**Java Collections**

**What is Java collection framework?**

* The Java Collections Framework is a set of classes and interfaces that provide a way to store and manage groups of objects in Java.
* part of the java.util package and includes various data structures like lists, sets, maps, and queues.
* The framework also provides algorithms to manipulate these collections, such as searching, sorting, and shuffling.
* **Efficiency**: The framework provides efficient implementations of commonly used data structures, saving developers from writing their own.
* **Flexibility**: You can easily change the underlying data structure without affecting the rest of your code.
* **Interoperability**: Collections can be easily converted and manipulated using built-in methods and algorithms.
* **Reusability**: By using standard interfaces and classes, your code becomes more reusable and easier to maintain.

**What is the difference between a List, Set, and Map?**

1. **List**

* Definition: A List is an ordered collection (also known as a sequence) that allows duplicate elements.
* Features:
* Order: Maintains the insertion order of elements, meaning elements can be accessed by their index position.
* Duplicates: Allows duplicate elements, meaning the same element can appear multiple times.
* Index-based Access: Provides methods to access, insert, and remove elements based on their index position.
* Common Implementations:
* ArrayList: A resizable array implementation of the List interface. It offers fast random access, but slower insertions and deletions compared to linked lists.
* LinkedList: A doubly linked list implementation of the List interface. It provides better performance for insertions and deletions but slower random access.
* Example:

import java.util.ArrayList;

import java.util.List;

public class ListExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Apple"); // Allows duplicates

System.out.println(list); // Output: [Apple, Banana, Apple]

}

}

1. **Set**

* Definition: A Set is an unordered collection that does not allow duplicate elements.
* Features:
* Order: Does not maintain any specific order of elements. Some implementations like LinkedHashSet maintain insertion order,while TreeSet sorts the elements based on natural order or a custom comparator.
* Duplicates: Does not allow duplicate elements. Adding a duplicate element results in no change to the set.
* Common Implementations:
* HashSet: A hash table-based implementation that offers constant-time performance for basic operations. Does not guarantee order.
* LinkedHashSet: Maintains a doubly linked list to preserve the insertion order.
* TreeSet: A sorted set implementation based on a red-black tree, which sorts elements either by their natural order or by a provided comparator.
* Example:

import java.util.HashSet;

import java.util.Set;

public class SetExample {

public static void main(String[] args) {

Set<String> set = new HashSet<>();

set.add("Apple");

set.add("Banana");

set.add("Apple"); // Duplicates are ignored

System.out.println(set); // Output: [Apple, Banana]

}

}

1. **Map**:

* Definition: A Map is a collection of key-value pairs. Each key is unique, and each key maps to exactly one value.
* Key Characteristics:
* Keys and Values: Stores data in pairs (key and value), where keys are unique, and values can be duplicate.
* No Index-Based Access: Elements are accessed using keys, not indexes.
* Common Implementations:
* HashMap: A hash table-based implementation that provides constant-time performance for basic operations and does not guarantee order.
* LinkedHashMap: Maintains a doubly-linked list to preserve the insertion order of elements.
* TreeMap: A red-black tree-based implementation that sorts entries based on keys, either in natural order or a specified comparator.
* Example:

import java.util.HashMap;

import java.util.Map;

public class MapExample {

public static void main(String[] args) {

Map<String, Integer> map = new HashMap<>();

map.put("Apple", 10);

map.put("Banana", 20);

map.put("Apple", 30); // Overwrites the existing value

System.out.println(map); // Output: {Apple=30, Banana=20}

}

}

**What is the difference between ArrayList and LinkedList?**

**Array list:**

* ArrayList is backed by a dynamic array, meaning its size can change dynamically as elements are added or removed.
* Provides fast random access to elements due to its array-based structure (O(1) time complexity for accessing elements by index)
* Slower insertion and deletion operations, especially in the middle of the list, because elements need to be shifted to accommodate changes (O(n) time complexity).
* Requires less memory overhead compared to LinkedList since it only stores object data without additional pointers.
* When the array's capacity is exceeded, a new array is created, and existing elements are copied over, which can be costly if the list is large.
* Use cases:
* When you need quick access to elements using an index.
* When the list is primarily used for reading and there are fewer insertions or deletions.
* Example:

import java.util.ArrayList;

public class ArrayListExample {

public static void main(String[] args) {

ArrayList<String> arrayList = new ArrayList<>();

