**Explain Java8 Features?**

Lambda Expressions

Functional Interfaces

Streams API

Default Methods

Optional

Method References

Date and Time API

**Explain Lambda expression?**

Lambda expression is a block of code like an anonymous function.

it has no name.

it represents a block of code as an object.

it can have zero or more parameters.

Instead of creating an anonymous class to implement the functional interface, we can use a lambda expression.

**What is Functional interface?**

An interface contains exactly one abstract method.

It can have multiple default or static methods.

We use Lambda expression to provide implementations for the abstract method of a functional interface.

We mark a functional interface with the @FunctionalInterface annotation.

**What are predefined functional interfaces in java?**

Predicate:

It represents a boolean-valued function of one argument.

It is used for filtering elements in collections.

Example: Predicate<String> isLong = s -> s.length() > 5;

Consumer:

It accepts a single input argument and returns no result.

It is used for performing actions on elements in collections.

Example: Consumer<Integer> printNumber = n -> System.out.println(n);

Function:

It accepts one argument and produces a result.

It is used for mapping elements in collections.

Example: Function<Integer, String> convertToString = n -> Integer.toString(n);

Supplier:

It represents a supplier of results.

It has no input arguments and produces a result of type T.

It is used for lazy initialization or generating values.

Example: Supplier<Double> randomNumber = () -> Math.random();

UnaryOperator:

It represents an operation on a single operand.

It produces a result of the same type as its operand.

It is used for transformations.

Example: UnaryOperator<Integer> square = n -> n \* n;

BinaryOperator:

It represents an operation upon two operands of the same type.

It produce a result of the same type as the operands.

It is used for combining values.

Example: BinaryOperator<Integer> sum = (a, b) -> a + b;

BiFunction:

It Represents a function that accepts two arguments and produces a result.

It is similar to Function, but it takes two input arguments instead of one.

Example: BiFunction<Integer, Integer, String> combine = (a, b) -> a.toString() + b.toString();

BiConsumer:

It represents an operation that accepts two input arguments and returns no result.

It's similar to Consumer, but it takes two input arguments instead of one.

Example:

BiConsumer<String, Integer> printRepeatedly = (s, count) -> {

for (int i = 0; i < count; i++) {

System.out.println(s);

}

};

**Differences between interface and functional interface?**

Interface:

Interface in Java is a reference type, similar to a class.

Before Java 8, interfaces have only abstract methods.

Now Interface can contain abstract methods, default methods, static methods, and constant variables.

Interface is used to achieve polymorphism and code reusability.

Functional Interface:

Functional interface that contains exactly one abstract.

Functional interfaces can have any number of default or static methods, but they must have exactly one abstract method.

Functional interface designed to use with lambda expressions and method references.

**Already we have interfaces in java why functional interfaces requried?**

To support lambda expressions and method references we use functional interface.

**What is Method Reference?**

Refer to methods or constructors without invoking them.

We can refer method of functional interface.

we can refer to static method defined in the class.

We can refer instance methods also.

we can refer a constructor by using the new keyword.

**What is default method in java?**

Default methods provide a default implementation that can be used by classes by implementing the interface.

Classes can implement multiple interfaces with default methods without conflicts.

Classes implementing the interface **can override default methods** if they wish to provide a different implementation.

Default methods allow interfaces to provide method implementations by adding new methods with default implementations without breaking existing code.

**What is diamond problem in java?**

The diamond problem occurs when a class implements multiple interfaces that have conflicting default methods with the same signature.

This can create ambiguity, as the compiler doesn't know which implementation to choose.

To resolve the diamond problem in Java we can use default methods.

**How to resolve diamond problem in java using default methods?**

We can resolve the conflict by providing an explicit implementation.

If you want to use the default implementation of one specific interface and ignore the other, you can explicitly choose using InterfaceName.super.method().

Example:

class MyClass implements InterfaceA, InterfaceB {

@Override

public void method() {

InterfaceA.super.method(); // Or InterfaceB.super.method();

}

}

**What are the uses of static methods in java8?**

Static methods are used for utility methods.

Static method does not depend on an instance of the implementing class.

Static methods can be used as factory methods to create instances of classes that implement the interface.

**Explain Optional class in java8?**

It provides methods to check the presence of value for particular variable.

It returns true if there is a value present, otherwise false.

To avoid abnormal termination, we use the Optional class.

isPresent, ifPresent, empty are common methods of optional class.

**Explain Streams in java8?**

Streams are used for processing collections of objects.

We use stream to filter, collect, print, and convert from one data structure to other etc.

Stream does not store elements.

Operations performed on a stream does not modify it's source.

**Explain Stream operations?**

Stream operations are mainly two types.

Terminal operations & Intermediate operations.

