CHAPTER 12

■ ■ ■

Collections

QUESTIONS AND EXERCISES

1. What is the Collections framework?

Answer:

Java Collections Framework consists of interfaces, implementation classes, and some utility classes to handle wide range of collection types such as lists, queues, sets, and maps.

2. What is the name of the interface that all collections in the Collections Framework, except maps, implement?

Answer:

```
java.util.Collection<E>
```

3. List the names of different types of operations that you can perform on collections in the Collection Framework.

Answer:

- Basic operations: Getting size, adding a new element, removing an element, etc.
- Bulk (or group) operations: Removing all elements, checking if a collection contains all elements from another collection, etc.
- Aggregate operations: Computing sum of all elements, etc.
- Array operations: Converting a collection into an array.
- Comparison operations: Comparing two collections for equality.

4. What method in the `Collection<E>` interface lets you obtain the size of the collection?

   Answer:

   The `size()` method returns number of elements or size of the collection.

5. What methods in the `Collection<E>` interface let you remove all elements in the collection in one go and one element at a time?

   Answer:

   The `removeAll()` and `remove()` methods, respectively.

6. What method in the `Collection<E>` interface would you use to check that the collection contains a given object?

   Answer:

   The `contains()` method.

7. How do you check if a collection is empty?

   Answer:

   Use the `isEmpty()` method. It returns `true` if collection is empty and `false` otherwise.

8. Name the method in the `Collection<E>` interface that lets you convert a collection to an array.

   Answer:

   The `toArray()` method.

9. Enumerate three ways to iterate over elements of a collection. Can you use a simple `for` loop statement to iterate over the elements of a collection? What is a fail-fast iterator?

Answer:

Three ways to iterate a collection are:

- Using an Iterator
- Using a for-each loop
- Using the `forEach()` method

Yes. A simple `for` loop can be used to iterate over elements of collections if collections allow index based access.

In a fast-fail iterator, if the collection is modified by any means, except using the `remove()` method of the same iterator after the iterator is obtained, the attempt to access the next element using the iterator will throw a `ConcurrentModificationException`.

10. Java supports mathematical sets, sorted sets, and navigable sets. Differentiate between the three types of sets and name the interfaces and at least one implementation class for those interfaces representing these three types of sets.

Answer:

A set is a collection of unique objects and supports union, intersection and difference operations. In mathematical sets, the ordering of elements is irrelevant. A sorted set imposes ordering on its elements. A navigable set is a specialized sorted set that lets you work with its subsets in a variety of ways.

The `Set` interface represents a mathematical set and `HashSet` is its implementation class. The `SortedSet` interface represents a sorted set and `TreeSet` is its implementation class. The `NavigableSet` interface represents a navigable set and `TreeSet` is its implementation class.

11. How do you traverse the elements in a set?

Answer:

An iterator, a for-each loop, or the `forEach()` method of the `Set` can be used to traverse the elements in a set.

12. Consider the following two immutable sets of integers:

```
Set<Integer> s1 = Set.of(10, 20, 30, 40);
Set<Integer> s2 = Set.of(10, 15, 20, 25, 30);
```

Write a snippet of code to print the union, intersection, and difference of the two sets, `s1` and `s2`, as computed in mathematics.

Solution:

```
Set<Integer> s1 = Set.of(10, 20, 30, 40);
Set<Integer> s2 = Set.of(10, 15, 20, 25, 30);

Set<Integer> s1UnionS2 = new HashSet(s1);
s1UnionS2.addAll(s2);  // s1UnionS2 is the union of s1 and s2
System.out.println("s1 union s2: " + s1UnionS2);

Set<Integer> s1IntersectS2 = new HashSet(s1);
s1IntersectS2.retainAll(s2); // s1IntersectS2 is the intersection of s1 and s2
System.out.println("s1 intersection s2: " + s1IntersectS2);

Set<Integer> s1DifferenceS2 = new HashSet<>(s1);
s1DifferenceS2.removeAll(s2);
System.out.println("s1 difference s2: " + s1DifferenceS2);
```

13. Spot the problem with the following snippet of code that attempts to create an immutable set of integers:

```
Set<Integer> s1 = Set.of(20, 10, 30, 10);
```

Answer:

The code attempts to add a duplicate element, 10, which will cause an `IllegalArgumentException` at runtime.