// Adding elements

arrayList.add("Apple");

arrayList.add("Banana");

arrayList.add("Cherry");

// Accessing elements by index

System.out.println("Element at index 1: " + arrayList.get(1)); // Output: Banana

// Inserting an element at index

arrayList.add(1, "Orange"); // Inserting "Orange" at index 1

System.out.println("After insertion: " + arrayList);

// Output: [Apple, Orange, Banana, Cherry]

// Removing an element

arrayList.remove("Banana");

System.out.println("After removal: " + arrayList); // Output: [Apple, Orange, Cherry]

}

}

**LinkedList:**

* LinkedList is implemented as a doubly-linked list, where each element (node) contains references to the previous and next elements.
* Slower access time compared to ArrayList (O(n) time complexity for accessing elements by index), as it requires traversal from the head or tail of the list.
* Efficient insertions and deletions, particularly at the beginning and end of the list (O(1) time complexity), since nodes can be easily linked or unlinked.
* Requires more memory than ArrayList due to the storage of additional pointers for each element (to the next and previous nodes).
* Uses:
* When the application involves frequent insertions and deletions, especially in the middle or at the start of the list.
* When memory overhead is not a significant concern.
* Example:

|  |  |
| --- | --- |
| import java.util.LinkedList;  public class LinkedListExample {  public static void main(String[] args) {  LinkedList<String> linkedList = new LinkedList<>();  // Adding elements  linkedList.add("Apple");  linkedList.add("Banana");  linkedList.add("Cherry");  // Accessing elements by index  System.out.println("Element at index 1: " + linkedList.get(1));  // Output: Banana  // Inserting an element at index  linkedList.add(1, "Orange"); // Inserting "Orange" at index 1  System.out.println("After insertion: " + linkedList);  // Output: [Apple, Orange, Banana, Cherry] | // Removing an element linkedList.remove("Banana");  System.out.println("After removal: " + linkedList);  // Output: [Apple, Orange, Cherry]  // Efficient addition at the beginning and end  linkedList.addFirst("Mango");  linkedList.addLast("Pineapple");  System.out.println("After additions: " + linkedList);  // Output: [Mango, Apple, Orange, Cherry, Pineapple]  }  } |

**What is linked list and its implementation?**

* A linked list is a linear data structure consisting of a sequence of elements, where each element is a separate object called a node.
* Each node contains two components: the data and a reference (or a pointer) to the next node in the sequence.
* Example:

null |1|x|->|addresofx|2|y|

* Types of Linked Lists
* Singly Linked List: Each node points to the next node, and the last node points to null.
* Doubly Linked List: Each node points to both the next and the previous nodes, allowing traversal in both directions.
* Circular Linked List: The last node points back to the first node, forming a circle.

class Node {

int data; // The data stored in the node

Node next; // Reference to the next node in the list

Node(int data) {

this.data = data;

this.next = null; // Initially, the next node is null

}

}

**What is the difference between hashset and treeset?**

* HashSet is implemented using a hash table. It uses a hash function to compute an index into an array of buckets, from which the desired value can be found.
* Does not maintain any order of the elements. The order of the elements can change over time, especially when elements are added or removed.
* Provides constant-time performance for the basic operations (add, remove, contains, and size), assuming the hash function disperses elements properly among the buckets.
* Allows one null element.
* Very efficient for operations that involve checking for the presence of an element, adding new elements, or removing elements.
* Elements must have a valid implementation of hashCode() and equals() methods for proper functioning.
* Example:

import java.util.HashSet;

import java.util.Set;

public class HashSetExample {

public static void main(String[] args) {

Set<String> hashSet = new HashSet<>();

