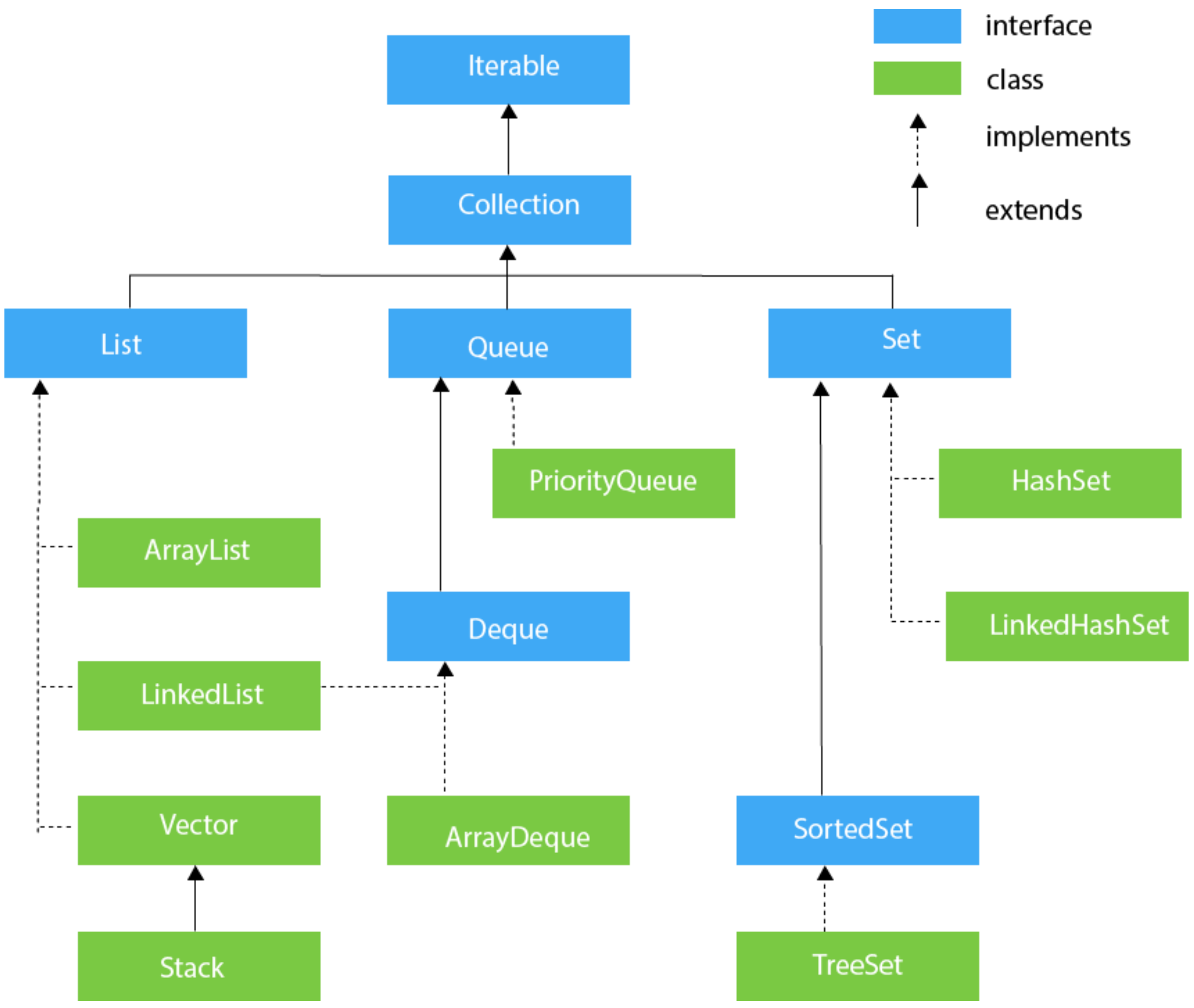
ItsRunTym

JAVA Collection Framework

50 interview questions/answers

Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The java.util package contains all the classes and interfaces for the Collection framework.



1. What is the Java Collection Framework? Explain its core interfaces.

Explanation: The Java Collection Framework (JCF) provides a set of classes and interfaces to handle collections of objects. It includes interfaces like Collection, List, Set, Queue, and Map, and their implementations such as ArrayList, HashSet, LinkedList, and HashMap. These interfaces and classes provide a standard way to store, access, and manipulate collections.

