E get(int index):`** Returns the element at the specified position in this list.
4. **`int indexOf(Object o):`** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
5. **`int lastIndexOf(Object o):`** Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
6. **`ListIterator<E> listIterator():`** Returns a list iterator over the elements in this list (in proper sequence).
7. **`E remove(int index):`** Removes the element at the specified position in this list.
8. **`E set(int index, E element):`** Replaces the element at the specified position in this list with the specified element.
9. **`List<E> subList(int fromIndex, int toIndex):`** Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.

Some of the commonly used classes that implement the `List` interface are `ArrayList`, `LinkedList`, and `Vector`. Each of these classes has different performance characteristics, with `ArrayList` being the most commonly used due to its general-purpose design.

8.1 ArrayList

`ArrayList` is a resizable array implementation of the `List` interface in Java. It implements all optional list operations and permits all elements, including `null`. It provides methods to manipulate the size of the array that is used internally to store the list.

Here are some key characteristics and methods of `ArrayList`:

1. **Resizable Array:** Unlike an array, an `ArrayList` can dynamically resize itself when elements are added or removed.
2. **Ordering:** `ArrayList` maintains the insertion order of elements, meaning you can access elements in the order they were inserted.
3. **Null Elements:** `ArrayList` allows null and duplicate elements.
4. **Random Access:** `ArrayList` supports fast random access of elements because it implements the `RandomAccess` interface. You can get any element from an array list in constant time.
5. **Not Synchronized:** `ArrayList` is not synchronized, which means it is not thread-safe. If multiple threads access and modify an `ArrayList` concurrently, it must be synchronized externally.

Key methods of `ArrayList` include all methods from the `List` interface, and some methods inherited from `AbstractList`, `AbstractCollection`, and `Object`. Here are a few:

- **`void ensureCapacity(int minCapacity)`:** Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
- **`void trimToSize()`:** Trims the capacity of this `ArrayList` instance to be the list's current size.

Here's an example of using an `ArrayList`:

[ ]:
```
ArrayList<String> list = new ArrayList<>();
list.add("Apple");
list.add("Banana");
list.add("Cherry");
System.out.println(list.get(1)); // Outputs: Banana
```

```
Banana
```

In this example, an `ArrayList` of `String` objects is created, and three elements are added to it. The `get` method is then used to retrieve the second element (index 1) from the list.

9 LinkedList

`LinkedList` is a class in Java that implements the `List` and `Deque` interfaces. It uses a doubly-linked list internally to store its elements. This means that each element in the `LinkedList` has a reference to the element that comes before it and the element that comes after it.

Here are some key characteristics and methods of `LinkedList`:

1. **Doubly-Linked List:** Each element (or node) contains a reference to the preceding node and the following node, which allows for an efficient insertion or removal of elements from any position in the list.
2. **Ordering:** Like `ArrayList`, `LinkedList` also maintains the insertion order of elements.
3. **Null Elements:** `LinkedList` allows null and duplicate elements.
4. **Not Synchronized:** `LinkedList` is not synchronized, which means it is not thread-safe. If multiple threads access and modify a `LinkedList` concurrently, it must be synchronized externally.

Key methods of `LinkedList` include all methods from the `List` and `Deque` interfaces, and some methods inherited from `AbstractSequentialList`, `AbstractList`, `AbstractCollection`, and `Object`. Here are a few:

- **`void addFirst(E e):`** Inserts the specified element at the beginning of this list.
- **`void addLast(E e):`** Appends the specified element to the end of this list.
- **`E getFirst():`** Returns the first element in this list.
- **`E getLast():`** Returns the last element in this list.
- **`E removeFirst():`** Removes and returns the first element from this list.
- **`E removeLast():`** Removes and returns the last element from this list.

Here's an example of using a `LinkedList`:

[ ]:
```
LinkedList<String> list = new LinkedList<>();
list.add("Apple");
list.add("Banana");
list.add("Cherry");
System.out.println(list.getFirst()); // Outputs: Apple
```

In this example, a `LinkedList` of `String` objects is created, and three elements are added to it. The `getFirst` method is then used to retrieve the first element from the list.

10 What is the difference between ArrayList and LinkedList in Java?

`ArrayList` and `LinkedList` are two commonly used classes in Java that implement the `List` interface. Here are some key differences between them:

1. **Internal Data Structure:**

- `ArrayList` uses a dynamic array to store its elements. This allows for fast random access of elements as you can directly access any element with its index.
- `LinkedList` uses a doubly-linked list internally. Each element (or node) contains a reference to the preceding node and the following node. This allows for efficient insertion and removal at both ends, but accessing elements in the middle can be slower.