// Adding elements to HashSet

hashSet.add("Apple");

hashSet.add("Banana");

hashSet.add("Orange");

hashSet.add("Mango");

// Adding a duplicate element

hashSet.add("Apple"); // Will not be added

// Adding a null element

hashSet.add(null);

// Displaying the HashSet

System.out.println("HashSet: " + hashSet);

// Output (order may vary): HashSet: [null, Apple, Banana, Orange, Mango]

}

}

* TreeSet: TreeSet is implemented using a red-black tree, which is a self-balancing binary search tree.
* Maintains elements in sorted order, which can be either natural ordering (as defined by the Comparable interface) or specified through a Comparator provided at the set’s creation.
* Provides logarithmic time performance for the basic operations (add, remove, contains), due to the underlying tree structure.
* Does not allow null elements because it relies on comparisons for ordering, and null cannot be compared to other elements.
* Useful when a sorted representation of elements is required, or when frequent range queries are needed.
* Elements must either implement the Comparable interface or a Comparator must be provided at creation to define the order.

import java.util.Set;

import java.util.TreeSet;

public class TreeSetExample {

public static void main(String[] args) {

Set<String> treeSet = new TreeSet<>();

// Adding elements to TreeSet

treeSet.add("Apple");

treeSet.add("Banana");

treeSet.add("Orange");

treeSet.add("Mango");

// Attempting to add a duplicate element

treeSet.add("Apple"); // Will not be added

// Attempting to add a null element (throws NullPointerException)

// treeSet.add(null); // Uncommenting this will throw a NullPointerException

// Displaying the TreeSet

System.out.println("TreeSet: " + treeSet);

// Output (sorted order): TreeSet: [Apple, Banana, Mango, Orange]

}

}

**What is the difference between HashMap and TreeMap?**

1. HashMap:

* Implements the Map interface.
* Backed by a hash table.
* Does not guarantee any order of keys.
* Allows null keys and values.
* Provides constant-time performance for get and put operations (O(1) average time complexity).

1. TreeMap:

* Implements the NavigableMap interface.
* Backed by a Red-Black tree.
* Guarantees that the keys are in sorted order.
* Does not allow null keys (but allows null values).
* Provides log(n) time cost for get and put operations.

(O(log n) time complexity).

**Difference between comparable and comparator?**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Comparable** | **Comparator** |
| * Package * Method * Single/Multiple * Modification * Usage | 1. java.lang 2. compareTo(Object o) 3. Defines a single sorting sequence 4. Modifies the class 5. Used when a single, natural ordering is required | 1. java.util 2. compare(Object o1, Object o2) 3. Can define multiple sorting sequences 4. Does not require modification of the class 5. Used when multiple orderings are needed or when the class should not be modified |

**Explain the internal working of HashMap.**

* HashMap uses an array of linked lists (buckets) for storing data. Each entry is stored in a bucket determined by the hash code of the key.
* When you add a key-value pair, the hash code of the key is calculated and used to find the correct bucket.
* If a collision occurs (multiple keys map to the same bucket), the entry is added to the linked list in that bucket.

**What are the different ways to iterate over a Map in Java?**

* Map<String, Integer> map = new HashMap<>();

map.put("apple", 1);

map.put("banana", 2);

* Using entrySet

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

System.out.println(entry.getKey() + ": " + entry.getValue());

}

* Using keySet

for (String key : map.keySet()) {

System.out.println(key + ": " + map.get(key));

}

* Using values

for (Integer value : map.values()) {

System.out.println(value);

}

* Using forEach (Java 8+)

map.forEach((key, value) -> System.out.println(key + ": " + value));

**Diff between fail fast and fail safe.**

**Fail fast iterator:**

* It throws a ConcurrentModificationException in modifying the object during the iteration process.
* No clone object is created during the iteration process.
* It requires low memory during the process.
* It does not allow modification during iteration.
* It is fast.
* HashMap, ArrayList, Vector, HashSet, etc.