Intermediate operations transform streams and these are lazy means they are not executed until a terminal operation is invoked.

Terminal operations produce a result by trigger the processing of the stream pipeline.

Intermediate operations return a new stream.

Filter, map, flatmap, distinct, sorted, skip are intermediate operations.

collect, foreach, reduce, count, findFirst are terminal operations.

**Explain intermediate operations of stream?**

Intermediate operations transform streams and these are lazy means they are not executed until a terminal operation is invoked.

Intermediate operations return a new stream.

Filter, map, flatmap, distinct, sorted, skip are intermediate operations.

Filter:

Filters the elements of the stream based on a given predicate.

Map:

Transforms each element of the stream using a given function.

FlatMap:

Transforms each element into a stream of other elements and flattens the result.

Distinct:

Removes duplicate elements from the stream.

Sorted:

Sorts the elements of the stream in natural order.

Skip:

Skips the first n elements of the stream.

**Explain terminal operations of stream?**

Terminal operations produce a result by trigger the processing of the stream pipeline.

They are the endpoint of a stream and finalize the chain of operations.

collect:

Collects the elements of the stream into a collection

forEach:

Performs an action for each element of the stream.

reduce:

Reduces the elements of the stream to a single value using an associative accumulation function.

count:

Returns the number of elements in the stream.

min:

Returns the minimum element of the stream.

max:

Returns the maximum element of the stream.

findFirst:

Finds the first element of the stream.

**Explain Date & Time API in java8?**

Local date, local time classes represents date and time without a time zone.

we have Zoned Date Time class that displays combined date and time with zone.

Calculations with date and time are easy by using methods like plus, minus, until, and between.

Period, Duration, Instant are useful classes in new API.

**Explain advantages of Date & Time API in java8?**

New Date and time API classes overcome the drawbacks of existed API.

Previous Date and calendar classes does not handle time zones.

Previous Date and calendar classes has limited functionality so developers need to write additional utility methods or use external libraries.

Previous Date and Calendar classes are mutable so their state can be change across different parts of shared application.

Previous Date and calendar classes are not thread-safe.

**Explain Collectors class?**

Collectors class provides that accumulating elements into collections.

toList,toSet,toMap,groupingBy,counting,joining are common methods provided by Collectors.

**Explain parallel streams?**

It perform parallel processing of data using the Stream API.

It speed up data processing tasks by dividing it into multiple threads and utilizing the multiple cores of modern processors.

we can create a parallel stream from a collection or by calling the parallel() method.

In stream API we use parallelStream method or parallel stream processing.

When we process elements with a parallel stream, the Stream API internally uses the ForkJoinPool to split the workload and process elements in parallel.

Parallel processing is made easy without the need for explicit thread management.

**Explain ForEach method?**

It is used to perform the specified action on each element.

This method is a terminal operation of stream.

When using parallel streams, the forEach method can process elements in parallel.

The order of processing is not guaranteed in parallel stream so forEachOrdered introduced.

**Explain flatMap?**

It used to transform each element of a stream into another stream and then flatten the resulting streams into a single stream.

If we have a list of lists and we want to create a single list of all the elements then flatMap can be used.

This is useful for nested collections.

The flatMap method takes each element from the input stream, applies a mapping function that returns a stream of new values, and then concatenates these streams into a single stream. This process is called "flattening".

When each element of the stream can be expanded into multiple elements then flatMap is the tool to use.

**Example for Flattening a List of Lists:**

List<List<String>> listOfLists = Arrays.asList(

Arrays.asList("A", "B", "C"),

Arrays.asList("D", "E", "F"),

Arrays.asList("G", "H", "I")

);

List<String> flattenedList = listOfLists.stream().flatMap(List::stream).collect(Collectors.toList());

System.out.println(flattenedList); // Output: [A, B, C, D, E, F, G, H, I]

**Example 2: Expanding Elements**

Suppose you have a list of sentences and you want to split each sentence into words and then flatten the result into a single stream of words.

List<String> sentences = Arrays.asList("Hello world", "Java 8 Streams", "FlatMap example");

List<String> words = sentences.stream()

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

.collect(Collectors.toList());

System.out.println(words);

// Output: [Hello, world, Java, 8, Streams, FlatMap, example]

**Use optional with flatmap?**

Optional<String> name = Optional.of("Java");

Optional<String> upperCaseName = name.flatMap(n -> Optional.of(n.toUpperCase()));

upperCaseName.ifPresent(System.out::println); // Output: JAVA

**Explain comparable and comparator?**

Comparable and Comparator are two interfaces used for comparing objects.

The Comparable interface is used to define the natural ordering of objects of a class.

A class that implements Comparable must override the compareTo method, which is used to compare the current object with another object of the same type.