2. **Performance:**
   - `ArrayList` is generally faster for lookup operations (`get` and `set` methods) because it allows direct access to any element via the index.
   - `LinkedList` is generally faster for add and remove operations at the start or end (`addFirst`, `removeFirst`, `addLast`, `removeLast` methods) because it only needs to update a couple of references. However, adding or removing from the middle of the `LinkedList` requires traversing from the head or tail to the insertion or removal point.
3. **Memory Overhead:**
   - `ArrayList` is more memory-efficient (i.e., it uses less memory) because each element in the `ArrayList` holds only the data (or reference to the data).
   - `LinkedList` uses more memory because each element in the `LinkedList` needs to store the data and two references for neighbor nodes.
4. **Use Cases:**
   - `ArrayList` is a better choice when you have more get/set operations, and you know the index of elements.
   - `LinkedList` is a better choice when you have more add/remove operations, and these operations are more at the beginning or end of the list.

Remember, the choice between `ArrayList` and `LinkedList` depends on the specific requirements of your use case.

10.1 Vector

`Vector` is a class in Java that implements the `List` interface. It is very similar to `ArrayList`, but with two key differences: `Vector` is synchronized, and it contains many legacy methods that are not part of the collections framework.

Here are some key characteristics and methods of `Vector`:

1. **Resizable Array:** Like `ArrayList`, `Vector` also uses a dynamic array for storing the data, which can grow as needed.
2. **Ordering:** `Vector` maintains the insertion order of elements.
3. **Null Elements:** `Vector` allows null and duplicate elements.
4. **Synchronized:** Unlike `ArrayList`, `Vector` is synchronized. This means it is thread-safe. Multiple threads can access and modify a `Vector` without risking data inconsistency.

Key methods of `Vector` include all methods from the `List` interface, and some methods inherited from `AbstractList`, `AbstractCollection`, and `Object`. It also includes some legacy methods that existed before the Collection framework. Here are a few:

- **`void addElement(E obj)`:** Adds the specified component to the end of this vector, increasing its size by one.

- **`boolean removeElement(Object obj):`** Removes the first (lowest-indexed) occurrence of the argument from this vector.
- **`void insertElementAt(E obj, int index):`** Inserts the specified object as a component in this vector at the specified `index`.
- **`E firstElement():`** Returns the first component (the item at index `0`) of this vector.
- **`E lastElement():`** Returns the last component of the vector.

Here's an example of using a `Vector`:

[ ]:
```
Vector<String> vector = new Vector<>();
vector.addElement("Apple");
vector.addElement("Banana");
vector.addElement("Cherry");
System.out.println(vector.firstElement()); // Outputs: Apple
```

```
Apple
```

In this example, a `Vector` of `String` objects is created, and three elements are added to it. The `firstElement` method is then used to retrieve the first element from the vector.

10.2 Stack

`Stack` is a class in Java that extends `Vector` and represents a last-in-first-out (LIFO) stack of objects. It allows you to have direct control over where an element is inserted or removed from the stack.

Here are some key characteristics and methods of `Stack`:

1. **LIFO Structure:** The `Stack` class represents a last-in-first-out (LIFO) stack of objects. The element pushed last is the first one to come off the stack.
2. **Extends Vector:** `Stack` is a subclass of `Vector`, and thus, it is synchronized and can grow as needed.

Key methods of `Stack` include:

- **`E push(E item):`** Pushes an item onto the top of this stack.
- **`E pop():`** Removes the object at the top of this stack and returns that object as the value of this function.
- **`E peek():`** Looks at the object at the top of this stack without removing it from the stack.
- **`int search(Object o):`** Returns the 1-based position where an object is on this stack.

Here's an example of using a `Stack`:

[ ]:
```
Stack<String> stack = new Stack<>();
stack.push("Apple");
stack.push("Banana");
stack.push("Cherry");
System.out.println(stack.pop()); // Outputs: Cherry
```

```
Cherry
```

In this example, a `Stack` of `String` objects is created, and three elements are pushed onto it. The `pop` method is then used to remove and return the top element from the stack.

11 Queue Interface

The `Queue` interface in Java is part of the Java Collections Framework and extends the `Collection` interface. It is used to hold the elements about to be processed and provides various operations like the insertion, removal, etc. It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of the list, (i.e., it follows the FIFO or the First-In-First-Out principle).