14. Consider the following snippet of code:

```
Set<Integer> s1 = Set.of();
System.out.println("s1.isEmpty(): " + s1.isEmpty());
s1.add(2018);
System.out.println("s1.isEmpty(): " + s1.isEmpty());
```

Will this code compile? If your answer is yes, what will be the output? If you

think the code will compile, but will throw a runtime exception, describe the reason for the exception and a way to fix the problem.

Answer:

Yes, the code will compile, but an UnsupportedOperationException is thrown at runtime. The static `of()` method of the `Set` interface creates an immutable `Set`. Once created, the set cannot be modified. There are two options to fix this problem. If element 2018 is not needed in set `s1`, remove line `s1.add(2018)`. Alternately, initialize set `s1` with parameter 2018 to the `of()` method.

15. In your application, you need to work with a sorted range of unique integers. Which interface and implementation class of the set collection family will you use for this purpose?

Answer:

Will use `NavigableSet` interface and `TreeSet<E>` class.

16. How do you sort the elements in a set in natural order and custom order?

Answer:

If elements of a `SortedSet` implement the `Comparable` interface, the `SortedSet` will use the `compareTo()` method of elements to sort them. Custom sorting is achieved by specifying a `Comparator`.

17. What is the difference between a `List` and a `Set`?

Answer:

Unlike a `Set`, a `List` allows duplicate elements and accessing elements by their indexes. A `List` maintains ordering of its elements whereas ordering of elements is not maintained in a `Set`.

18. Consider the following incomplete snippet of code:

```
List<String> list = List.of("Li", "Xi", "Bo", "Da", "Fa", "Bo");
int i1 = /* your code goes here */;
int i2 = /* your code goes here */;
System.out.printf("First and last indexes of Bo are %d and %d.%n",
        i1, i2);
```

Complete this snippet of code to print the first and the last indexes of the element `"Bo"`. The output should be as follows:

```
First and last indexes of Bo are 2 and 5.
```

Answer:

```
List<String> list = List.of("Li", "Xi", "Bo", "Da", "Fa", "Bo");
int i1 = list.indexOf("Bo");
int i2 = list.lastIndexOf("Bo");
System.out.printf("First and last indexes of Bo are %d and %d.%n", i1, i2);
```

19. What is the difference between an `Iterator` and `ListIterator`? Can you use an `Iterator` to traverse elements in a `List`?

Answer:

An `Iterator` is forward-only and it cannot be reused. A `ListIterator` is a special type of `Iterator` to iterate over elements of a list in both direction at the same time.

Yes, `Iterator` can be used to traverse `List` elements in forward direction.

20. Suppose you need to use a list in your program in which you need to frequently insert and remove elements from the beginning of the list. What implementation class of the `List` interface would you choose to achieve this?

Answer:

`LinkedList<E>`

21. The following snippet of code contains a logical error. Describe the error.

```
List<Integer> list = new ArrayList<>();
list.add(0, 0);
list.add(1, 10);
list.add(2, 20);
list.add(5, 50);
System.out.println(list);
```

Answer:

The first argument to the `add()` method must be between 0 and the size of the `List`. At line #5, since the first argument 5 is greater than 3 (size of the list), an `IndexOutOfBoundsException` is thrown.

22. Consider the following snippet of incomplete code:

```
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);

System.out.println(list);

/* your code goes here */

System.out.println(list);
```

Complete this snippet of code so that each element in the list is replaced by a value, which is double the current value. You are encouraged to use the following `replaceAll()` method of the `List<E>` interface to achieve this:

```
default void replaceAll(UnaryOperator<E> operator)
```

The expected output is as follows:

```
[10, 20, 30]
[20, 40, 60]
```

Answer:

```
List<Integer> list = new ArrayList<>();

list.add(10);
list.add(20);
```

```
list.add(30);

System.out.println(list);
list.replaceAll(e -> e * 2);
System.out.println(list);
```

23. Name the interface whose instances represent simple queues in a Java program.

Answer:

The `Queue<E>` interface.