Example:

java

```
import java.util.*;

public class CollectionFrameworkExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Apple");
        list.add("Banana");

        Set<String> set = new HashSet<>();
        set.add("Orange");
        set.add("Apple");

        Map<String, Integer> map = new HashMap<>();
        map.put("Key1", 1);
        map.put("Key2", 2);

        System.out.println("List: " + list);
        System.out.println("Set: " + set);
        System.out.println("Map: " + map);
    }
}
```

2. What are the differences between Collection and Collections in Java?

Explanation: Collection is a root interface in the Java Collection Framework that represents a group of objects. Collections, on the other hand, is a utility class providing static methods to operate on or return collections (e.g., sorting, searching).

Example:

java

```java
import java.util.*;

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

        // Collections utility class
        Collections.sort(list);
        System.out.println("Sorted List: " + list);

        // Collection interface
        Collection<String> collection = new ArrayList<>(list);
        System.out.println("Collection Size: " + collection.size());
    }
}
```

3. How does ArrayList differ from LinkedList? When would you use one over the other?

Explanation: ArrayList is backed by a dynamic array, providing fast random access and slower insertions/removals in the middle. LinkedList is backed by a doubly linked list, providing faster insertions/removals but slower access time.

Example:

java

```java
import java.util.*;
```

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

        List<String> linkedList = new LinkedList<>();
        linkedList.add("Dog");
        linkedList.add("Elephant");
        linkedList.add("Fox");

        // Random access
        System.out.println("ArrayList get(1): " + arrayList.get(1)); // Fast access

        // Insertion/removal
        linkedList.add(2, "Giraffe");
        linkedList.remove("Elephant");
        System.out.println("LinkedList after modifications: " + linkedList);
    }
}
```

4. Explain the concept of a Set in Java and its implementations.

Explanation: A Set is a collection that does not allow duplicate elements. It is implemented by HashSet, LinkedHashSet, and TreeSet.

- **HashSet**: Uses a hash table, does not guarantee any order.
- **LinkedHashSet**: Maintains insertion order using a linked list.
- **TreeSet**: Implements NavigableSet and sorts elements according to their natural ordering or a comparator.

Example:

java

```
import java.util.*;


public class SetExamples {
    public static void main(String[] args) {
        Set<String> hashSet = new HashSet<>(Arrays.asList("One", "Two", "Three"));
        Set<String> linkedHashSet = new LinkedHashSet<>(Arrays.asList("A", "B", "C"));
        Set<String> treeSet = new TreeSet<>(Arrays.asList("X", "Y", "Z"));


        System.out.println("HashSet: " + hashSet);
        System.out.println("LinkedHashSet: " + linkedHashSet);
        System.out.println("TreeSet: " + treeSet);
    }
}
```

5. What is the difference between a List and a Set?

Explanation: A List is an ordered collection that allows duplicate elements and maintains insertion order. A Set is a collection that does not allow duplicate elements and does not guarantee any specific order (except LinkedHashSet and TreeSet).

Example:

java

```
import java.util.*;


public class ListVsSet {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("Apple", "Banana", "Apple"));
        Set<String> set = new HashSet<>(Arrays.asList("Apple", "Banana", "Apple"));


        System.out.println("List: " + list); // Allows duplicates
        System.out.println("Set: " + set);   // No duplicates
    }
}
```

6. How does HashSet ensure the uniqueness of elements?

Explanation: HashSet uses a hash table to store elements. Each element’s hash code is computed and used to place the element in a bucket. When checking for uniqueness, HashSet checks if the hash code matches and if the element is equal to any existing element in that bucket.

Example:

java

```
import java.util.*;


public class HashSetUniqueness {
    public static void main(String[] args) {
        Set<String> hashSet = new HashSet<>();
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Apple"); // Duplicate, will not be added


        System.out.println("HashSet: " + hashSet); // Output: [Apple, Banana]
    }
}
```

7. What is a Map in Java? How does it differ from a Collection?

Explanation: A Map is an object that maps keys to values, where each key is unique. It differs from a Collection as a Collection stores individual elements, while a Map stores key-value pairs.

Example:

java

```
import java.util.*;
```

```
public class MapVsCollection {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("Apple", 1);
        map.put("Banana", 2);

        Collection<Integer> values = map.values();

        System.out.println("Map: " + map);
        System.out.println("Values: " + values);
    }
}
```

8. Explain the working of HashMap in Java. How does it store and retrieve data?

Explanation: HashMap uses a hash table for storage. It computes a hash code for each key and uses it to determine the bucket location. Each bucket holds a linked list of entries to handle hash collisions. It provides constant-time complexity for basic operations like add, remove, and contains.

Example:

java