Here are some key methods provided by the `Queue` interface:

1. **`boolean add(E e)`:** This method is used to add elements at the tail of the queue. It returns true if the element is added successfully, otherwise it throws an exception.
2. **`E element()`:** It is used to retrieves, but does not remove, the head of this queue. It throws an exception if the queue is empty.
3. **`boolean offer(E e)`:** This method is used to insert the specified element into this queue. It returns true if the element was added to this queue, else false.
4. **`E remove()`:** This method is used to retrieve and remove the head of this queue. It throws an exception if the queue is empty.
5. **`E poll()`:** This method is used to retrieve and remove the head of this queue, or returns null if this queue is empty.
6. **`E peek()`:** This method is used to retrieve, but does not remove, the head of this queue, or returns null if this queue is empty.

Some of the commonly used classes that implement the `Queue` interface are `LinkedList`, `PriorityQueue`, and `ArrayDeque`. Each of these classes has different performance characteristics and usage.

Here's an example of using a `Queue` in Java:

[ ]:
```java
Queue<String> queue = new LinkedList<>();
queue.add("Apple");
queue.add("Banana");
queue.add("Cherry");
System.out.println(queue.poll()); // Outputs: Apple
```

```
Apple
```

In this example, a `Queue` of `String` objects is created using `LinkedList`, and three elements are added to it. The `poll` method is then used to retrieve and remove the head of the queue.

12 PriorityQueue

`PriorityQueue` is a class in Java that implements the `Queue` interface and provides the functionality of a priority queue. A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their ordering in the queue.

Here are some key characteristics and methods of `PriorityQueue`:

1. **Priority:** Elements in the `PriorityQueue` are ordered according to their natural ordering, or by a `Comparator` provided at queue construction time.
2. **Non-Synchronized:** `PriorityQueue` is not synchronized. If multiple threads access a priority queue concurrently, it must be synchronized externally.
3. **Nulls are not allowed:** `PriorityQueue` doesn't allow null values to be inserted. This is because the `PriorityQueue` uses the `compareTo` method, and many objects' `compareTo` will throw a `NullPointerException` when passed a null.
4. **Non-Blocking:** `PriorityQueue` is a non-blocking queue. It doesn't enforce a maximum limit on the number of elements that can be inserted.

Key methods of `PriorityQueue` include all methods from the `Queue` interface, and some methods inherited from `AbstractQueue`, `AbstractCollection`, and `Object`. Here are a few:

- **`boolean add(E e)`:** Inserts the specified element into this priority queue.
- **`E poll()`:** Retrieves and removes the head of this queue, or returns null if this queue is empty.
- **`E peek()`:** Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

Here's an example of using a `PriorityQueue`:

[ ]:
```
PriorityQueue<String> queue = new PriorityQueue<>();
queue.add("Apple");
queue.add("Banana");
queue.add("Cherry");
System.out.println(queue.poll()); // Outputs: Apple
```

```
Apple
```

In this example, a `PriorityQueue` of `String` objects is created, and three elements are added to it. The `poll` method is then used to retrieve and remove the head of the queue. The head of this queue is the least element with respect to the specified ordering, which is the natural ordering for `String` objects.

13 Deque Interface?

The `Deque` interface in Java, which stands for Double Ended Queue, is a part of the Java Collections Framework. It extends the `Queue` interface and supports element insertion and removal at both ends.

Here are some key characteristics and methods of `Deque`:

1. **Double Ended:** Elements can be added or removed from both ends (head or tail) of the deque.
2. **Nulls are not allowed:** `Deque` implementations do not allow insertion of null elements. Inserting null into a deque will result in a `NullPointerException`.
3. **LIFO and FIFO:** `Deque` can be used both as a Queue (FIFO) and Stack (LIFO) because it can add or remove elements from both ends.

Key methods of `Deque` include:

- **`void addFirst(E e):`** Inserts the specified element at the front of this deque.
- **`void addLast(E e):`** Inserts the specified element at the end of this deque.
- **`E removeFirst():`** Retrieves and removes the first element of this deque.
- **`E removeLast():`** Retrieves and removes the last element of this deque.
- **`E getFirst():`** Retrieves, but does not remove, the first element of this deque.
- **`E getLast():`** Retrieves, but does not remove, the last element of this deque.

Here's an example of using a `Deque` in Java:

[ ]:
```
Deque<String> deque = new LinkedList<>();
deque.addFirst("Apple");
deque.addLast("Banana");
deque.addLast("Cherry");
System.out.println(deque.removeFirst()); // Outputs: Apple
```

```
Apple
```

In this example, a `Deque` of `String` objects is created using `LinkedList`, and three elements are added to it. The `removeFirst` method is then used to retrieve and remove the first element of the deque.

