Collection in Java

Daniel Mercado Cavazos

Index

Introductio to Java Collections.

In Java, **collections** are objects that group multiple elements into a single unit. They are like containers that hold and organize data so that you can perform various operations on the data, such as adding, removing, and searching for elements.

The **Java Collections Framework** provides a set of classes and interfaces that make it easy to work with groups of objects. Here are some key points to understand:

1. **Interfaces**: These define the standard operations that collections should support. The main interfaces in the Java Collections Framework include:
    - **Collection**: The root interface for all collection types.
    - **List**: An ordered collection (also known as a sequence). Examples: `ArrayList`, `LinkedList`.
    - **Set**: A collection that does not allow duplicate elements. Examples: `HashSet`, `TreeSet`.
    - **Queue**: A collection used to hold elements prior to processing. Examples: `LinkedList`, `PriorityQueue`.
    - **Map**: A collection that maps keys to values, without duplicate keys. Examples: `HashMap`, `TreeMap`.

The Java Collections Framework provides a comprehensive, high-performance set of classes and interfaces for working with collections of objects. Understanding the different types of collections, their performance characteristics, and how to use them effectively is crucial for any Java developer.

Important Concepts in Java Collections Framework

- **Generics**: Introduced in Java 5, generics allow you to define collections that enforce type safety at compile-time. For example, `List<String>` ensures that only `String` objects can be added to the list.
- **Autoboxing/Unboxing**: Collections only store objects, so primitive types (like `int`, `double`) are automatically converted to their wrapper types (like `Integer`, `Double`) when added to a collection.
- **Algorithms**: The JCF provides various algorithms for sorting, searching, and modifying collections, implemented as static methods in the `Collections` class (e.g., `Collections.sort()`, `Collections.binarySearch()`).
- **Iterator and Iterable**: `Iterator<E>` is an interface for iterating over collections, providing methods like `hasNext()`, `next()`, and `remove()`. Collections implementing the `Iterable<E>` interface can be used in enhanced for-loops.

Performance Considerations

Different implementations of collections have different performance characteristics:

- **ArrayList** is fast for random access but slow for insertions/deletions in the middle of the list.
- **LinkedList** is slower for random access but fast for insertions/deletions.
- **HashSet** and **HashMap** offer constant-time performance for most operations, but their performance can degrade if the hash function results in many collisions.
- **TreeSet** and **TreeMap** guarantee log-time complexity for basic operations, with the added benefit of maintaining order.

Concurrency in Collections

Java provides concurrent versions of its collections (in the `java.util.concurrent` package):

- **ConcurrentHashMap**: A thread-safe version of `HashMap`.
- **CopyOnWriteArrayList**: A thread-safe version of `ArrayList`, optimized for situations where reads are more frequent than writes.

Additionally, synchronization wrappers (e.g., `Collections.synchronizedList`) can make existing collections thread-safe, though they are generally less efficient than the concurrent collections.

Java Stream API Integration

With Java 8, the `Stream` API was introduced, allowing for functional-style operations on collections (like `map`, `filter`, `reduce`). Streams can be sequential or parallel, enabling efficient bulk operations on large data sets.

List interface

The `List` interface in Java is part of the Java Collections Framework and represents an ordered collection (also known as a sequence). It is a subtype of the `Collection` interface, which means it inherits all the methods from `Collection` and adds several methods of its own to allow working with elements based on their position in the list.

Key Characteristics of the `List` Interface

1. **Ordered Collection**: The `List` interface maintains the order of elements. Elements can be accessed by their position (index) in the list.
2. **Duplicate Elements**: Unlike `Set`, a `List` can contain duplicate elements. This allows the same element to appear more than once.
3. **Indexed Access**: You can access elements in a `List` by their position using an index. The first element is at index 0, the second at index 1, and so on.
4. **Positional Access**: The `List` interface provides methods to manipulate elements at a specific position in the list (e.g., `get(int index)`, `set(int index, E element)`).

Common Implementations of the `List` Interface

Several classes implement the `List` interface, each with different performance characteristics:

- **ArrayList**: Resizable array implementation, best for random access.
- **LinkedList**: Doubly linked list implementation, best for frequent insertions/deletions.
- **Vector**: Synchronized resizable array implementation, similar to `ArrayList` but thread-safe.
- **Stack**: A subclass of `Vector` that represents a last-in-first-out (LIFO) stack of elements.