**Fail safe iterator:**

* It does not throw Exception.
* A copy or clone object is created during the iteration process.
* It requires more memory during the process.
* It is slightly slower than Fail Fast.
* CopyOnWriteArrayList, ConcurrentHashMap, etc.

**What is a PriorityQueue and how does it work & types?**

* PriorityQueue is a special type of queue where elements are ordered based on their priority, defined by their natural ordering or a specified comparator.
* It uses a binary heap structure for storing elements, which allows efficient retrieval of the highest or lowest priority element.
* Example: We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

* All the values are arranged in ascending order.
* Now, we will observe how the priority queue will look after performing the following operations:
* **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
* **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
* **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
* **add(5):** It will insert 5 elements after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.
* Types of Priority Queue
* **Ascending order priority queue**:
* In ascending order priority queue, a lower priority number is given as a higher priority in a priority.
* For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.
* **Descending order priority queue:**
* In descending order priority queue, a higher priority number is given as a higher priority in a priority.
* For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest

**How would you remove duplicates from an ArrayList?**

You can use a HashSet to remove duplicates since it does not allow duplicate elements.

List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 2, 3, 4, 4));

Set<Integer> set = new HashSet<>(list);

list.clear();

list.addAll(set);

**What are the different ways to iterate over list?**

1. **Using a for Loop**

* A traditional for loop is one of the most straightforward methods for iterating over a list, especially when you need access to the index.

import java.util.List;

import java.util.ArrayList;

public class ForLoopExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// Iterate using a traditional for loop

for (int i = 0; i < list.size(); i++) {

String element = list.get(i);

System.out.println(element);

}

}

}

* + Allows easy access to the index.
  + Flexible for performing operations based on indices.

1. **Using an Enhanced for Loop (For-Each Loop)**

* The enhanced for loop, also known as the for-each loop, provides a more concise way to iterate over collections.

import java.util.List;

import java.util.ArrayList;

public class EnhancedForLoopExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// Iterate using an enhanced for loop

for (String element : list) {

System.out.println(element);

}

}

}

* More concise and readable than a traditional for loop.
* No risk of IndexOutOfBoundsException as it does not use indices.

1. Using an Iterator

* An Iterator provides a way to traverse the elements of a collection and is especially useful when you need to remove elements during iteration.

import java.util.List;

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// Iterate using an Iterator

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String element = iterator.next();

System.out.println(element);

// Remove element during iteration

if (element.equals("Banana")) {

iterator.remove();

}

}

System.out.println("List after removal: " + list);

}

}

1. **Using a ListIterator**

* A ListIterator is a bi-directional iterator that allows traversal of the list in both directions.
* It also provides methods to modify elements.

import java.util.List;

import java.util.ArrayList;

import java.util.ListIterator;

public class ListIteratorExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// Iterate using a ListIterator

ListIterator<String> listIterator = list.listIterator();

while (listIterator.hasNext()) {

String element = listIterator.next();

System.out.println(element);

// Modify element during iteration

if (element.equals("Banana")) {

listIterator.set("Blueberry");

}

}

System.out.println("List after modification: " + list);

}

}

1. **Using Java 8 forEach and Lambda Expressions**

* Java 8 introduced the forEach method, which allows iteration using lambda expressions, making the code more concise and expressive.

import java.util.List;

import java.util.ArrayList;

public class ForEachLambdaExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// Iterate using forEach and lambda expression

list.forEach(element -> System.out.println(element));

// Alternatively, using method reference

list.forEach(System.out::println);

}

}

1. **Using Java Streams (Java 8+)**

* Streams provide a high-level abstraction for processing sequences of elements. They are powerful for complex operations, including filtering, mapping, and reducing data.

import java.util.List;

import java.util.ArrayList;

public class StreamExample {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

// Iterate using streams

list.stream()

.filter(element -> element.startsWith("A"))

.forEach(System.out::println);

// Collect elements to a new list

List<String> filteredList = list.stream()

.filter(element -> element.startsWith("A"))

.collect(Collectors.toList());

System.out.println("Filtered List: " + filteredList);

}

}

**How does the HashSet work internally**

HashSet is backed by a HashMap. When you add an element to a HashSet, it actually adds the element as a key to the underlying HashMap with a constant dummy value.