14 ArrayDeque?

`ArrayDeque` is a class in Java that implements the `Deque` interface. It provides a resizable-array implementation of a double-ended queue. `ArrayDeque` has no capacity restrictions and grows as necessary to support usage. It is not thread-safe; in the absence of external synchronization, it does not support concurrent access by multiple threads.

Here are some key characteristics and methods of `ArrayDeque`:

1. **Resizable Array:** `ArrayDeque` uses a resizable array, which can grow as needed.
2. **Nulls are not allowed:** `ArrayDeque` does not allow null elements. Adding null to an `ArrayDeque` will result in a `NullPointerException`.
3. **Faster:** `ArrayDeque` is typically faster than `LinkedList` for the same operations. `ArrayDeque` provides constant-time performance for the standard deque operations (add, remove, update, and check), assuming the deque size stays within a constant factor of its capacity.

Key methods of `ArrayDeque` include all methods from the `Deque` interface, and some methods inherited from `AbstractCollection`, and `Object`. Here are a few:

- **`void addFirst(E e):`** Inserts the specified element at the front of this deque.
- **`void addLast(E e):`** Inserts the specified element at the end of this deque.
- **`E removeFirst():`** Retrieves and removes the first element of this deque.
- **`E removeLast():`** Retrieves and removes the last element of this deque.
- **`E getFirst():`** Retrieves, but does not remove, the first element of this deque.
- **`E getLast():`** Retrieves, but does not remove, the last element of this deque.

Here's an example of using an `ArrayDeque`:

[ ]:
```
ArrayDeque<String> deque = new ArrayDeque<>();
deque.addFirst("Apple");
deque.addLast("Banana");
deque.addLast("Cherry");
System.out.println(deque.removeFirst()); // Outputs: Apple
```

```
Apple
```

In this example, an `ArrayDeque` of `String` objects is created, and three elements are added to it. The `removeFirst` method is then used to retrieve and remove the first element of the deque.

15 Set Interface ?

The `Set` interface in Java is a member of the Java Collections Framework. It extends the `Collection` interface and represents a collection that contains no duplicate elements.

Here are some key characteristics of `Set`:

1. **No Duplicates:** `Set` doesn't allow duplicate elements. That means you can have at most one null value in a Set and you can't have a pair of elements e1 and e2 such that e1.equals(e2).
2. **Order Not Guaranteed:** The `Set` interface does not guarantee any specific order of its elements. However, some implementations of the `Set` interface like `LinkedHashSet` maintain the order of elements.

Key methods of `Set` include all methods from the `Collection` interface, and some methods inherited from `Object`. Here are a few:

- **`boolean add(E e):`** Adds the specified element to this set if it is not already present.
- **`void clear():`** Removes all of the elements from this set.
- **`boolean contains(Object o):`** Returns true if this set contains the specified element.
- **`boolean isEmpty():`** Returns true if this set contains no elements.
- **`boolean remove(Object o):`** Removes the specified element from this set if it is present.
- **`int size():`** Returns the number of elements in this set.

Here's an example of using a `Set` in Java:

[ ]:
```java
Set<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
System.out.println(set.contains("Banana")); // Outputs: true
```

```
true
```

In this example, a `Set` of `String` objects is created using `HashSet`, and three elements are added to it. The `contains` method is then used to check if "Banana" is in the set.

16 HashSet?

`HashSet` is a class in Java that implements the `Set` interface, backed by a hash table (which is actually a `HashMap` instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.

Here are some key characteristics and methods of `HashSet`:

1. **No Duplicates:** Like any other `Set` implementation, `HashSet` doesn't allow duplicate elements.
2. **No Order:** `HashSet` does not guarantee any specific order of elements due to the hash function.
3. **Nulls are allowed:** `HashSet` allows one null element.
4. **Not Synchronized:** `HashSet` is not synchronized. If multiple threads access a hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

Key methods of `HashSet` include all methods from the `Set` and `Collection` interfaces, and some methods inherited from `Object`. Here are a few:

- **`boolean add(E e)`:** Adds the specified element to this set if it is not already present.
- **`void clear()`:** Removes all of the elements from this set.
- **`boolean contains(Object o)`:** Returns true if this set contains the specified element.
- **`boolean isEmpty()`:** Returns true if this set contains no elements.
- **`boolean remove(Object o)`:** Removes the specified element from this set if it is present.
- **`int size()`:** Returns the number of elements in this set.