```
import java.util.*;

public class HashMapWorking {
    public static void main(String[] args) {
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("One", 1);
        hashMap.put("Two", 2);

        System.out.println("HashMap get(\"One\"): " + hashMap.get("One"));
        System.out.println("HashMap: " + hashMap);
    }
}
```

9. What are the main differences between HashMap and Hashtable?

Explanation:

- **Synchronization**: HashMap is not synchronized, while Hashtable is synchronized.
- **Null Keys/Values**: HashMap allows one null key and multiple null values. Hashtable does not allow null keys or values.
- **Performance**: HashMap generally performs better due to the lack of synchronization.

Example:

java

```
import java.util.*;

public class HashMapVsHashtable {
    public static void main(String[] args) {
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("Key1", 1);
        hashMap.put(null, 2); // Allowed

        Map<String, Integer> hashtable = new Hashtable<>();
        hashtable.put("Key1", 1);
        // hashtable.put(null, 2); // Throws NullPointerException

        System.out.println("HashMap: " + hashMap);
        System.out.println("Hashtable: " + hashtable);
    }
}
```

10. What is the difference between ArrayList and Vector?

Explanation:

- **Synchronization**: ArrayList is not synchronized, while Vector is synchronized.

- **Growth Policy**: ArrayList grows dynamically, while Vector doubles its size when more space is needed.
- **Performance**: ArrayList is generally faster than Vector due to lack of synchronization.

Example:

java

```
import java.util.*;

public class ArrayListVsVector {
   public static void main(String[] args) {
      List<String> arrayList = new ArrayList<>();
      arrayList.add("Apple");

      List<String> vector = new Vector<>();
      vector.add("Banana");

      System.out.println("ArrayList: " + arrayList);
      System.out.println("Vector: " + vector);
   }
}
```

11. How do you convert an array to a List in Java?

Explanation: You can use Arrays.asList() to convert an array to a List. This method returns a fixed-size list backed by the specified array.

Example:

java

```
import java.util.*;

public class ArrayToList {
   public static void main(String[] args) {
        String[] array = {"Apple", "Banana", "Cherry"};
        List<String> list = Arrays.asList(array);


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

12. What are fail-fast and fail-safe iterators?

Explanation:

- **Fail-fast**: These iterators detect changes to the collection while iterating (e.g., ArrayList, HashSet) and throw ConcurrentModificationException.
- **Fail-safe**: These iterators work on a clone of the collection (e.g., CopyOnWriteArrayList) and do not throw exceptions if the collection is modified.

Example:

java

```
import java.util.*;


public class FailFastVsFailSafe {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));


        try {
            for (String s : list) {
                list.add("D"); // Modifying the list during iteration
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("Fail-fast iterator detected modification.");
        }
        List<String> copyOnWriteList = new CopyOnWriteArrayList<>(Arrays.asList("A", "B",
"C"));
        for (String s : copyOnWriteList) {
            copyOnWriteList.add("D"); // Safe modification
        }
        System.out.println("CopyOnWriteArrayList: " + copyOnWriteList);
    }
}
```

13. What is the purpose of the Iterator interface in Java?

Explanation: The Iterator interface provides a way to traverse elements of a collection. It includes methods like hasNext(), next(), and remove() to iterate over and manipulate elements.

Example:

java

```
import java.util.*;

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

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

14. How does ConcurrentModificationException occur in collections?

Explanation: ConcurrentModificationException occurs when a collection is modified while iterating over it using an iterator, and the modification is detected by the iterator.

Example:

```
java


import java.util.*;


public class ConcurrentModificationExceptionExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));


        try {
            for (String s : list) {
                list.remove(s); // Modifying the list during iteration
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("ConcurrentModificationException occurred.");
        }
    }
}
```

15. What is the difference between Iterator and ListIterator?

Explanation:

- **Iterator**: Can traverse a collection in one direction (forward) and provides methods like hasNext(), next(), and remove().
- **ListIterator**: Extends Iterator and allows bidirectional traversal, modification, and accessing the current index. Provides methods like hasPrevious(), previous(), and add().

Example:

```
java


import java.util.*;


public class ListIteratorExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
        ListIterator<String> listIterator = list.listIterator();


        System.out.println("Forward iteration:");
        while (listIterator.hasNext()) {
            System.out.println(listIterator.next());
        }


        System.out.println("Backward iteration:");
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}
```

16. Explain the internal structure of LinkedHashMap. How does it maintain the insertion order?

Explanation: LinkedHashMap maintains a linked list of entries in addition to the hash table. This linked list preserves the insertion order of the keys, allowing predictable iteration order.

Example:

java

