Java 8 was a significant update to the Java programming language and introduced several new features. Here are some of the key features:

**Lambda Expressions:** These are anonymous functions that you can use to reduce the boilerplate code and can be used in functional programming.

**Stream API**: This is used to process collections of objects in a functional manner.

**Functional Interfaces**: An interface with exactly one abstract method becomes Functional Interface. @FunctionalInterface annotation is added so that we can mark an interface as functional interface.

@FunctionalInterface

public interface MyInterface {

    void method();

}

**Default and Static Methods in Interfaces**: In Java 8, interfaces can have default and static methods.

**Optional Class:** The Optional class is used to represent nullable object references. It can help prevent null pointer exceptions.

**Method References**: Method references help to point methods by their names. A method reference is described using "::" symbol.

**Parallel Array Sorting:** The Arrays.parallelSort() method was added, which uses multithreading for sorting large arrays.

int[] numbers = {5, 1, 6, 2, 4, 3};

Arrays.parallelSort(numbers);

**Date and Time API:** Java 8 introduced a new date and time API under the package java.time. It is more efficient and easier to use than the older java.util.Date and java.util.Calendar.

**Collectors Class**: The Collectors class provides various methods for operations like averaging, grouping, joining etc.

The Collectors class in Java provides a set of static methods for creating Collector implementations, these collectors are used to accumulate elements of a stream into various forms, such as lists, sets, maps, or other custom results.

toList(), toSet(), toMap(), groupingBy(), partitioningBy(),joining()

Here’s a detailed look at some common usages and examples of the Collectors class:

1. **Collecting to a List**

The toList() collector collects the elements of a stream into a List.

List<String> list = Stream.of("a", "b", "c", "d") .collect(Collectors.toList());

System.out.println(list); // Output: [a, b, c, d]

2. **Collecting to a Set**

The toSet() collector collects the elements of a stream into a Set, which removes duplicates.

Set<String> set = Stream.of("a", "b", "a", "c") .collect(Collectors.toSet());

System.out.println(set); // Output: [a, b, c]

3. **Collecting to a Map**

The toMap() collector collects the elements of a stream into a Map. You need to provide two functions: one for the key and one for the value.

Map<String, Integer> map = Stream.of("a", "b", "c") .collect(Collectors.toMap( k -> k, // Key mapper

String::length // Value mapper ));

System.out.println(map); // Output: {a=1, b=1, c=1}

4. **Grouping Elements**

The groupingBy() collector groups the elements of a stream by a classifier function.

Map<Integer, List<String>> grouped = Stream.of("apple", "banana", "kiwi", "apricot")

.collect(Collectors.groupingBy(String::length));

System.out.println(grouped); // Output: {5=[apple], 6=[banana, apricot], 4=[kiwi]}

5. **Partitioning Elements**

The partitioningBy() collector partitions the elements of a stream into two groups based on a predicate.

Map<Boolean, List<String>> partitioned = Stream.of("apple", "banana", "kiwi", "apricot")

.collect(Collectors.partitioningBy(s -> s.length() > 5));

System.out.println(partitioned); // Output: {false=[apple, kiwi], true=[banana, apricot]}

6. **Joining Elements**

The joining() collector concatenates the elements of a stream into a single String. You can also provide delimiters, prefixes, and suffixes.

String result = Stream.of("apple", "banana", "kiwi") .collect(Collectors.joining(", ", "[", "]"));

System.out.println(result); // Output: [apple, banana, kiwi]

7. **Summarizing Statistics**

The summarizingInt(), summarizingDouble(), and summarizingLong() collectors compute statistics, such as count, sum, min, average, and max.

IntSummaryStatistics stats = Stream.of(1, 2, 3, 4, 5) .collect(Collectors.summarizingInt(Integer::intValue));

System.out.println(stats); // Output: IntSummaryStatistics{count=5, sum=15, min=1, average=3.000000, max=5}

8. **Counting Elements**

The counting() collector counts the number of elements in a stream.

long count = Stream.of("apple", "banana", "kiwi").collect(Collectors.counting());

System.out.println(count); // Output: 3

9. **Reducing Elements**

The reducing() collector performs a reduction on the elements of the stream using an associative accumulation function and returns an Optional.