HashSet<String> set = new HashSet<>();

set.add("hello");

// Internally adds "hello" as a key to a HashMap

**Differences Between Hashtable and HashMap.**

Hashtable and HashMap are both implementations of the Map interface in Java, but they have some key differences:

|  |  |  |
| --- | --- | --- |
| **Features** | **Hashtable** | **HashMap** |
| Synchronization | It is synchronized, meaning it is thread-safe and can be shared between multiple threads without additional synchronization. However, this comes with a performance cost due to locking. | It is not synchronized by default. If you need thread safety, you need to explicitly synchronize it, either using Collections.synchronizedMap() or using ConcurrentHashMap. |
| Null Keys and Values | Does not allow null keys or values. Attempting to use null will result in a NullPointerException. | Allows one null key and multiple null values. This is more flexible when dealing with data that might have null entries. |
| Performance | Due to its synchronized nature, Hashtable generally has slower performance compared to HashMap in single-threaded scenarios. | Faster for single-threaded applications since it doesn't incur the overhead of synchronization. |
| Legacy | Part of the original Java 1.0. It is considered a legacy class, though still widely used in some older applications. | Introduced in Java 1.2 as part of the Java Collections Framework, providing more modern capabilities. |

**When choosing a collection type, consider the following factors.**

1. **Data Structure:**
2. **List**: Use when you need an ordered collection that allows duplicates. Ideal for maintaining insertion order.
3. **Set**: Use when you need a collection with no duplicates. Ideal for unique elements.
4. **Map**: Use when you need to store key-value pairs and require fast lookup by key.
5. **Performance**:
6. **ArrayList vs. LinkedList**: Use ArrayList for fast random access and LinkedList for fast insertions and deletions at the cost of slower random access.
7. **HashSet vs. TreeSet**: Use HashSet for fast operations (O(1) average) without order, and TreeSet for ordered operations with a natural ordering or custom comparator (O(log n) operations).

1. **Thread Safety**: Use synchronized collections like Collections.synchronizedList() or ConcurrentHashMap when thread safety is required.
2. **Memory Usage**: Consider the memory overhead of each collection type, especially when dealing with large datasets.
3. **Null Handling**: Choose collections based on their ability to handle null values if your data may contain null entries.
4. **Ordering**: Use LinkedHashMap or LinkedHashSet if maintaining insertion order is important. Use TreeMap or TreeSet for natural or custom ordering.

**Differences Between Collection and Stream APIs in Java 8**

Collection API and Stream API serve different purposes in Java:

|  |  |  |
| --- | --- | --- |
| **Features** | **Collection API** | **Stream API** |
| Purpose | Provides a way to store and manage groups of objects. Allows operations like add, remove, update, and iterate over elements. | Provides a way to process sequences of elements, supporting functional-style operations like map, filter, and reduce. |
| Mutability /  Immutability | Collections are mutable; elements can be added or removed. | Streams are immutable and do not store data. They process data from a source (like a Collection) and produce a result. |
| Data Storage /  Operations | Collections physically store the elements. | Supports intermediate operations (transformations) and terminal operations (final result). |
| Traversing /  Parallel Processing | Use iterators or enhanced for-loops for traversal. | Streams can easily be processed in parallel using parallelStream() for improved performance. |

**How to Merge Two Maps in Java.**

To merge two maps and handle key collisions, you can use the putAll method or Java 8's merge method with a custom merge function.