```
import java.util.*;


public class LinkedHashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put("One", 1);
        linkedHashMap.put("Two", 2);
        linkedHashMap.put("Three", 3);
        System.out.println("LinkedHashMap: " + linkedHashMap);
    }
}
```

17. What is the difference between TreeSet and HashSet?

Explanation:

- **TreeSet**: Implements NavigableSet and sorts elements according to their natural ordering or a comparator.
- **HashSet**: Uses a hash table, does not guarantee any specific order.

Example:

java

```
import java.util.*;

public class TreeSetVsHashSet {
    public static void main(String[] args) {
        Set<String> treeSet = new TreeSet<>(Arrays.asList("Banana", "Apple", "Cherry"));
        Set<String> hashSet = new HashSet<>(Arrays.asList("Banana", "Apple", "Cherry"));

        System.out.println("TreeSet (sorted): " + treeSet);
        System.out.println("HashSet (unordered): " + hashSet);
    }
}
```

18. Explain how a PriorityQueue works. What are its typical use cases?

Explanation: PriorityQueue is a queue that orders elements according to their natural ordering or a comparator. It is typically used when elements need to be processed in a priority order.

Example:

java

```
import java.util.*;


public class PriorityQueueExample {
   public static void main(String[] args) {
      Queue<Integer> priorityQueue = new PriorityQueue<>(Arrays.asList(3, 1, 4, 1, 5, 9));


      System.out.println("PriorityQueue: ");
      while (!priorityQueue.isEmpty()) {
         System.out.println(priorityQueue.poll()); // Elements are retrieved in priority order
      }
   }
}
```

19. What is a Queue in Java, and how does it differ from other collections?

Explanation: A Queue is a collection designed for holding elements prior to processing. It typically follows a FIFO (first-in-first-out) order. It differs from other collections like List and Set in its primary purpose and ordering of elements.

Example:

java

```
import java.util.*;


public class QueueExample {
   public static void main(String[] args) {
      Queue<String> queue = new LinkedList<>(Arrays.asList("A", "B", "C"));


      System.out.println("Queue (FIFO order):");
      while (!queue.isEmpty()) {
         System.out.println(queue.poll());
      }
   }
}
```

20. How does a Stack differ from other collection classes?

Explanation: Stack is a subclass of Vector that implements a last-in-first-out (LIFO) stack of objects. Unlike other collections, Stack allows elements to be pushed and popped according to LIFO order.

Example:

java

```
import java.util.*;

public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("A");
        stack.push("B");
        stack.push("C");

        System.out.println("Stack (LIFO order):");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

21. How does ConcurrentHashMap differ from HashMap?

Explanation: ConcurrentHashMap is a thread-safe variant of HashMap. It uses a segmented locking mechanism for improved concurrency, allowing multiple threads to read and write concurrently without locking the entire map.

Example:

java

```
import java.util.concurrent.*;
```

```
public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        ConcurrentMap<String, Integer> concurrentHashMap = new ConcurrentHashMap<>();
        concurrentHashMap.put("A", 1);
        concurrentHashMap.put("B", 2);

        System.out.println("ConcurrentHashMap: " + concurrentHashMap);
    }
}
```

22. What is the purpose of WeakHashMap in Java?

Explanation: WeakHashMap is a map implementation where keys are weakly referenced. This means that if a key is no longer referenced elsewhere, it may be garbage collected, allowing the map to automatically clean up unused entries.

Example:

java

```
import java.util.*;

public class WeakHashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> weakHashMap = new WeakHashMap<>();
        String key = new String("Key");
        weakHashMap.put(key, 1);

        System.out.println("WeakHashMap before GC: " + weakHashMap);
        key = null; // Make key eligible for GC
        System.gc(); // Suggest GC

        System.out.println("WeakHashMap after GC: " + weakHashMap);
    }
}
```

23. How does IdentityHashMap differ from HashMap?

Explanation: IdentityHashMap uses reference equality (i.e., ==) instead of object equality (i.e., equals()) for keys and values, unlike HashMap. This means that two distinct instances with the same content will be treated as different keys.

Example:

java

```
import java.util.*;

public class IdentityHashMapExample {
    public static void main(String[] args) {
        Map<String, Integer> identityHashMap = new IdentityHashMap<>();
        String key1 = new String("Key");
        String key2 = new String("Key");

        identityHashMap.put(key1, 1);
        identityHashMap.put(key2, 2);

        System.out.println("IdentityHashMap: " + identityHashMap); // Shows both keys
    }
}
```

24. What is CopyOnWriteArrayList, and how is it different from ArrayList?

Explanation: CopyOnWriteArrayList is a thread-safe variant of ArrayList that creates a new copy of the underlying array for each modification, ensuring thread safety without requiring synchronization.