Here's an example of using a `HashSet`:

[ ]:
```java
HashSet<String> set = new HashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
System.out.println(set.contains("Banana")); // Outputs: true
```

```
true
```

In this example, a `HashSet` of `String` objects is created, and three elements are added to it. The `contains` method is then used to check if "Banana" is in the set.

17 LinkedHashSet?

`LinkedHashSet` is a class in Java that extends `HashSet` and implements the `Set` interface. It maintains a doubly-linked list running through all of its entries for its predictable iteration order. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order).

Here are some key characteristics and methods of `LinkedHashSet`:

1. **No Duplicates:** Like any other `Set` implementation, `LinkedHashSet` doesn't allow duplicate elements.
2. **Order Maintained:** Unlike `HashSet`, `LinkedHashSet` maintains the insertion order of elements. The elements are returned in the order they were inserted when iterating over the set.
3. **Nulls are allowed:** `LinkedHashSet` allows one null element.
4. **Not Synchronized:** `LinkedHashSet` is not synchronized. If multiple threads access a linked hash set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

Key methods of `LinkedHashSet` include all methods from the `Set` and `Collection` interfaces, and some methods inherited from `Object`. Here are a few:

- **`boolean add(E e)`:** Adds the specified element to this set if it is not already present.
- **`void clear()`:** Removes all of the elements from this set.
- **`boolean contains(Object o)`:** Returns true if this set contains the specified element.
- **`boolean isEmpty()`:** Returns true if this set contains no elements.
- **`boolean remove(Object o)`:** Removes the specified element from this set if it is present.
- **`int size()`:** Returns the number of elements in this set.

Here's an example of using a `LinkedHashSet`:

[ ]:
```java
LinkedHashSet<String> set = new LinkedHashSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
System.out.println(set.contains("Banana")); // Outputs: true
```

```
true
```

In this example, a `LinkedHashSet` of `String` objects is created, and three elements are added to it. The `contains` method is then used to check if "Banana" is in the set.

18 SortedSet Interface?

The `SortedSet` interface in Java is a member of the Java Collections Framework and extends the `Set` interface. It is a set that further provides a total ordering on its elements. The elements are ordered using their natural ordering, or by a `Comparator` typically provided at sorted set creation time.

Here are some key characteristics of `SortedSet`:

1. **No Duplicates:** Like any other `Set` implementation, `SortedSet` doesn't allow duplicate elements.
2. **Order Maintained:** `SortedSet` maintains the natural order of elements if no comparator is provided. If a comparator is provided during sorted set creation, the elements will be ordered using the comparator.
3. **Nulls:** Whether or not nulls are allowed depends on the specific implementation. For example, `TreeSet` (which implements `SortedSet`) does not allow null elements.

Key methods of `SortedSet` include all methods from the `Set` interface, and some additional methods to deal with the ordered set. Here are a few:

- **`Comparator<? super E> comparator()`:** Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
- **`E first()`:** Returns the first (lowest) element currently in this set.
- **`E last()`:** Returns the last (highest) element currently in this set.
- **`SortedSet<E> subSet(E fromElement, E toElement)`:** Returns a view of the portion of this set whose elements range from `fromElement`, inclusive, to `toElement`, exclusive.

Here's an example of using a `SortedSet` in Java:

[ ]:
```
SortedSet<String> set = new TreeSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
System.out.println(set.first()); // Outputs: Apple
```

```
Apple
```

In this example, a `SortedSet` of `String` objects is created using `TreeSet`, and three elements are added to it. The `first` method is then used to retrieve the first (lowest) element in the set.

19 TreeSet?

`TreeSet` is a class in Java that implements the `NavigableSet` interface and extends `AbstractSet`. It creates a collection that uses a tree for storage. Objects are stored in a sorted and ascending order. Access and retrieval times are quite fast, which makes `TreeSet` an excellent choice when storing large amounts of sorted information that must be found quickly.

Here are some key characteristics and methods of `TreeSet`:

1. **No Duplicates:** Like any other `Set` implementation, `TreeSet` doesn't allow duplicate elements.
2. **Order Maintained:** `TreeSet` maintains ascending order of elements based on their values. It uses natural ordering of elements for sorting.
3. **Nulls are not allowed:** `TreeSet` does not allow null elements. Adding null to a `TreeSet` will result in a `NullPointerException`.
4. **Not Synchronized:** `TreeSet` is not synchronized. If multiple threads access a tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.