A class can have only one natural ordering, which is defined by the compareTo method.

Returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object.

The Comparator interface is used to define multiple ways to compare two objects. This is useful when you need different sorting criteria.

You can define multiple Comparator instances to compare objects in different ways.

**what are IO Enhancements in java8?**

Java 8 expanded the capabilities of the NIO package.

Introduced the Files utility class, which provides static methods for performing various file-related operations, such as reading/writing files, checking file existence, creating directories, copying/moving files, deleting files, and more.

Path, Files, FileSystem, FileVisitor, and WatchService are new classes in NIO.

The java.io.BufferedReader and java.io.BufferedWriter classes were enhanced to support reading and writing data as streams.

The try-with-resources statement enhanced to support automatic closing of resources declared outside the try block.

Java 8 introduced support for asynchronous I/O operations using the CompletableFuture class and AsynchronousFileChannel class.

**What are the Java8 concurrency enhancements?**

CompletableFuture class enables a more fluent and declarative style of asynchronous programming compared to using callbacks or traditional Future objects.

Streams can leverage parallelism using the parallel() method, which converts a sequential stream into a parallel stream, allowing operations to be automatically parallelized across multiple threads.

The Arrays.parallelSort() method was introduced in Java 8, allowing arrays to be sorted in parallel using multiple threads. This method is useful for sorting large arrays more efficiently on multi-core processors, potentially improving performance compared to sequential sorting algorithms.

Java 8 introduced several enhancements to the ConcurrentHashMap class, making it more efficient and easier to use in concurrent applications. These enhancements include methods like forEach, compute, merge, and reduce, which allow for more flexible and expressive concurrent data manipulation.

The StampedLock class was introduced in Java 8, providing a new locking mechanism with better performance characteristics compared to traditional ReadWriteLock. StampedLock supports three modes of locking: reading, writing, and optimistic reading, allowing for more fine-grained control over concurrency.

Enhanced Atomic classes to perform complex atomic operations on variables shared between threads, reducing the need for explicit synchronization. Included updateAndGet, accumulateAndGet to provide atomic updates with a specified update function.

Enhanced Executor and ExecutorService to create and manage thread pools and execute tasks asynchronously. New factory methods like newWorkStealingPool and newCachedThreadPool provide more flexibility in creating different types of thread pools.

**Explain ConcurrentHashMap?**

ConcurrentHashMap is a thread-safe implementation of a hash table in Java, designed for use in concurrent environments.

ConcurrentHashMap allows multiple threads to read and write without locking the entire map.

ConcurrentHashMap uses a technique called lock striping, which divides the map into segments, each protected by its own lock. This allows multiple threads to access different segments simultaneously, reducing contention and improving concurrency.

ConcurrentHashMap provides atomic operations like putIfAbsent, remove, replace, compute, merge, and other bulk operations.

**Explain differences between Hashmap and ConcurrentHashMap?**

HashMap is not thread-safe. If multiple threads access a HashMap concurrently it must be synchronized externally to avoid data inconsistency and corruption.

ConcurrentHashMap is designed for concurrent use. It allows multiple threads to read and write concurrently without the need for external synchronization.

HashMap is generally faster than ConcurrentHashMap when used in a single-

HashMap allows one null key and multiple null values.

ConcurrentHashMap does not allow null keys or values to avoid ambiguity in concurrency.

HashMap uses a hash table data structure with an array of linked lists (or trees in the case of many collisions, introduced in Java 8) to store entries.

ConcurrentHashMap uses a technique called lock striping, where the map is divided into segments, each protected by its own lock. This allows for higher concurrency.

HashMap does not support atomic operations directly.

ConcurrentHashMap provides atomic operations like putIfAbsent, remove, replace, compute, merge, and other bulk operations.

**What is atomic operation in Java?**

Atomicity is crucial in multithreaded environments to prevent race conditions and ensure thread safety.

Atomic operations completes in a single step without any possibility of interference from other threads.

Atomic operations maintain the consistency of the data, ensuring that no thread sees the data in an inconsistent state.

Atomic operations are inherently thread-safe without requiring explicit synchronization mechanisms like locks.

Many atomic operations rely on a technique called Compare-and-Swap (CAS).

CAS is a hardware-supported atomic instruction used to achieve synchronization.

CAS compares the current value of a memory location to an expected value and, if they match, updates the memory location to a new value. This process is atomic and ensures that the value is not changed by other threads between the comparison and the update.

Atomic classes are suitable for single variable operations. For compound operations, explicit synchronization is still needed.

In some scenarios, particularly with atomic references, an update may be incorrectly applied if an intermediate change is not detected. This can be addressed using AtomicStampedReference. we call it as ABA Problem.