Example:

java

```
import java.util.*;
import java.util.concurrent.*;


public class CopyOnWriteArrayListExample {
    public static void main(String[] args) {
        List<String> copyOnWriteList = new CopyOnWriteArrayList<>(Arrays.asList("A", "B",
"C"));
        copyOnWriteList.add("D");


        System.out.println("CopyOnWriteArrayList: " + copyOnWriteList);
    }
}
```

25. Explain the concept of NavigableMap in Java. How does it extend SortedMap?

Explanation: NavigableMap extends SortedMap and provides navigation methods to retrieve entries based on search criteria. It supports operations like lowerEntry(), floorEntry(), ceilingEntry(), and higherEntry().

Example:

java

```
import java.util.*;


public class NavigableMapExample {
    public static void main(String[] args) {
        NavigableMap<Integer, String> navigableMap = new TreeMap<>();
        navigableMap.put(1, "One");
        navigableMap.put(2, "Two");
        navigableMap.put(3, "Three");


        System.out.println("Floor Entry for 2: " + navigableMap.floorEntry(2));
        System.out.println("Ceiling Entry for 2: " + navigableMap.ceilingEntry(2));
    }
```

}

26. What are the differences between Queue, Deque, and BlockingQueue?

Explanation:

- **Queue**: Represents a collection designed for holding elements prior to processing in FIFO order.
- **Deque**: Double-ended queue that allows elements to be added or removed from both ends.
- **BlockingQueue**: Extends Queue to support blocking operations when the queue is full or empty.

Example:

java

```
import java.util.*;

import java.util.concurrent.*;


public class QueueTypes {

  public static void main(String[] args) {

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

    Deque<String> deque = new ArrayDeque<>();

    BlockingQueue<String> blockingQueue = new LinkedBlockingQueue<>();


    queue.offer("QueueElement");

    deque.offerFirst("DequeElement");

    blockingQueue.offer("BlockingQueueElement");


    System.out.println("Queue: " + queue);

    System.out.println("Deque: " + deque);

    System.out.println("BlockingQueue: " + blockingQueue);

  }

}
```

27. What is the role of Comparator and Comparable in sorting collections?

Explanation:

- **Comparable**: Interface that defines the natural ordering of objects. Implementing this interface allows objects to be sorted using Collections.sort() or Arrays.sort().
- **Comparator**: Interface that defines an external ordering of objects. It is used to sort collections in a custom order.

Example:

java

```
import java.util.*;

public class ComparableComparatorExample {
    static class Person implements Comparable<Person> {
        String name;
        int age;

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

        @Override
        public int compareTo(Person other) {
            return Integer.compare(this.age, other.age); // Natural order by age
        }

        @Override
        public String toString() {
            return name + " (" + age + ")";
        }
    }

    static class PersonComparator implements Comparator<Person> {
        @Override
        public int compare(Person p1, Person p2) {
            return p1.name.compareTo(p2.name); // Custom order by name
        }
    }

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
            new Person("John", 25),
            new Person("Jane", 22),
            new Person("Alice", 30)
        );

        Collections.sort(people); // Sorts by age
        System.out.println("Sorted by age: " + people);

        people.sort(new PersonComparator()); // Sorts by name
        System.out.println("Sorted by name: " + people);
    }
}
```

28. How does the hashCode() and equals() methods affect the behavior of collections?

Explanation:

- **hashCode()**: Determines the bucket in a hash-based collection like HashMap. Objects with the same hash code may end up in the same bucket.
- **equals()**: Determines object equality. Two objects with the same hash code but different equals() results will be stored separately.

Example:

java

```
import java.util.*;

public class HashCodeEqualsExample {
    static class Person {
        String name;

        Person(String name) {
            this.name = name;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (obj == null || getClass() != obj.getClass()) return false;
            Person person = (Person) obj;
            return Objects.equals(name, person.name);
        }

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

        @Override
        public String toString() {
            return name;
        }
    }


    public static void main(String[] args) {
        Set<Person> set = new HashSet<>();
        set.add(new Person("Alice"));
        set.add(new Person("Alice")); // Duplicate based on equals() and hashCode()


        System.out.println("HashSet: " + set);
    }
}
```

29. What is the load factor in a HashMap, and how does it affect performance?

Explanation: The load factor is a measure of how full a hash table is allowed to get before its capacity is automatically increased. A higher load factor means less space but more collisions, while a lower load factor means more space but less frequent resizing.