import java.util.HashMap;

import java.util.Map;

public class MapMergeExample {

public static void main(String[] args) {

Map<String, Integer> map1 = new HashMap<>();

map1.put("A", 1);

map1.put("B", 2);

Map<String, Integer> map2 = new HashMap<>();

map2.put("B", 3);

map2.put("C", 4);

// Using putAll (overwrites values from map2)

Map<String, Integer> result = new HashMap<>(map1);

result.putAll(map2);

System.out.println("Merged Map (putAll): " + result);

// Using merge (custom logic for key collisions)

Map<String, Integer> mergedMap = new HashMap<>(map1);

map2.forEach((key, value) ->

mergedMap.merge(key, value, (v1, v2) -> v1 + v2) // Sum values on key collision

);

System.out.println("Merged Map (merge): " + mergedMap);

}

}

\* Using putAll: This method adds all entries from one map to another, overwriting values in the destination map when keys collide.

\* Using merge: This allows you to define a merge function to handle key collisions. In this example, values are summed if a key exists in both maps.

**Converting a List to a Set in Java.**

Converting a List to a Set can be done in several ways, each with its implications regarding duplicates and order.

//Using HashSet:

\*List<String> list = Arrays.asList("Apple", "Banana", "Apple", "Orange");

Set<String> set = new HashSet<>(list);

System.out.println("Set: " + set);

\*Implications: Removes duplicates. Does not maintain the order of elements.

Using LinkedHashSet:

Set<String> linkedSet = new LinkedHashSet<>(list);

System.out.println("LinkedHashSet: " + linkedSet);

Implications: Removes duplicates. Maintains the insertion order of elements.

\*Using TreeSet:

Set<String> treeSet = new TreeSet<>(list);

System.out.println("TreeSet: " + treeSet);

\*Implications: Removes duplicates. Sorts elements according to their natural order or a custom comparator if provided.

Using Streams (Java 8+):

Set<String> streamSet = list.stream().collect(Collectors.toSet());

System.out.println("Stream Set: " + streamSet);

Implications: Similar to HashSet, removes duplicates without maintaining order.

**What are all new features introduced in java8?**

Ans:1.Lambda Expressions:\*Lambda expressions are anonymous functions that provide a simple syntax for implementing functional interfaces

\*Purpose: Enable functional programming by allowing you to write code more concisely.

\*Syntax: (parameters) -> expression or (parameters) -> { statements; }

\*List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

// Using a lambda expression to sort the list

names.sort((s1, s2) -> s1.compareTo(s2));

// Equivalent to:

// names.sort(new Comparator<String>() {

// @Override

// public int compare(String s1, String s2) {

// return s1.compareTo(s2);

// }

// });

// Before Java 8: Using an anonymous class

Runnable r1 = new Runnable() {

@Override

public void run() {

System.out.println("Hello World!");

}

};

// With Java 8: Using a lambda expression

Runnable r2 = () -> System.out.println("Hello World!");

2. Functional Interfaces:

-------------------------

Ans: A functional interface is an interface that contains only one abstract method.

\*They can have multiple default or static methods but only one abstract method.

\*Java 8 introduced the @FunctionalInterface annotation to mark an interface as functional.

@FunctionalInterface

interface MyFunctionalInterface {

void doSomething();

}

Java 8 provides several built-in functional interfaces in the java.util.function package, such as:

\*Predicate<T>: Represents a boolean-valued function of one argument.

\*Consumer<T>: Represents an operation that accepts a single input argument and returns no result.

\*Function<T, R>: Represents a function that accepts one argument and produces a result.

\*Supplier<T>: Represents a supplier of results.

\*BiFunction<T, U, R>: Represents a function that accepts two arguments and produces a result.

3 Streams API

--------------

\*The Streams API is a new abstraction introduced in Java 8 that allows you to process sequences of elements (like collections) in a functional style.