Key methods of `TreeSet` include all methods from the `Set`, `NavigableSet` and `Collection` interfaces, and some methods inherited from `Object`. Here are a few:

- **`boolean add(E e):`** Adds the specified element to this set if it is not already present.
- **`E first():`** Returns the first (lowest) element currently in this set.
- **`E last():`** Returns the last (highest) element currently in this set.
- **`boolean remove(Object o):`** Removes the specified element from this set if it is present.
- **`int size():`** Returns the number of elements in this set.

Here's an example of using a `TreeSet`:

[ ]:
```
TreeSet<String> set = new TreeSet<>();
set.add("Apple");
set.add("Banana");
set.add("Cherry");
System.out.println(set.first()); // Outputs: Apple
```

```
Apple
```

In this example, a `TreeSet` of `String` objects is created, and three elements are added to it. The `first` method is then used to retrieve the first (lowest) element in the set.

20 Explains Java Map Interface and Java Map Hierarchy?

The `Map` interface in Java is a part of the Java Collections Framework and represents a mapping between a key and a value, essentially an object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.

Here are some key characteristics of `Map`:

1. **Key-Value Pairs:** `Map` stores elements in key-value pairs. The key is used to identify the value in the map.
2. **No Duplicates:** `Map` doesn't allow duplicate keys but it allows duplicate values.
3. **Null Keys and Values:** `Map` allows one null key and multiple null values.

Key methods of `Map` include:

- **`void clear():`** Removes all of the mappings from this map.

- **`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<Map.Entry<K,V>> entrySet()`:** Returns a `Set` view of the mappings contained in this map.
- **`V get(Object key)`:** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- **`boolean isEmpty()`:** Returns true if this map contains no key-value mappings.
- **`V put(K key, V value)`:** Associates the specified value with the specified key in this map.
- **`V remove(Object key)`:** Removes the mapping for the specified key from this map if present.
- **`int size()`:** Returns the number of key-value mappings in this map.

Java Map Hierarchy:

The `Map` interface has several implementations, including `HashMap`, `LinkedHashMap`, `TreeMap`, `Hashtable`, `EnumMap`, `WeakHashMap`, `IdentityHashMap`, and `ConcurrentHashMap`.

- **`HashMap`:** It is the implementation of Map, but it doesn't maintain any order.
- **`LinkedHashMap`:** It is similar to `HashMap` but it maintains insertion order.
- **`TreeMap`:** It is similar to `HashMap` but it maintains ascending order.
- **`Hashtable`:** It is similar to `HashMap` but it is synchronized.
- **`EnumMap`:** It is a specialized map implementation for use with enum type keys.
- **`WeakHashMap`:** It is an implementation of the Map interface that stores only weak references to its keys.
- **`IdentityHashMap`:** This class implements the Map interface with a hash table, using reference-equality in place of object-equality when comparing keys (and values).
- **`ConcurrentHashMap`:** It is similar to `HashMap` but it provides better concurrency level.

Here's an example of using a `Map` in Java:

[ ]:
```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);
System.out.println(map.get("Banana")); // Outputs: 2
```

```
2
```

In this example, a `Map` of `String` keys to `Integer` values is created using `HashMap`, and three key-value pairs are added to it. The `get` method is then used to retrieve the value associated with the key "Banana".

20.1 Useful methods of Map interface?

The `Map` interface in Java provides several useful methods for manipulating key-value pairs. Here are some of the most commonly used methods:

- **`void clear()`:** Removes all of the mappings from this map.
- **`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<Map.Entry<K,V>> entrySet()`:** Returns a `Set` view of the mappings contained in this map. Each element in this set is a `Map.Entry` object.
- **`V get(Object key)`:** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- **`boolean isEmpty()`:** Returns true if this map contains no key-value mappings.
- **`V put(K key, V value)`:** Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
- **`V remove(Object key)`:** Removes the mapping for the specified key from this map if present.
- **`int size()`:** Returns the number of key-value mappings in this map.
- **`Collection<V> values()`:** Returns a `Collection` view of the values contained in this map.
- **`Set<K> keySet()`:** Returns a `Set` view of the keys contained in this map.
- **`void putAll(Map<? extends K, ? extends V> m)`:** Copies all of the mappings from the specified map to this map.
- **`V getOrDefault(Object key, V defaultValue)`:** Returns the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key.
- **`V putIfAbsent(K key, V value)`:** If the specified key is not already associated with a value (or is mapped to `null`), associates it with the given value.

