Java Collections Notes

1. List

List is an ordered collection that allows duplicates and provides indexed access.

Common Implementations: ArrayList, LinkedList, Vector

Real-world use: Shopping cart in an e-commerce app.

Example:

```
import java.util.ArrayList;

public class ListExample {
   public static void main(String[] args) {
      ArrayList<String> shoppingList = new ArrayList<>();
      shoppingList.add("Milk");
      shoppingList.add("Bread");
      shoppingList.add("Eggs");
      shoppingList.add("Milk"); // duplicates allowed

      for (String item : shoppingList) {
         System.out.println(item);
      }
   }
}
```

Common Methods:

- add(E e): Adds element

- get(int index): Gets element at index

- set(int index, E element): Replaces element

- remove(int index): Removes element

- size(): Returns size

- contains(Object o): Checks existence

- indexOf(Object o): Finds index

2. Set

Set is an unordered collection that does not allow duplicates.

Common Implementations: HashSet, LinkedHashSet, TreeSet

Real-world use: Storing unique usernames.

Example:

```
import java.util.HashSet;

public class SetExample {
   public static void main(String[] args) {
      HashSet<String> cities = new HashSet<>();
      cities.add("Delhi");
      cities.add("Mumbai");
      cities.add("Delhi"); // duplicate, won't be added

      for (String city : cities) {
         System.out.println(city);
      }
   }
}
```

Common Methods:

- add(E e): Adds element

- remove(Object o): Removes element

- contains(Object o): Checks existence

- size(): Returns size

- isEmpty(): Checks if empty

3. Map

Map stores key-value pairs where keys are unique.

Common Implementations: HashMap, LinkedHashMap, TreeMap

Real-world use: Storing student scores by name.

Example:

```
import java.util.HashMap;

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

```
        studentMarks.put("Alice", 85);
        studentMarks.put("Bob", 90);
        studentMarks.put("Alice", 95); // overwrites previous value

        for (String name : studentMarks.keySet()) {
            System.out.println(name + ": " + studentMarks.get(name));
        }
    }
}
```

Common Methods:

- put(K key, V value): Adds key-value

- get(Object key): Gets value

- remove(Object key): Removes key

- containsKey(Object key): Checks key

- containsValue(Object value): Checks value

- keySet(): Returns keys

- values(): Returns values

4. Queue

Queue follows FIFO (First-In-First-Out) structure.

Common Implementations: LinkedList, PriorityQueue, ArrayDeque

Real-world use: Customer service ticketing system.

Example:

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

public class QueueExample {
    public static void main(String[] args) {
        Queue<String> serviceQueue = new LinkedList<>();
        serviceQueue.add("Customer1");
        serviceQueue.add("Customer2");

        while (!serviceQueue.isEmpty()) {
            System.out.println("Serving: " + serviceQueue.poll());
```

```
    }
  }
}
```

Common Methods:

- add(E e): Adds element

- offer(E e): Adds safely

- poll(): Removes head

- peek(): Views head

- remove(): Removes head

5. Deque (Double-Ended Queue)

Deque allows insertion and removal from both ends.

Common Implementations: ArrayDeque, LinkedList

Real-world use: Browser back/forward navigation.

Example:

```
import java.util.ArrayDeque;
import java.util.Deque;

public class DequeExample {
  public static void main(String[] args) {
    Deque<String> browserHistory = new ArrayDeque<>();
    browserHistory.addFirst("Page1");
    browserHistory.addFirst("Page2");

    System.out.println("Back to: " + browserHistory.removeFirst());
  }
}
```

Common Methods:

- addFirst(E e): Adds to front

- addLast(E e): Adds to end

- removeFirst(): Removes front

- removeLast(): Removes end

- peekFirst(): Views front

- peekLast(): Views end

Comparison Table

Feature Comparison:

• Order:

- HashMap: No guaranteed order

- LinkedHashMap: Maintains insertion order

- TreeMap: Sorted by natural order or comparator

• Performance:

- HashMap: Fastest (O(1))