24. What is the difference between the FIFO and LIFO queues? Name the implementation class that implements a simple FIFO and LIFO queue.

Answer:

In a FIFO (or First-In-First-Out) queue, the element entering the queue first will leave the queue first. The `LinkedList<E>` class can be used as a `FIFO` queue.

In a LIFO (or Last-In-First-Out) queue, the head (exit point) and tail (entry point) are the same; the element entering the queue first leaves the queue last. An instance of the `ArrayDeque` class or the `LinkedList<E>` class can be used as a `LIFO` queue when used as `Deque`.

25. What is the difference between using the `add(E e)` and `offer(E e)` methods of the `Queue<E>` interface to insert elements to the queue?

Answer:

If the element cannot be added, the `add(E e)` method throws an `IllegalStateException` whereas the `offer(E e)` method returns `false`.

26. What is a priority queue? Name the implementation class for priority queues in Java.

Answer:

A priory queue is a queue in which each element has an associated priority. The element with the highest priority is removed next from the queue. Java provides `PriorityQueue<E>` as an implementation class for an unbounded priority queue. You can use natural order of the elements of the queue as its priority. In this case, the elements of the queue must implement the `Comparable` interface. You can also supply a `Comparator`, which will determine the priority order of the elements. When you add a new element to a priority queue, it is positioned in the queue based on its priority. How the priority is decided in the queue is up to you to implement.

27. Write a complete program that uses a priority queue to store names of a few people. The output of your program should demonstrate that a person with a shorter name has a higher priority in the queue. Add a few names to the queue and remove them one at a time. Print the removed elements and the remaining elements in the queue every time you remove an element.

Answer:

```
// PriorityQueueTest.java
package com.jdojo.collections.exercises;

import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

public class PriorityQueueTest {
    public static void main(String[] args) {
        Comparator<String> comparator = Comparator.comparing(String::length);

        Queue<String> pq = new PriorityQueue<>(comparator);
        pq.add("Richard");
        pq.add("John");
        pq.add("Ken");
        pq.add("Li");
        pq.add("Donna");
        System.out.println("Priority queue: " + pq);

        while (!pq.isEmpty()) {
            System.out.println("Head element: " + pq.peek());
            pq.remove();
            System.out.println("Removed one element from the queue.");
            System.out.println("Priority queue: " + pq);
        }
    }
}
```

28. What is the difference between a `Queue` and a `Deque`? Name two implementation classes of the `Deque<E>` interface.

Answer:

A `Deque` allows insertion and removal of elements from both ends whereas a `Queue` allows insertion and removal from one end only. The `ArrayDeque<E>` and `LinkedList<E>` classes are two implementation classes for the `Deque<E>` interface.

29. Can you use a `Deque` to represent a stack? If your answer is yes, demonstrate it with an example.

Answer:

Yes. You can use a `Deque` as aa stack.

```
// DequeAsStack.java
package com.jdojo.collections.exercises;

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

public class DequeAsStack {
    public static void main(String[] args) {
        // Create a Deque and use it as stack
        Deque<String> deque = new ArrayDeque<>();
        deque.push("Apple");
        deque.push("Banana");
        deque.push("Orange");
        deque.push("Plum");

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

        // Let's remove all elements from the Deque
        while (!deque.isEmpty()) {
            System.out.println("Popped: " + deque.pop());
            System.out.println("Stack: " + deque);
        }

        System.out.println("Stack is empty: " + deque.isEmpty());
    }
}
```

30. What is a blocking queue? Name the interface whose instances represent blocking queues. What is fairness of a blocking queue?

Answer:

A blocking queue extends the behavior of a queue in dealing with extreme cases like adding an element to full queue and removing element from empty queue. Such operations on a blocking queue may block until the operation may succeed or may block with a timeout.

The interface whose instances represent blocking queues is `BlockingQueue<E>`.