Here's an example of using some of these methods:

[ ]:
```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);
System.out.println(map.containsKey("Banana")); // Outputs: true
System.out.println(map.get("Banana")); // Outputs: 2
System.out.println(map.size()); // Outputs: 3
```

```
true
2
3
```

In this example, a `Map` of `String` keys to `Integer` values is created using `HashMap`, and three key-value pairs are added to it. The `containsKey`, `get`, and `size` methods are then used to interact with the map.

20.2 Map.Entry Interface?

The `Map.Entry` interface in Java is a subinterface of `Map`. It represents a key-value pair contained in a map. Each key-value pair is encapsulated as an object of `Map.Entry`.

The `Map.Entry` interface provides methods to manipulate a single map entry:

- **`K getKey()`:** Returns the key corresponding to this entry.
- **`V getValue()`:** Returns the value corresponding to this entry.
- **`V setValue(V value)`:** Replaces the value corresponding to this entry with the specified value.

These methods provide a way to access the key and value of a map entry, and even allow the value to be updated.

Here's an example of using `Map.Entry`:

[ ]:
```
Map<String, Integer> map = new HashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.
 ↪getValue());
}
```

```
Key = Apple, Value = 1
Key = Cherry, Value = 3
Key = Banana, Value = 2
```

In this example, a `Map` of `String` keys to `Integer` values is created using `HashMap`, and three key-value pairs are added to it. Then, the `entrySet` method is used to get a `Set` view of the map, which is iterated over using a for-each loop. In each iteration, the `getKey` and `getValue` methods are used to access the key and value of the current map entry.

21

22 LinkedHashMap?

`LinkedHashMap` is a hash table and linked list implementation of the `Map` interface in Java, with predictable iteration order. It extends `HashMap` and implements the `Map` interface.

Here are some key characteristics of `LinkedHashMap`:

1. **Order Maintained:** Unlike `HashMap`, `LinkedHashMap` 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). This can be altered during construction to be access-order, where the last accessed entry is moved to the end.
2. **No Duplicates:** Like any other `Map` implementation, `LinkedHashMap` doesn't allow duplicate keys but it allows duplicate values.
3. **Nulls:** `LinkedHashMap` allows one null key and multiple null values.
4. **Not Synchronized:** `LinkedHashMap` is not synchronized. If multiple threads access a linked hash map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally.

Key methods of `LinkedHashMap` include all methods from the `Map` interface, and some additional methods inherited from `HashMap`. Here are a few:

- **`void clear()`:** Removes all of the mappings from this map.
- **`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.
- **`V get(Object key)`:** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- **`V put(K key, V value)`:** Associates the specified value with the specified key in this map.

Here's an example of using a `LinkedHashMap`:

[ ]:
```
LinkedHashMap<String, Integer> map = new LinkedHashMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.
 ↪getValue());
}
```

```
Key = Apple, Value = 1
Key = Banana, Value = 2
Key = Cherry, Value = 3
```

In this example, a `LinkedHashMap` of `String` keys to `Integer` values is created, and three key-value pairs are added to it. Then, the `entrySet` method is used to get a `Set` view of the map, which is iterated over using a for-each loop. In each iteration, the key and value of the current map entry are printed.

23 TreeMap and HashMap?

TreeMap

`TreeMap` in Java is a Red-Black tree based implementation of the `Map` interface. This class guarantees that the map will be in ascending key order, sorted according to the natural order for the key's class (see `Comparable`), or by the comparator provided at the time of creation.

Key characteristics of `TreeMap`:

1. **Ordering:** The `TreeMap` class is efficient for traversing the keys in a sorted order. The keys are ordered using their natural ordering, or by a `Comparator` provided at map creation time, depending on which constructor is used.
2. **Nulls:** `TreeMap` does not allow null keys but allows multiple null values.
3. **Not Synchronized:** `TreeMap` is not synchronized. If multiple threads access a map concurrently, and at least one of the threads modifies the map structurally, it must be synchronized externally.
4. **Performance:** This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put`, and `remove` operations.

SortedMap

`SortedMap` is an interface in Java Collection Framework. It is a child interface of `Map`. It is used to store key-value pairs in sorted order of keys. It is a map that further provides a total ordering on its keys. The map is ordered according to the natural ordering of its keys, or by a `Comparator` typically provided at sorted map creation time.

Key characteristics of `SortedMap`:

1. **Ordering:** The `SortedMap` interface provides operations for normal `Map` as well as for the following operations:
   - Range view — performs arbitrary range operations on the sorted map.
   - Endpoints — returns the first or the last key in the sorted map.
   - Comparator access — returns the comparator, if any, used to sort the map.
2. **Nulls:** Whether or not null keys and values are allowed depends on the implementation.