Example:

java

```
import java.util.*;


public class HashMapLoadFactor {
    public static void main(String[] args) {
        Map<String, Integer> hashMap = new HashMap<>(10, 0.75f); // Initial capacity 10, load
factor 0.75


        for (int i = 0; i < 15; i++) {
            hashMap.put("Key" + i, i);
        }


        System.out.println("HashMap: " + hashMap);
    }
```

}

30. What are the differences between synchronizedList and CopyOnWriteArrayList?

Explanation:

- **synchronizedList**: Wraps a list to make it synchronized. All operations are synchronized, which can affect performance.
- **CopyOnWriteArrayList**: Provides thread safety by making a new copy of the underlying array for each modification, allowing read operations to occur concurrently.

Example:

java

```
import java.util.*;
import java.util.concurrent.*;

public class SynchronizedListVsCopyOnWriteArrayList {
   public static void main(String[] args) {
       List<String> synchronizedList = Collections.synchronizedList(new
ArrayList<>(Arrays.asList("A", "B", "C")));
       List<String> copyOnWriteList = new CopyOnWriteArrayList<>(Arrays.asList("A", "B",
"C"));

       // Example operations
       synchronizedList.add("D");
       copyOnWriteList.add("D");

       System.out.println("SynchronizedList: " + synchronizedList);
       System.out.println("CopyOnWriteArrayList: " + copyOnWriteList);
   }
}
```

31. How can you make a Collection thread-safe in Java?

Explanation: You can make a collection thread-safe using methods from Collections (e.g., synchronizedList(), synchronizedMap()) or using concurrent collections like ConcurrentHashMap or CopyOnWriteArrayList.

Example:

java

```
import java.util.*;

import java.util.concurrent.*;


public class ThreadSafeCollections {

   public static void main(String[] args) {

      List<String> synchronizedList = Collections.synchronizedList(new
ArrayList<>(Arrays.asList("A", "B", "C")));

      ConcurrentMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();

      concurrentMap.put("Key1", 1);

      concurrentMap.put("Key2", 2);


      System.out.println("SynchronizedList: " + synchronizedList);

      System.out.println("ConcurrentMap: " + concurrentMap);

   }

}
```

32. What are Set and its different implementations? How do they differ from List?

Explanation:

- **Set**: A collection that does not allow duplicate elements. Implementations include HashSet (unordered), LinkedHashSet (ordered by insertion), and TreeSet (sorted).
- **List**: A collection that allows duplicate elements and maintains insertion order. Implementations include ArrayList and LinkedList.

Example:

java

```
import java.util.*;
```

```
public class SetVsList {
    public static void main(String[] args) {
        Set<String> hashSet = new HashSet<>(Arrays.asList("A", "B", "C"));
        List<String> arrayList = new ArrayList<>(Arrays.asList("A", "B", "C"));

        System.out.println("HashSet (no duplicates, unordered): " + hashSet);
        System.out.println("ArrayList (duplicates allowed, ordered): " + arrayList);
    }
}
```

33. What is the role of Collections utility class in Java?

Explanation: The Collections utility class provides static methods for operating on collections, such as sorting, searching, and shuffling. It also provides methods for creating synchronized and unmodifiable collections.

Example:

java

```
import java.util.*;

public class CollectionsUtility {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("Banana", "Apple", "Cherry"));
        Collections.sort(list); // Sorting the list
        System.out.println("Sorted list: " + list);

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

        List<String> unmodifiableList = Collections.unmodifiableList(list);
        System.out.println("Unmodifiable list: " + unmodifiableList);
    }
}
```

34. What is the difference between Collections.emptyList() and Collections.singletonList()?

Explanation:

- **Collections.emptyList()**: Returns an immutable empty list.
- **Collections.singletonList()**: Returns an immutable list containing a single element.

Example:

java

```
import java.util.*;

public class EmptyAndSingletonList {
    public static void main(String[] args) {
        List<String> emptyList = Collections.emptyList();
        List<String> singletonList = Collections.singletonList("SingleElement");

        System.out.println("Empty List: " + emptyList);
        System.out.println("Singleton List: " + singletonList);
    }
}
```

35. How does TreeMap work, and what is its typical use case?

Explanation: TreeMap implements NavigableMap and is sorted according to the natural ordering of its keys or a comparator. It provides efficient log(n) time complexity for get, put, and remove operations. It is typically used when a sorted map is needed.