- LinkedHashMap: Slightly slower due to order maintenance

- TreeMap: Slower (O(log n)) due to tree structure

• Null Keys:

- HashMap: Allows one null key

- LinkedHashMap: Allows one null key

- TreeMap: Does not allow null keys

• Use Case:

- HashMap: Fast lookup

- LinkedHashMap: Predictable iteration order

- TreeMap: Sorted data

Real-world Industrial Examples

HashMap

Use Case: Caching user sessions in a web application

```
Map<String, Session> sessionCache = new HashMap<>();
sessionCache.put("user123", new Session(...));
```

LinkedHashMap

Use Case: Maintaining recent search history in a UI

```
Map<String, String> recentSearches = new LinkedHashMap<>();
recentSearches.put("1", "Java Collections");
recentSearches.put("2", "Spring Boot");
```

TreeMap

Use Case: Storing stock prices by timestamp

```
Map<Long, Double> stockPrices = new TreeMap<>();
stockPrices.put(1620000000000L, 150.25);
stockPrices.put(1620003600000L, 152.75);
```

Common Methods

HashMap Methods

• put(K key, V value): Adds a key-value pair

• get(Object key): Retrieves the value for a key

• remove(Object key): Removes the key-value pair

• containsKey(Object key): Checks if key exists

• containsValue(Object value): Checks if value exists

• keySet(): Returns a set of keys

• values(): Returns a collection of values

• entrySet(): Returns a set of key-value pairs

LinkedHashMap Methods

Same as HashMap, with predictable iteration order.

• getFirstEntry(): Not available

• getLastEntry(): Not available

• Maintains insertion order during iteration

TreeMap Methods

• put(K key, V value): Adds a key-value pair

• get(Object key): Retrieves the value for a key

• remove(Object key): Removes the key-value pair

• firstKey(): Returns the first (lowest) key

• lastKey(): Returns the last (highest) key

• headMap(K toKey): Returns a view of the portion of this map whose keys are strictly less than toKey

• tailMap(K fromKey): Returns a view of the portion of this map whose keys are greater than or equal to fromKey

• subMap(K fromKey, K toKey): Returns a view of the portion of this map whose keys range from fromKey to toKey

THREAD SAFE COLEECTIONS

In Java, thread-safe collections ensure safe access by multiple threads without causing data inconsistency. Here are some thread-safe collections:

1. Legacy Thread-Safe Collections

- **`Vector`**: Synchronized alternative to `ArrayList`.
- **`Hashtable`**: Synchronized alternative to `HashMap`.

2. Collections from `java.util.concurrent` Package

- **`ConcurrentHashMap`**: A highly efficient thread-safe alternative to `Hashtable`.
- **`CopyOnWriteArrayList`**: Thread-safe alternative to `ArrayList`, suitable for scenarios with more reads than writes.
- **`CopyOnWriteArraySet`**: Thread-safe alternative to `HashSet`.
- **`ConcurrentSkipListMap`**: Thread-safe alternative to `TreeMap`.
- **`ConcurrentSkipListSet`**: Thread-safe alternative to `TreeSet`.

3. Synchronized Wrappers (from `Collections` Utility Class)

- **`Collections.synchronizedList(List)`**: Wraps a `List` to make it thread-safe.
- **`Collections.synchronizedSet(Set)`**: Wraps a `Set` to make it thread-safe.
- **`Collections.synchronizedMap(Map)`**: Wraps a `Map` to make it thread-safe.

Notes:

- **Performance Consideration**: Legacy classes like `Vector` and `Hashtable` are less efficient compared to modern alternatives like `ConcurrentHashMap`.
- **Use Case**: Choose the appropriate collection based on your application's read/write ratio and performance needs.

Iterator in Java Collections

An Iterator is an object that enables you to traverse through a collection, one element at a time. It is part of the java.util package and is commonly used with collections like List, Set, and Map.

Common Iterator Methods:

- hasNext(): Returns true if there are more elements to iterate.

- next(): Returns the next element in the iteration.

- remove(): Removes the last element returned by the iterator.