Important Methods in the `List` Interface

Here are some of the most commonly used methods provided by the `List` interface:

1. **Adding Elements**:
    - `boolean add(E element)`: Appends the specified element to the end of the list.
    - `void add(int index, E element)`: Inserts the specified element at the specified position in the list.
2. **Removing Elements**:
    - `E remove(int index)`: Removes the element at the specified position in the list.
    - `boolean remove(Object o)`: Removes the first occurrence of the specified element from the list, if present.
3. **Accessing Elements**:
    - `E get(int index)`: Returns the element at the specified position in the list.
    - `E set(int index, E element)`: Replaces the element at the specified position in the list with the specified element.
4. **Searching**:
    - `int indexOf(Object o)`: Returns the index of the first occurrence of the specified element, or -1 if the element is not found.
    - `int lastIndexOf(Object o)`: Returns the index of the last occurrence of the specified element, or -1 if the element is not found.
5. **Size and Emptiness**:
    - `int size()`: Returns the number of elements in the list.
    - `boolean isEmpty()`: Returns `true` if the list contains no elements.
6. **Sub-list**:
    - `List<E> subList(int fromIndex, int toIndex)`: Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.
7. **Iteration**:
    - The `List` interface supports the use of an `Iterator` to iterate over the elements in sequence.

- You can also use a `ListIterator`, which provides bi-directional iteration and allows modification of the list during iteration.

Example:

```
import java.util.*;

public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>();

        // Adding elements
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");
        names.add("Alice"); // Duplicate

        // Accessing elements
        System.out.println("Element at index 1: " + names.get(1)); // Bob

        // Modifying elements
        names.set(2, "Dave");
        System.out.println("Modified list: " + names); // [Alice, Bob, Dave, Alice]

        // Removing elements
        names.remove(3); // Removes "Alice" at index 3
        System.out.println("After removal: " + names); // [Alice, Bob, Dave]

        // Iterating over the list
        for (String name : names) {
            System.out.println(name);
        }

        // Finding an element
        int index = names.indexOf("Bob");
        System.out.println("Index of Bob: " + index); // 1
    }
}
```

Set interface

The Set interface in Java is part of the Java Collections Framework and represents a collection that does not allow duplicate elements. It is a subtype of the Collection interface and is designed for scenarios where you want to store unique elements, ensuring that no duplicates are present.

Key Characteristics of the `Set` Interface

1. **No Duplicate Elements**: A `Set` cannot contain duplicate elements. If you try to add a duplicate element to a `Set`, the existing element is not replaced, and the `add` operation returns `false`.
2. **Unordered Collection**: The `Set` interface does not guarantee that the order of elements will be preserved. The order depends on the specific implementation of the `Set`.
3. **Unique Use Cases**: `Set` is ideal for use cases where the uniqueness of elements is important, such as storing a collection of user IDs, product codes, or any other kind of identifiers.

Common Implementations of the `Set` Interface

Several classes implement the `Set` interface, each with different characteristics:

1. **HashSet**:
    - **Implementation**: Backed by a hash table (actually a `HashMap` instance).
    - **Order**: Does not guarantee any order of elements.
    - **Performance**: Offers constant-time performance for basic operations like `add`, `remove`, and `contains` (assuming the hash function disperses elements properly).
    - **Use Case**: Best for quick lookups and avoiding duplicates.
2. **LinkedHashSet**:
    - **Implementation**: Extends `HashSet` with a linked list running through all of its entries.
    - **Order**: Maintains the insertion order of elements.

- **Performance**: Slightly slower than `HashSet` due to maintaining the order, but still offers constant-time performance.
- **Use Case**: Ideal when you need both uniqueness and predictable iteration order.

3. **TreeSet**:
   - **Implementation**: Backed by a red-black tree (actually a `NavigableMap`).
   - **Order**: Maintains elements in their natural order (defined by `Comparable` or a custom `Comparator`).
   - **Performance**: Provides guaranteed log-time performance for basic operations.
   - **Use Case**: Useful when you need a sorted set of elements.