Example:

java

```
import java.util.*;

public class TreeMapExample {
    public static void main(String[] args) {
        Map<String, Integer> treeMap = new TreeMap<>();
        treeMap.put("Banana", 2);
        treeMap.put("Apple", 1);
        treeMap.put("Cherry", 3);

        System.out.println("TreeMap (sorted by key): " + treeMap);
    }
}
```

36. What is EnumSet and when would you use it?

Explanation: EnumSet is a specialized Set implementation for use with enum types. It is highly efficient, using bit vectors internally to represent the set of enum values. It is ideal when working with enum types and you need a set-like structure.

Example:

java

```
import java.util.*;

public class EnumSetExample {
    enum Color {
        RED, GREEN, BLUE
    }

    public static void main(String[] args) {
        EnumSet<Color> colorSet = EnumSet.of(Color.RED, Color.GREEN);

        System.out.println("EnumSet: " + colorSet);
    }
}
```

37. What is the difference between ArrayDeque and LinkedList?

Explanation:

- **ArrayDeque**: Implements Deque and provides a resizable array implementation, which is more efficient for stack and queue operations compared to LinkedList.
- **LinkedList**: Implements both List and Deque and uses a doubly-linked list internally, which can be less efficient for stack and queue operations compared to ArrayDeque.

Example:

java

```
import java.util.*;

public class ArrayDequeVsLinkedList {
    public static void main(String[] args) {
        Deque<String> arrayDeque = new ArrayDeque<>(Arrays.asList("A", "B", "C"));
        Deque<String> linkedList = new LinkedList<>(Arrays.asList("A", "B", "C"));

        System.out.println("ArrayDeque: " + arrayDeque);
        System.out.println("LinkedList: " + linkedList);
    }
}
```

38. Explain how TreeSet maintains order.

Explanation: TreeSet maintains order by storing elements in a red-black tree, which is a self-balancing binary search tree. This ensures that elements are sorted according to their natural ordering or a provided comparator.

Example:

java

```
import java.util.*;

public class TreeSetOrder {
    public static void main(String[] args) {
        Set<String> treeSet = new TreeSet<>(Arrays.asList("Banana", "Apple", "Cherry"));
        System.out.println("TreeSet (sorted): " + treeSet);
    }
}
```

39. How does LinkedHashMap maintain insertion order?

Explanation: LinkedHashMap maintains insertion order by using a linked list to keep track of the order of entries. The entries are stored in a hash table and linked list, which preserves the order in which keys were added.

Example:

java

```
import java.util.*;

public class LinkedHashMapOrder {
    public static void main(String[] args) {
        Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put("One", 1);
        linkedHashMap.put("Two", 2);
        linkedHashMap.put("Three", 3);

        System.out.println("LinkedHashMap (insertion order): " + linkedHashMap);
    }
}
```

40. How can you synchronize a Map in Java?

Explanation: You can synchronize a Map using Collections.synchronizedMap() which wraps the map with a synchronized view.

Example:

java

```
import java.util.*;
```

```
public class SynchronizedMapExample {
    public static void main(String[] args) {
        Map<String, Integer> synchronizedMap = Collections.synchronizedMap(new
HashMap<>());
        synchronizedMap.put("Key1", 1);
        synchronizedMap.put("Key2", 2);

        System.out.println("SynchronizedMap: " + synchronizedMap);
    }
}
```

41. What is the purpose of the ConcurrentMap interface?

Explanation: ConcurrentMap is an interface that extends Map and provides additional methods for concurrency control, such as putIfAbsent(), remove(), and replace() which are atomic and thread-safe.

Example:

java

```
import java.util.concurrent.*;

public class ConcurrentMapExample {
    public static void main(String[] args) {
        ConcurrentMap<String, Integer> concurrentMap = new ConcurrentHashMap<>();
        concurrentMap.put("Key1", 1);
        concurrentMap.putIfAbsent("Key1", 2); // No effect, already present

        System.out.println("ConcurrentMap: " + concurrentMap);
    }
}
```

42. How do you use Collections.sort() to sort a list of objects?

Explanation: You can use Collections.sort() to sort a list of objects if the objects implement Comparable, or you can pass a Comparator to define custom sorting.

Example:

java

```
import java.util.*;

public class CollectionsSortExample {
    static class Person implements Comparable<Person> {
        String name;
        int age;

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

        @Override
        public int compareTo(Person other) {
            return Integer.compare(this.age, other.age);
        }

        @Override
        public String toString() {
            return name + " (" + age + ")";
        }
    }

    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
            new Person("John", 25),
            new Person("Jane", 22),
            new Person("Alice", 30)
        );

        Collections.sort(people); // Sorts by natural ordering (age)
        System.out.println("Sorted by age: " + people);
    }
}
```

43. What is the purpose of Collections.unmodifiableCollection()?

Explanation: Collections.unmodifiableCollection() provides a read-only view of the specified collection. It is used to create immutable collections that prevent modifications.