Fairness is used to handle situations where multiple threads are blocked to perform insertion or removal. If a blocking queue is fair, it will allow the longest waiting thread to perform the operation when a condition arises that allows the operation to proceed. If the blocking queue is not fair, the order in which the blocked threads can perform the operation is not specified. Specific implementations determine fairness availability.

31. What is a map? Name the interface whose instances represent maps in Java.

Answer:

A map is a collection that stores key-value pairs. Keys in a map are unique. A map is represented by the `Map<K,V>` interface.

32. Name two implementation classes for the `Map<K,V>` interface in Java.

Answer:

`HashMap<K,V>` and `LinkedHashMap<K,V>` are the two implementation classes for the `Map<K,V>` interface.

33. What method do you use to get the number of entries in a `Map`? How do get a `Set` of all keys in a `Map`? How do you get a `Collection` of all values in a `Map`?

Answer:

The `size()` method returns number of entries (key-value pairs) in a `Map<K,V>`. The `keySet()` method returns `Set<K>` of all keys in the `Map<K,V>`. The `values()` method returns `Collection<V>` of all values in the `Map<K,V>`.

34. Consider the following snippet of code that creates a `Map` and populates it with names and their lucky numbers. Complete the code to print the unique names and unique lucky numbers in the map.

```
Map<String,Integer> map = new HashMap<>();
map.put("Bo", 1);
map.put("Co", 8);
map.put("Do", 19);
map.put("Lo", 1);
map.put("Mo", 8);

/* your code goes here */
```

Solution:

```
Map<String, Integer> map = new HashMap<>();
map.put("Bo", 1);
map.put("Co", 8);
map.put("Do", 19);
map.put("Lo", 1);
map.put("Mo", 8);

System.out.println("Unique names in map:");

// Get the set of keys
Set<String> keys = map.keySet();
keys.forEach(System.out::println);

System.out.println("Unique lucky numbers in map:");

// Create a set from values to get unique values
Set<Integer> values = new HashSet<>(map.values());
values.forEach(System.out::println);
```

35. Create immutable maps with the following five <key, value> entries of country codes and country names once using the `of()` method and once using the `ofEntries()` method of the `Map` interface: `<1, "United State">`, `<24, "Austria">`, `<66, "Thailand">`, `<49, "Germany">`, and `<91, "India">`. The keys are integers and values are strings. Print each entry on a separate line.

Solution:

```
// Create a Map using key-value pairs
Map<Integer, String> mapOf = Map.of(1, "United State",
        24, "Austria",
        66, "Thailand",
```

```
        49, "Germany",
        91, "India");

// Print the entries in the map
System.out.println("Entries in the map using of():");
mapOf.forEach((k, v) -> System.out.printf("key=%d, value=$s%n", k, v));

// Create a Map using Map.entry() method
Map<Integer, String> mapOfEntries = Map.ofEntries(entry(1, "United State"),
        entry(24, "Austria"),
        entry(66, "Thailand"),
        entry(49, "Germany"),
        entry(91, "India"));

System.out.println("\nEntries in the map using ofEntries():");
mapOfEntries.forEach((k, v) -> System.out.printf("key=%d, value=$s%n", k, v));
```

36. What is the `Collections` class? Name a few purposes for which this class is used.

Answer:

The `Collections` class is utility class in the Collections Framework. It provides methods for shuffling elements in a collection, rotating its elements, sorting elements of a list, etc.

37. What is wrong with the following snippet of code?

```
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(5, 50);
```

Answer:

Element 50 is being inserted at index 5, which is greater than size of list, which is 2. This will result in an `IndexOutOfBoundsException` at runtime.