Important Methods in the `Set` Interface

The `Set` interface inherits all methods from the `Collection` interface but has some unique behaviors due to the no-duplicates constraint. Here are some commonly used methods:

1. **Adding Elements**:
   - `boolean add(E element)`: Adds the specified element to the set if it is not already present. Returns `true` if the element was added, `false` otherwise.
2. **Removing Elements**:
   - `boolean remove(Object o)`: Removes the specified element from the set if it is present. Returns `true` if the element was removed.
3. **Checking for Existence**:
   - `boolean contains(Object o)`: Returns `true` if the set contains the specified element.
4. **Size and Emptiness**:
   - `int size()`: Returns the number of elements in the set.
   - `boolean isEmpty()`: Returns `true` if the set contains no elements.
5. **Iteration**:

- You can iterate over a `Set` using an enhanced `for` loop or an `Iterator`.

Example:

```
import java.util.*;

public class SetExample {
    public static void main(String[] args) {
        Set<String> uniqueNames = new HashSet<>();

        // Adding elements
        uniqueNames.add("Alice");
        uniqueNames.add("Bob");
        uniqueNames.add("Charlie");
        uniqueNames.add("Alice"); // Duplicate, will not be added

        // Size of the set
        System.out.println("Size of the set: " + uniqueNames.size()); // 3

        // Checking for existence
        System.out.println("Does set contain 'Bob'? " +
uniqueNames.contains("Bob")); // true

        // Iterating over the set
        for (String name : uniqueNames) {
            System.out.println(name);
        }

        // Removing an element
        uniqueNames.remove("Charlie");
        System.out.println("After removal: " + uniqueNames); // [Alice,
Bob]
    }
}
```

Queue interface

The `Queue` interface in Java is a part of the Java Collections Framework and represents a collection designed for holding elements prior to processing. Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. However, there are different types of queues with different ordering principles.

Key Characteristics of the Queue Interface

1. **FIFO Behavior**: The `Queue` interface is typically based on FIFO, where elements are added at the end and removed from the front. However, other orderings (e.g., priority-based) are also supported by different implementations.
2. **No Blocking Operations**: Unlike `BlockingQueue`, which is in the `java.util.concurrent` package, the `Queue` interface does not provide blocking operations. Operations on a `Queue` return immediately, either succeeding or failing.
3. **Primary Operations**:
    - **Enqueue**: Adding an element to the queue.
    - **Dequeue**: Removing an element from the queue.

Common Implementations of the Queue Interface

There are several implementations of the `Queue` interface, each with different behaviors:

1. **LinkedList**:
    - **Implementation**: Implements both the `Queue` and `Deque` interfaces, functioning as a doubly-linked list.
    - **Order**: Maintains elements in FIFO order.
    - **Use Case**: Best for use cases where you need a basic queue that can grow dynamically.
2. **PriorityQueue**:

- **Implementation**: Implements a priority heap where elements are ordered according to their natural order or by a `Comparator` provided at queue creation.
- **Order**: Not a FIFO queue; elements are ordered based on priority.
- **Use Case**: Suitable when you need to process elements according to priority rather than order of insertion.

3. **ArrayDeque**:
   - **Implementation**: Implements both `Deque` and `Queue` interfaces using a resizable array.
   - **Order**: Maintains elements in FIFO order when used as a queue.
   - **Use Case**: Ideal when you need a double-ended queue (deque) that can be used as both a stack and a queue.

Important Methods in the Queue Interface

The `Queue` interface provides several methods that can be grouped into two categories: methods that throw exceptions and methods that return special values when an operation fails.

1. **Inserting Elements**:
   - `boolean add(E e)`: Inserts the specified element into the queue. Throws an exception if the queue is full.
   - `boolean offer(E e)`: Inserts the specified element into the queue. Returns `false` if the queue is full.
2. **Removing Elements**:
   - `E remove()`: Removes and returns the head of the queue. Throws an exception if the queue is empty.
   - `E poll()`: Removes and returns the head of the queue. Returns `null` if the queue is empty.
3. **Inspecting Elements**:
   - `E element()`: Returns the head of the queue without removing it. Throws an exception if the queue is empty.

- `E peek()`: Returns the head of the queue without removing it. Returns `null` if the queue is empty.