Example:

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

public class IteratorExample {
   public static void main(String[] args) {
      ArrayList<String> items = new ArrayList<>();
      items.add("Apple");
      items.add("Banana");
      items.add("Cherry");

      Iterator<String> iterator = items.iterator();
      while (iterator.hasNext()) {
         String item = iterator.next();
         System.out.println(item);
      }
   }
}
```

Iterating Over a Map in Java

1. Using entrySet()

This is the most common and efficient way to iterate over both keys and values in a Map.

```
Map<String, Integer> map = new HashMap<>();
map.put("A", 1);
map.put("B", 2);
map.put("C", 3);

for (Map.Entry<String, Integer> entry : map.entrySet()) {
```

```
    System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());
}
```

2. Using keySet()

Useful when you only need keys or want to access values via keys.

```
for (String key : map.keySet()) {
    System.out.println("Key: " + key + ", Value: " + map.get(key));
}
```

3. Using values()

Use this when you only care about the values.

```
for (Integer value : map.values()) {
    System.out.println("Value: " + value);
}
```

4. Using Iterator with entrySet()

Allows removal of entries during iteration.

```
Iterator<Map.Entry<String, Integer>> iterator =
map.entrySet().iterator();

while (iterator.hasNext()) {
    Map.Entry<String, Integer> entry = iterator.next();
    System.out.println("Key: " + entry.getKey() + ", Value: " +
entry.getValue());

    if (entry.getKey().equals("B")) {
        iterator.remove();
    }
}
```

5. Using Iterator with keySet()

Iterate over keys and access values.

```
Iterator<String> keyIterator = map.keySet().iterator();

while (keyIterator.hasNext()) {
    String key = keyIterator.next();
```

```
    System.out.println("Key: " + key + ", Value: " + map.get(key));
}
```

6. Using Java 8+ forEach with Lambda

Concise and modern approach using lambda expressions.

```
map.forEach((key, value) -> {
    System.out.println("Key: " + key + ", Value: " + value);
});
```

Comparable vs Comparator in Java

Definitions

Comparable

Comparable is an interface in Java used to define the natural ordering of objects. It is implemented by the class itself and overrides the compareTo() method.

- Belongs to the object itself.
- Used when a class has a **natural ordering**.
- Implements compareTo() method.
- Only one sorting logic per class.

Example:

```
public class Employee implements Comparable<Employee> {
    int id;
    String name;

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

    @Override
    public int compareTo(Employee other) {
        return this.id - other.id; // Natural order by ID
```

```
    }
}
```

Comparator

Comparator is an interface in Java used to define custom ordering of objects. It is implemented externally and overrides the compare() method.

Example:

```
public class NameComparator implements Comparator<Employee>
{
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.name.compareTo(e2.name); // Custom order by name
    }
}
```

Real-World Usage in Projects

- Sorting employee records by ID (Comparable) or by name/salary (Comparator).
- Sorting products in an e-commerce app by price, rating, or popularity.
- Sorting files in a file management system by name, size, or date.
- Sorting players or students in a leaderboard by score or completion time.

Complete Example

```
import java.util.*;

class Employee implements Comparable<Employee> {
    int id;
    String name;

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

    @Override
    public int compareTo(Employee other) {
        return this.id - other.id;
    }
```

```
    @Override
    public String toString() {
        return id + " - " + name;
    }
}

class NameComparator implements Comparator<Employee> {
    @Override
    public int compare(Employee e1, Employee e2) {
        return e1.name.compareTo(e2.name);
    }
}

public class Main {
    public static void main(String[] args) {
        List<Employee> list = new ArrayList<>();
        list.add(new Employee(3, "Alice"));
        list.add(new Employee(1, "Charlie"));
        list.add(new Employee(2, "Bob"));

        // Sort using Comparable (by ID)
        Collections.sort(list);
        System.out.println("Sorted by ID: " + list);

        // Sort using Comparator (by Name)
        Collections.sort(list, new NameComparator());
        System.out.println("Sorted by Name: " + list);
    }
}
```