It supports operations such as map, filter, reduce, collect, and more, enabling bulk processing of data.

example:

List<String> names = Arrays.asList("John", "Jane", "Jack", "Doe");

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("J"))

.collect(Collectors.toList());

System.out.println(filteredNames); // Output: [John, Jane, Jack]

Key Concepts:

\*Intermediate Operations: Return a stream and are lazily executed (e.g., filter, map).

\*Terminal Operations: Trigger the execution of the stream pipeline and return a non-stream result (e.g., collect, forEach, reduce).

\*Parallel Streams: Java 8 streams can be executed in parallel to leverage multicore processors for better performance.

4. Default Methods

-------------------

\*Java 8 allows interfaces to have default methods, which are methods with a default implementation.

\*This feature enables developers to add new methods to interfaces without breaking existing implementations.

interface MyInterface {

void existingMethod();

default void newDefaultMethod() {

System.out.println("This is a default method.");

}

}

class MyClass implements MyInterface {

@Override

public void existingMethod() {

System.out.println("Existing method implementation.");

}

}

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.existingMethod(); // Output: Existing method implementation.

obj.newDefaultMethod(); // Output: This is a default method.

}

}

Benefits:

----------

\*Enables backward compatibility with old interfaces.

\*Allows the evolution of interfaces with new methods.

5. Method References

--------------------

\*Method references provide a way to refer to methods or constructors without invoking them, using a double colon (::) operator.

\* They are often used in conjunction with lambda expressions to make code more readable.

Types of Method References:

\*Static Method Reference: ClassName::staticMethodName

\*\*Instance Method Reference of a Particular Object: instance::instanceMethodName

\*Instance Method Reference of an Arbitrary Object of a Particular Type: ClassName::instanceMethodName

\*Constructor Reference: ClassName::new

// Static method reference

Function<String, Integer> parseIntFunction = Integer::parseInt;

Integer number = parseIntFunction.apply("123"); // Output: 123

// Instance method reference of an arbitrary object

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.forEach(System.out::println);

// Constructor reference

Supplier<List<String>> listSupplier = ArrayList::new;

List<String> list = listSupplier.get();

6. Optional Class

-----------------

\*The Optional class is a container that represents the presence or absence of a value.

\* It is used to avoid null references and to write more robust code that explicitly handles missing values.

Optional<String> optionalName = Optional.ofNullable(getName());

optionalName.ifPresent(name -> System.out.println("Name: " + name));

String defaultName = optionalName.orElse("Unknown");

System.out.println("Default Name: " + defaultName);

Benefits:

\*Reduces the risk of NullPointerException.

\*Encourages better handling of absent values.

\*Makes the code more expressive and readable.

7. New Date and Time API

Java 8 introduces a new Date and Time API under the java.time package, which is more comprehensive and user-friendly compared to

the old java.util.Date and java.util.Calendar classes.

Key Classes:

LocalDate: Represents a date (year, month, day) without time.

LocalTime: Represents a time (hour, minute, second, nanosecond) without date.

LocalDateTime: Represents both date and time.

ZonedDateTime: Represents date and time with a time zone.

Duration and Period: Represent amounts of time.

8 Streams for Collections

0-----------------------0

\*The Collection API in Java 8 has been enhanced to include methods for obtaining a stream from a collection.

This allows for more functional and declarative operations on collections.

List<String> names = Arrays.asList("John", "Jane", "Jack");

names.stream()

.filter(name -> name.startsWith("J"))

.forEach(System.out::println);

----------------------------------------------------------------------------------------------------------------------------------------

Q.How do you define and use a Functional Interface in Java 8?

Ans:@FunctionalInterface

public interface MyFunctionalInterface {

void execute();

}

MyFunctionalInterface func = () -> System.out.println("Executing...");

func.execute();

-------------------------------------------------------------------------------------------------------------------------------------

Q.What are Default Methods in interfaces?

Ans: Default Methods allow you to add new functionality to interfaces without breaking the classes that implement them.

They are defined with the default keyword and can have a method body.

