Adv Java 02 - Collections

Agenda

- Intro to Collections
    - Common Collection Interfaces
    - List Interface
    - Queue Interface
    - Set Interface
    - Map Interface
- Intro to Iterators
    - Using Iterators
    - Iterator Methods
- Additonal Concepts
    - Using Custom Object as Key with Hashmap etc

Introduction

Java Collections Framework provides a set of interfaces and classes to store and manipulate groups of objects. Collections make it easier to work with groups of objects, such as lists, sets, and maps. In this beginner-friendly tutorial, we'll explore the basics of Java Collections and how to use iterators to traverse through them.

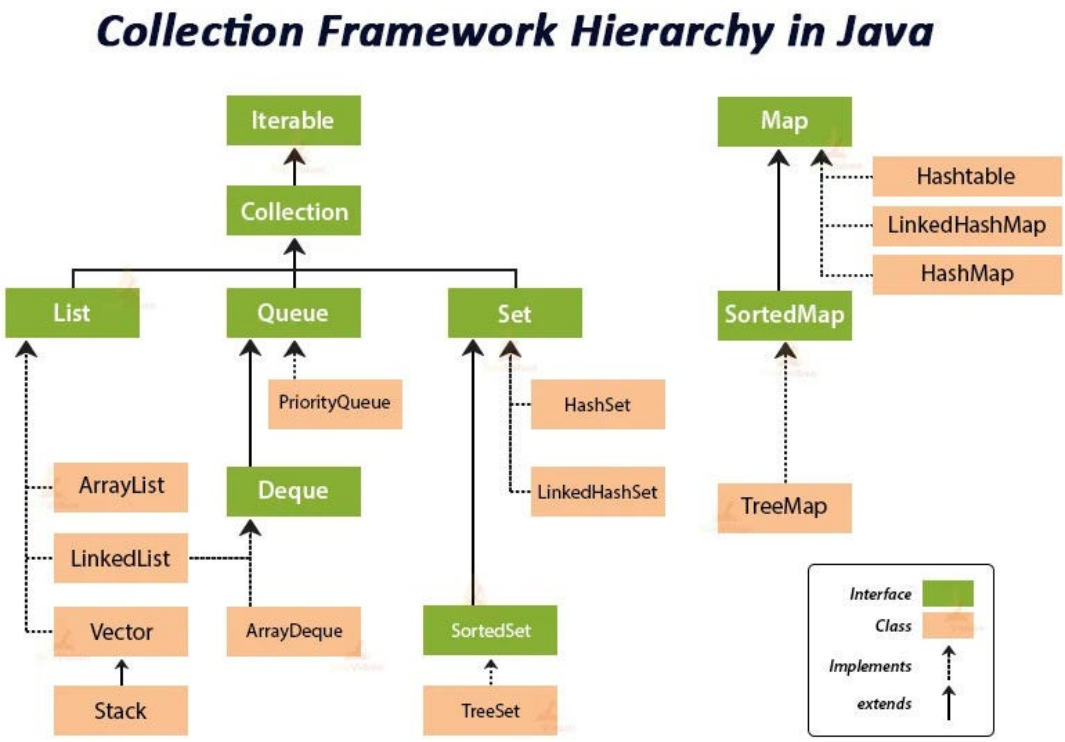
1. Introduction to Java Collections

Java Collections provide a unified architecture for representing and manipulating groups of objects. The Collections Framework includes interfaces, implementations, and algorithms that simplify the handling of groups of objects.

Collection PlayList - Video Tutorial

2. Common Collection Interfaces

There are several core interfaces in the Collections Framework:



Collection: The root interface for all collections. It represents a group of objects, and its subinterfaces include List, Set, and Queue.

List: An ordered collection that allows duplicate elements. Implementations include ArrayList, LinkedList, and Vector.

Queue:: The Queue interface in Java is part of the Java Collections Framework and extends the Collection interface. Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering. Implementations include ArrayDeque, LinkedList, PriorityQueue etc.

Set: An unordered collection that does not allow duplicate elements. Implementations include HashSet, LinkedHashSet, and TreeSet.

Map: A collection that maps keys to values. Implementations include HashMap, LinkedHashMap, TreeMap, and Hashtable.

Example-1 List Interface and ArrayList

The List interface extends the Collection interface and represents an ordered collection of elements. One of the common implementations is ArrayList. Let's see a simple example:

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

public class ListExample {
```

```
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
        myList.add("Java");
        myList.add("Python");
        myList.add("C++");

        System.out.println("List elements: " + myList);
    }
}
```

Example-2 Set Interface and HashSet

The Set interface represents an unordered collection of unique elements. One of the common implementations is HashSet. Here's a simple example:

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

public class SetExample {
    public static void main(String[] args) {
        Set<String> mySet = new HashSet<>();
        mySet.add("Apple");
        mySet.add("Banana");
        mySet.add("Orange");

        System.out.println("Set elements: " + mySet);
    }
}
```

Example-3 Map Interface and HashMap

The Map interface represents a collection of key-value pairs. One of the common implementations is HashMap. Let's see an example:

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

public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> myMap = new HashMap<>();
        myMap.put("Java", 20);
        myMap.put("Python", 15);
        myMap.put("C++", 10);

        System.out.println("Map elements: " + myMap);
    }
}
```

Introduction to Iterators

An iterator is an interface that provides a way to access elements of a collection one at a time. The Iterator interface includes methods for iterating over a collection and retrieving elements.

Let's see how to use iterators with a simple example using a List:

Example - 1 List Iterator

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

public class IteratorExample {
    public static void main(String[] args) {
        List<String> myList = new ArrayList<>();
        myList.add("Java");
        myList.add("Python");
        myList.add("C++");

        // Getting an iterator
        Iterator<String> iterator = myList.iterator();

        // Iterating through the elements
        while (iterator.hasNext()) {
            String element = iterator.next();
            System.out.println("Element: " + element);
        }
    }
}
```

Example-2 Iterating Over Priority Queue

```
public class PriorityQueueIteratorExample {
    public static void main(String[] args) {
        // Creating a PriorityQueue with Integer elements
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();

        // Adding elements to the PriorityQueue
        priorityQueue.offer(30);
        priorityQueue.offer(10);
        priorityQueue.offer(20);

        // Using Iterator to iterate over elements in PriorityQueue
        System.out.println("Elements in PriorityQueue using Iterator:");

        Iterator<Integer> iterator = priorityQueue.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

n this example, we create a PriorityQueue of integers and add three elements to it. We then use an Iterator to iterate over the elements and print them.

Keep in mind that when using a PriorityQueue, the order of retrieval is based on the natural order (if the elements are comparable) or a provided comparator. The element with the highest priority comes out first.

It's important to note that the iterator does not guarantee any specific order when iterating over the elements of a PriorityQueue.

Iterator Methods

The Iterator interface provides several methods, including:

- hasNext(): Returns true if the iteration has more elements.
- next(): Returns the next element in the iteration.
- remove(): Removes the last element returned by next() from the underlying collection (optional operation).

Here's an example demonstrating the use of these methods:

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

public class IteratorMethodsExample {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);

        Iterator<Integer> iterator = numbers.iterator();

        // Using hasNext() and next() methods
        while (iterator.hasNext()) {
            Integer number = iterator.next();
            System.out.println("Number: " + number);

            // Using remove() method (optional operation)
            iterator.remove();
        }

        System.out.println("Updated List: " + numbers);
    }
}
```

Additional Concepts

Hashmap with Custom Objects

Using a HashMap with custom objects in Java involves a few steps. Let's go through the process step by step. Suppose you have a custom object called Person with attributes like id, name, and age.

- Step 1: Create the Custom Object

```
public class Person {
  private int id;
  private String name;
  private int age;

  public Person(int id, String name, int age) {
      this.id = id;
      this.name = name;
      this.age = age;
  }

  // Getters and setters (not shown for brevity)

  @Override
  public String toString() {
      return "Person{id=" + id + ", name='" + name + "', age=" + age + '}';
  }
}
```

- Step 2: Use Person as a Key in HashMap Now, you can use Person objects as keys in a HashMap. For example:

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

public class HashMapExample {
    public static void main(String[] args) {
        // Create a HashMap with Person objects as keys
        Map<Person, String> personMap = new HashMap<>();

        // Add entries
        Person person1 = new Person(1, "Alice", 25);
        Person person2 = new Person(2, "Bob", 30);

        personMap.put(person1, "Employee");
        personMap.put(person2, "Manager");

        // Retrieve values using Person objects as keys
        Person keyToLookup = new Person(1, "Alice", 25);
        String position = personMap.get(keyToLookup);

        System.out.println("Position for " + keyToLookup + ": " + position);
    }
}
```

In this example, Person objects are used as keys, and the associated values represent their positions. Note that for keys to work correctly in a HashMap, the custom class (Person in this case) should override the hashCode() and equals() methods.

- Step 3: Override hashCode() and equals()

```
public class Person {
    // ... existing code

    @Override
    public int hashCode() {
        return Objects.hash(id, name, age);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Person person = (Person) obj;

        return id == person.id && age == person.age && Objects.equals(name,
person.name);
    }
}
```

By overriding these methods, you ensure that the HashMap correctly handles collisions and identifies when two Person objects are considered equal.

Important Considerations

Immutability: It's often a good practice to make the custom objects used as keys immutable. This helps in maintaining the integrity of the HashMap because the keys should not be modified after being used.

Consistent hashCode():

Ensure that the hashCode() method returns the same value for two objects that are considered equal according to the equals() method. This ensures proper functioning of the HashMap.

Performance: Consider the performance implications when using complex objects as keys. If the hashCode() and equals() methods are computationally expensive, it might affect the performance of the HashMap. By following these steps and considerations, you can effectively use custom objects as keys in a HashMap in Java.

Summary

Java Collections and Iterators are fundamental concepts for handling groups of objects efficiently. Understanding the different collection interfaces, implementing classes, and utilizing iterators will empower you to work with collections effectively in your Java applications. Practice and explore the various methods available in the Collections Framework to enhance your programming skills.

-- End --