Example:

```
import java.util.*;

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();

        // Adding elements
        queue.offer("Alice");
        queue.offer("Bob");
        queue.offer("Charlie");

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

        // Removing elements
        System.out.println("Removed: " + queue.poll()); // Alice
        System.out.println("Removed: " + queue.poll()); // Bob

        // Checking the size of the queue
        System.out.println("Queue size: " + queue.size()); // 1

        // Inspecting and removing the remaining element
        System.out.println("Head of the queue: " + queue.element());
// Charlie
        System.out.println("Removed: " + queue.remove()); // Charlie

        // Queue is now empty
        System.out.println("Queue empty? " + queue.isEmpty()); //
true
    }
}
```

Map Interface

Key Characteristics of the `Map` Interface

1. **Key-Value Pair Storage**: The `Map` interface stores elements in key-value pairs, where each key is associated with a specific value.
2. **Unique Keys**: Keys in a `Map` must be unique. If you try to add a key that already exists, the old value associated with that key will be replaced by the new value.
3. **No Inherent Order**: The `Map` interface does not guarantee any order of keys or values. However, specific implementations like `LinkedHashMap` and `TreeMap` do maintain a particular order.

Common Implementations of the `Map` Interface

Several classes implement the `Map` interface, each with different characteristics:

1. **HashMap**:
    - **Implementation**: Hash table-based implementation.
    - **Order**: Does not guarantee the order of keys or values.
    - **Performance**: Provides constant-time performance for basic operations like `get` and `put` (assuming the hash function disperses elements properly).
    - **Use Case**: Best for quick lookups when order is not important.
2. **LinkedHashMap**:
    - **Implementation**: Extends `HashMap` with a linked list running through all of its entries.
    - **Order**: Maintains insertion order (the order in which keys were added).

- **Performance**: Slightly slower than `HashMap` due to maintaining the order, but still offers constant-time performance.
- **Use Case**: Ideal when you need both fast lookups and a predictable iteration order.

3. **TreeMap**:
   - **Implementation**: Red-black tree-based implementation.
   - **Order**: Maintains keys in their natural order (defined by `Comparable` or a custom `Comparator`).
   - **Performance**: Provides log-time performance for basic operations.
   - **Use Case**: Useful when you need a sorted map.
4. **Hashtable**:
   - **Implementation**: Synchronized hash table.
   - **Order**: Does not guarantee any order.
   - **Performance**: Generally slower than `HashMap` due to synchronization.
   - **Use Case**: Suitable when thread safety is required.

Important Methods in the `Map` Interface

The `Map` interface provides several methods for managing key-value pairs:

1. **Inserting and Updating Values**:
   - `V put(K key, V value)`: Associates the specified value with the specified key. If the key already exists, the old value is replaced.
2. **Retrieving Values**:
   - `V get(Object key)`: Returns the value associated with the specified key, or `null` if the key is not found.
3. **Removing Entries**:
   - `V remove(Object key)`: Removes the mapping for the specified key and returns the associated value, or `null` if the key was not found.

4. **Checking for Keys/Values**:
    - `boolean containsKey(Object key)`: Returns `true` if the map contains a mapping for the specified key.
    - `boolean containsValue(Object value)`: Returns `true` if the map contains one or more keys mapping to the specified value.
5. **Size and Emptiness**:
    - `int size()`: Returns the number of key-value mappings in the map.
    - `boolean isEmpty()`: Returns `true` if the map contains no key-value mappings.
6. **Iterating Over Keys/Values/Entries**:
    - `Set<K> keySet()`: Returns a `Set` view of the keys contained in the map.
    - `Collection<V> values()`: Returns a `Collection` view of the values contained in the map.
    - `Set<Map.Entry<K, V>> entrySet()`: Returns a `Set` view of the key-value mappings contained in the map.

Example:

```
import java.util.*;

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

        // Adding entries
        ageMap.put("Alice", 30);
        ageMap.put("Bob", 25);
        ageMap.put("Charlie", 35);

        // Retrieving a value
        System.out.println("Age of Bob: " + ageMap.get("Bob")); // 25

        // Checking for a key
        System.out.println("Contains key 'Alice'? " +
ageMap.containsKey("Alice")); // true

        // Iterating over keys
        for (String name : ageMap.keySet()) {
            System.out.println(name + " is " + ageMap.get(name) + "
years old");
        }

        // Removing an entry
        ageMap.remove("Charlie");

        // Checking the size
        System.out.println("Size of the map: " + ageMap.size()); // 2
    }
}
```