Example:

java

Copy code

public interface MyInterface {

default void newMethod() {

System.out.println("New default method");

}

}

----------------------------------------------------------------------------------------------------------------------------------------------------------------

Q.What is an Optional in Java 8, and why is it used?

\*The Optional class is a container for optional values that may or may not be present.

\*It helps to avoid NullPointerException by providing methods like isPresent(), ifPresent(), orElse(), and orElseGet().

Optional<String> optionalName = Optional.ofNullable(getName());

optionalName.ifPresent(name -> System.out.println(name));

-------------------------------------------------------------------------------------------------------------------------------------------------------------

Q.How do you create and work with Streams in Java 8?

Ans:Creating Streams:

List<String> list = Arrays.asList("one", "two", "three");

Stream<String> stream = list.stream();

List<String> filtered = stream.filter(s -> s.length() > 3)

.map(String::toUpperCase)

.collect(Collectors.toList());

--------------------------------------------------------------------------------------------------------------------------------------------------------

Q.what is difference between map() and flatmap()?

Ans://map()

\*Purpose: The map() function is used to transform each element in a stream.

It applies a function to each element and collects the results into a new stream,

maintaining a one-to-one correspondence between input and output elements.

\*Output: Produces a single result for each input element.

\*Use Case: Use map() when you have a simple transformation where each element corresponds to exactly one result.

List<String> words = Arrays.asList("hello", "world", "java", "stream");

List<Integer> lengths = words.stream()

.map(String::length)

.collect(Collectors.toList());

System.out.println(lengths); // Output: [5, 5, 4, 6]

In this example:

\*map(String::length) transforms each string in the stream to its length (an integer).

\*The resulting stream contains integers representing the lengths of the original strings.

//flatMap()

\*Purpose: The flatMap() function is used for transforming and flattening a stream of collections or arrays into a single continuous stream.

It maps each element to a collection, then flattens the collections into a single stream.

\*Output: Produces a flat stream that contains all elements from the collections produced by the mapping function, effectively performing a one-to-many transformation.

\*Use Case: Use flatMap() when you need to flatten nested collections or when each element in the stream should map to multiple results.

List<String> sentences = Arrays.asList("hello world", "java streams", "flat map");

List<String> words = sentences.stream()

.flatMap(sentence -> Arrays.stream(sentence.split(" ")))

.collect(Collectors.toList());

System.out.println(words); // Output: [hello, world, java, streams, flat, map]

In this example:

\*flatMap(sentence -> Arrays.stream(sentence.split(" "))) maps each sentence to a stream of words.

\*flatMap() then flattens these streams of words into a single stream.

\*The resulting stream contains all words from all sentences, flattened into a single list.

Key Differences

\*Aspect map() flatMap()

\*Transformation //One-to-one transformation One-to-many transformation

\*Output Each input element maps to exactly one output element Each input element maps to zero or more output elements

\*Usage Use when transforming each element to a single result Use when transforming each element to multiple results

Use Cases

\*map() is typically used when you want to perform operations like converting a list of integers to their squares,

changing strings to uppercase, or extracting a property from objects.

\*flatMap() is commonly used for flattening nested structures, such as when working with lists of lists,

streams of arrays, or when parsing data structures like JSON into flat collections.

----------------------------------------------------------------------------------------------------------------------------------------------------

Q.What are Collectors in Java 8 Streams?

Ans: Collectors are utility classes used in conjunction with the collect() method to accumulate elements from a Stream into collections,

strings, or other types. Common collectors include toList(), toSet(), joining(), groupingBy(), and partitioningBy().

List<String> immutableList = Collections.unmodifiableList(Arrays.asList("a", "b", "c"));

// List<Character> duplicatechar=

// str.chars().mapToObj(c->(char) c).

// collect(Collectors.groupingBy(c -> c,Collectors.counting())).entrySet().

// stream().filter(entry->entry.getValue()>1).map(Map.Entry::getKey).collect(Collectors.toList());

//

// System.out.println(duplicatechar);