Example:

java

```
import java.util.*;

public class UnmodifiableCollectionExample {
    public static void main(String[] args) {
        List<String> list = Arrays.asList("A", "B", "C");
        Collection<String> unmodifiableList = Collections.unmodifiableCollection(list);

        System.out.println("Unmodifiable Collection: " + unmodifiableList);

        // The following line will throw UnsupportedOperationException
        // unmodifiableList.add("D");
    }
}
```

44. What are Collections.synchronizedList() and its usage?

Explanation: Collections.synchronizedList() returns a synchronized (thread-safe) list backed by the specified list. It ensures that all operations on the list are thread-safe.

Example:

java

```
import java.util.*;

public class SynchronizedListExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(Arrays.asList("A", "B", "C"));
        List<String> synchronizedList = Collections.synchronizedList(list);

        synchronizedList.add("D");
        System.out.println("Synchronized List: " + synchronizedList);
    }
}
```

45. What is the difference between ArrayList and Vector?

Explanation:

- **ArrayList**: Implements the List interface with a dynamically resizable array. It is not synchronized.
- **Vector**: Implements the List interface with a dynamically resizable array but is synchronized.

Example:

java

```
import java.util.*;

public class ArrayListVsVector {
    public static void main(String[] args) {
        List<String> arrayList = new ArrayList<>(Arrays.asList("A", "B", "C"));
        List<String> vector = new Vector<>(Arrays.asList("A", "B", "C"));


        System.out.println("ArrayList: " + arrayList);
        System.out.println("Vector: " + vector);
    }
}
```

46. What are Collections.nCopies() and its purpose?

Explanation: Collections.nCopies() returns an immutable list consisting of n copies of the specified object. It is useful when you need a fixed-size list where all elements are the same.

Example:

java

```
import java.util.*;


public class CollectionsNCopiesExample {
    public static void main(String[] args) {
        List<String> list = Collections.nCopies(5, "RepeatedElement");


        System.out.println("List of copies: " + list);
    }
}
```

47. What is PriorityQueue and how is it different from LinkedList?

Explanation: PriorityQueue is a queue where elements are ordered based on their priority. It does not guarantee a FIFO order like LinkedList but instead provides a way to access the highest-priority element.

Example:

java

```
import java.util.*;
```

```
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<>(Arrays.asList(5, 1, 3, 2,
4));

        System.out.println("PriorityQueue (min-heap): " + priorityQueue.poll()); // Retrieves
and removes the highest priority element
    }
}
```

48. How does ConcurrentSkipListMap work, and what is its use case?

Explanation: ConcurrentSkipListMap is a concurrent, navigable map implemented using a skip list. It provides high concurrency and thread-safe operations for maintaining sorted mappings.

Example:

java

```
import java.util.concurrent.*;

public class ConcurrentSkipListMapExample {
    public static void main(String[] args) {
        ConcurrentSkipListMap<String, Integer> skipListMap = new ConcurrentSkipListMap<>();
        skipListMap.put("One", 1);
        skipListMap.put("Two", 2);

        System.out.println("ConcurrentSkipListMap: " + skipListMap);
    }
}
```

49. What are EnumMap and its advantages?

Explanation: EnumMap is a specialized Map implementation for use with enum keys. It is efficient in terms of performance and space and maintains the natural order of enum constants.

Example:

java

```
import java.util.*;

public class EnumMapExample {
    enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

    public static void main(String[] args) {
        EnumMap<Day, String> enumMap = new EnumMap<>(Day.class);
        enumMap.put(Day.MONDAY, "Start of the week");
        enumMap.put(Day.FRIDAY, "End of the work week");

        System.out.println("EnumMap: " + enumMap);
    }
}
```

50. How does WeakHashMap handle garbage collection?

Explanation: WeakHashMap uses weak references for its keys, meaning that if a key is no longer referenced elsewhere, it can be collected by the garbage collector, and its entry in the WeakHashMap will be removed.

Example:

java

```
import java.util.*;

public class WeakHashMapExample {
    public static void main(String[] args) {
        Map<Object, String> weakHashMap = new WeakHashMap<>();
        Object key = new Object();
        weakHashMap.put(key, "Value");
        System.out.println("WeakHashMap before GC: " + weakHashMap);

        key = null; // Make key eligible for GC
        System.gc(); // Suggest GC

        System.out.println("WeakHashMap after GC: " + weakHashMap);
    }
}
```