Here's an example of using a `TreeMap`:

[ ]:
```
TreeMap<String, Integer> map = new TreeMap<>();
map.put("Apple", 1);
map.put("Banana", 2);
map.put("Cherry", 3);
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    System.out.println("Key = " + entry.getKey() + ", Value = " + entry.
 ↪getValue());
}
```

```
Key = Apple, Value = 1
Key = Banana, Value = 2
Key = Cherry, Value = 3
```

In this example, a `TreeMap` of `String` keys to `Integer` values is created, and three key-value pairs are added to it. Then, the `entrySet` method is used to get a `Set` view of the map, which is iterated over using a for-each loop. In each iteration, the key and value of the current map entry are printed. The keys are printed in ascending order, as guaranteed by `TreeMap`.

24 What is the difference between HashSet and LinkedHashSet?

`HashSet` and `LinkedHashSet` are both implementations of the `Set` interface in Java, but they have some key differences:

1. **Ordering:** `HashSet` does not maintain any order of its elements. The order of elements can vary on each run. On the other hand, `LinkedHashSet` maintains the insertion order of elements. Elements added first will be the first ones retrieved during iteration.
2. **Performance:** `HashSet` is generally faster for operations like `add`, `remove`, and `contains` because it has constant time performance for these operations irrespective of the number of elements. `LinkedHashSet` would have slightly lower performance for these operations but still performs well because it's implemented as a hash table with linked list running through it.
3. **Usage:** If you need a `Set` implementation where the order of elements matters, `LinkedHashSet` is an appropriate choice. If you don't care about the order and just want the operations to be faster, `HashSet` would be a better choice.

Here's an example of how the two differ:

[ ]:
```
Set<String> hashSet = new HashSet<>();
hashSet.add("Apple");
hashSet.add("Banana");
hashSet.add("Cherry");
System.out.println("HashSet: " + hashSet);

Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("Apple");
linkedHashSet.add("Banana");
linkedHashSet.add("Cherry");
System.out.println("LinkedHashSet: " + linkedHashSet);
```

```
HashSet: [Apple, Cherry, Banana]
LinkedHashSet: [Apple, Banana, Cherry]
```

In this example, both `HashSet` and `LinkedHashSet` have the same elements added in the same order. However, the output order in `HashSet` can vary on each run, while `LinkedHashSet` will always output the elements in the order they were inserted.

25 HashTable

`Hashtable` is part of the Java Collections Framework but it's considered a legacy class, predating the collections framework. It's similar to `HashMap` as it also stores key-value pairs, but there are some differences:

1. **Synchronization:** `Hashtable` is synchronized, which means it is thread-safe and can be shared between multiple threads. This comes with a cost in terms of performance. If a thread-safe implementation is not needed, it is recommended to use `HashMap` in place of `Hashtable`.
2. **Null keys and null values:** `Hashtable` does not allow null keys or null values. In contrast, `HashMap` allows one null key and multiple null values.
3. **Ordering:** `Hashtable` does not guarantee that the order of the map will remain constant over time.
4. **Iterator:** `Hashtable` provides an enumerator to iterate over the values. The iterator provided by `Hashtable` is not fail-fast, unlike `HashMap`.

Here are some key methods provided by `Hashtable`:

- **`void clear()`:** Clears the hashtable so that it contains no keys.
- **`boolean contains(Object value)`:** Tests if some key maps into the specified value in this hashtable.
- **`boolean containsKey(Object key)`:** Tests if the specified object is a key in this hashtable.
- **`V get(Object key)`:** Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- **`boolean isEmpty()`:** Tests if this hashtable maps no keys to values.
- **`V put(K key, V value)`:** Maps the specified key to the specified value in this hashtable.
- **`V remove(Object key)`:** Removes the key (and its corresponding value) from this hashtable.

Here's an example of using a `Hashtable`:

[ ]:
```
Hashtable<String, Integer> hashtable = new Hashtable<>();
hashtable.put("Apple", 1);
hashtable.put("Banana", 2);
hashtable.put("Cherry", 3);
System.out.println("Value for key 'Banana': " + hashtable.get("Banana"));
```

```
Value for key 'Banana': 2
```

In this example, a `Hashtable` of `String` keys to `Integer` values is created, and three key-value pairs are added to it. Then, the `get` method is used to retrieve the value associated with the key "Banana".

26 Thank You!