38. Complete the following snippet of code that sorts a `List<Integer>` using the default `sort()` method in the `List` interface:

```
List<Integer> list = new ArrayList<>();
```

```
list.add(40);
list.add(10);
list.add(30);
list.add(20);
System.out.println("List: " + list);
list.sort(/* your code goes here */);
System.out.println("Sorted List: " + list);
```

The expected output is as follows:

```
List: [40, 10, 30, 20]
Sorted List: [10, 20, 30, 40]
```

Solution:

```
List<Integer> list = new ArrayList<>();
list.add(40);
list.add(10);
list.add(30);
list.add(20);
System.out.println("List: " + list);

// Sort the list. You can also use list.sort(null). See the note for more details
list.sort(Comparator.comparing(Integer::intValue));
System.out.println("Sorted List: " + list);
```

Note:

To sort the list, you can also use `list.sort(null)` in this case. If you pass null to the `sort()` method of the `List` interface, the list will be sorted in natural order if the elements in the list implements the `Comparable` interface. In this case, elements in the list are of the type `Integer`, which implements the `Comparable` interface.

39. Consider the following snippet of code that uses a binary search to search for `20` in the list:

```
List<Integer> list = new ArrayList<>();
list.add(40);
list.add(10);
list.add(30);
list.add(20);
System.out.println("List: " + list);

int index = Collections.binarySearch(list, 20);
```

```
System.out.println("Index of 20 in the list is " + index);
```

The current output is as follows:

```
List: [40, 10, 30, 20]
Index of 20 in the list is -3
```

The output indicates that `20` is not in the list. However, `20` is present in the list. Fix this snippet of code so that `20` is found in the list using the binary search. Describe your findings.

Solution:

```
List<Integer> list = new ArrayList<>();
list.add(40);
list.add(10);
list.add(30);
list.add(20);
System.out.println("List: " + list);
list.sort(null);
System.out.println("Sorted list: " + list);
int index = Collections.binarySearch(list, 20);
System.out.println("Index of 20 in the list is " + index);
```

Binary search works on a sorted list. Since the list is not sorted, behavior is unpredictable. Fix is to sort the list before calling the `binarySearch()` method.

40. What will be the output when you run the following snippet of code:

```
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(20);
list.add(30);
list.add(40);
System.out.println("List: " + list);

Collections.rotate(list, 4);
System.out.println("Rotated List: " + list);
```

Answer:

```
List: [10, 20, 30, 40]
Rotated List: [10, 20, 30, 40]
```

41. Write a snippet of code to create a modifiable `Set<Integer>`. Get an unmodifiable view of this set and demonstrate that you can still modify the original modifiable set and those modifications are reflected in the read-only set. Also demonstrate that an attempt to modify the read-only set throws an `UnsupportedOperationException`.

Solution:

```
Set<Integer> set = new HashSet<>();
set.add(10);
set.add(20);
set.add(30);
System.out.println("Initial modifiable set: " + set);

Set<Integer> unmodifiableSet = Collections.unmodifiableSet(set);
System.out.println("Unmodifiable set from the initial set: " + unmodifiableSet);

System.out.println("Adding two elements to the modifiable set");
set.add(40);
set.add(50);

System.out.println("Modifiable set after adding two elements: " + set);
System.out.println("Changes reflected in the unmodifiable set: " + unmodifiableSet);

System.out.println("Attempting to add an element to the unmodifiable set");
try {
    unmodifiableSet.add(60);
} catch (UnsupportedOperationException e) {
    System.out.println("UnsupportedOperationException thrown");
}
```

42. What is the advantage of using checked collections?

Answer:

A checked collection throws a `ClassCastException` when an element of any type other than the one mentioned during creation of collection is added to it. This makes debugging easier.

43. Write a snippet of code to create a singleton immutable `List<String>` with a lone element `"Hello"`.

Solution:

```
// Before Java 9
List<String> singleton1 = Collections.singletonList("Hello");
```

```
// From Java 9
List<String> singleton2 = List.of("Hello");
```

44. What are hash-based collections? What kind of special care must be taken with the class if the objects of the class will be stored in hash-based collections?

Answer:

Hash-based collections facilitate fast and efficient storage and retrieval of their elements. They use buckets to store elements. When an element is added to the collection, the element's hash code is used to determine the bucket in which the element will be stored.

The guidelines to ensure a class works as expected in hash-based collections are as follows::

- Consider using elements of an immutable class a `Set` and as key in a `Map`
- Implement `equals()` and `hashCode()` methods carefully making sure contracts between two methods are fulfilled

---