Optional<String> concatenated = Stream.of("apple", "banana", "kiwi") .collect(Collectors.reducing((s1, s2) -> s1 + ", " + s2));

concatenated.ifPresent(System.out::println);

// Output: apple, banana, kiwi

10. **Custom Collectors**

You can also create custom collectors using the Collector interface and the Collectors utility methods.

public static void main(String[] args) {

Collector<String, List<String>, List<String>> customCollector =

Collector.of(

ArrayList::new, // Supplier

ArrayList::add, // Accumulator

(left, right) -> { // Combiner

left.addAll(right);

return left;

},

ArrayList::new // Finisher

);

List<String> result = Stream.of("apple", "banana", "kiwi").collect(customCollector);

System.out.println(result); // Output: [apple, banana, kiwi]

}

These examples illustrate various ways to use the Collectors class to perform common operations on streams in Java. Each collector has its own use case and can be combined to handle complex data processing scenarios.

Lambda Expressions in Java:

**What is a Lambda Expression in Java?**

A Lambda Expression is a function which can be created without belonging to any class. It is a way to represent functional interface in a concise way and is used primarily to define inline implementation of a functional interface.

**What is the syntax of a Lambda Expression?**

The syntax of a lambda expression is: (parameters) -> expression or (parameters) -> { statements; }

**Can we use Lambda Expressions to replace all anonymous classes in Java?**

No, we can only use Lambda Expressions to replace those anonymous classes that implement a functional interface (an interface with only one abstract method).

**What is a functional interface?**

A functional interface is an interface that contains only one abstract method. They can have any number of default or static methods. Functional interfaces are also known as Single Abstract Method interfaces (SAM Interfaces).

**Can we define our own functional interface?**

Yes, we can define our own functional interfaces and they can be used with lambda expressions.

**What is the type of a Lambda Expression?**

The type of a lambda expression is a functional interface type.

**Can a Lambda Expression be passed as a method argument?**

Yes, a lambda expression can be passed as a method argument if the parameter type is a functional interface.

**Can Lambda Expressions have return statements?**

Yes, lambda expressions can have return statements. If the body of lambda expression has a single statement, the return type of the functional interface method matches the return type of that statement.

Here's an example of a lambda expression:

In this example, () -> System.out.println("Hello, World!") is a lambda expression that implements the Runnable interface, which is a functional interface.

**What is the Stream API in Java?**

The Stream API in Java is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

**What are the benefits of using the Stream API?**

The Stream API supports functional-style operations on streams of elements, such as map-reduce transformations on collections. It can greatly improve productivity by providing a high-level abstraction for performing complex data processing operations.

**What is the difference between a Stream and a Collection in Java?**

Collections are in-memory data structures that hold elements within it. Each element in the collection is computed before it actually becomes part of the collection. On the other hand, Streams are fixed data structures whose elements are computed on demand.

**What are the different types of Streams?**

There are two types of Streams: sequential and parallel. Sequential streams work just like for-loop using a single core, whereas parallel streams divide the work in multiple cores of the processor.

**What are some operations provided by the Stream API?**

The Stream API provides a set of operations divided into intermediate and terminal operations. Intermediate operations that transform a Stream into another Stream (such as filter and map). Terminal operations are operations that terminate the Stream and start the processing (such as collect, count, forEach, reduce).

**Can we reuse Streams in Java?**

No, streams in Java are not reusable. Once you have used a stream, you cannot use it again. If you need to traverse the same data source again, you must return a new stream.

**How can I create a Stream in Java?**

You can create a Stream from a Collection, an array, or an I/O channel. Here's an example of creating a Stream from a List:

List<String> list = Arrays.asList("a", "b", "c"); Stream<String> stream = list.stream();

**Can we use primitive types with Streams?**

Yes, Java provides specialized streams for the primitive type’s int, long and double namely IntStream, LongStream and DoubleStream respectively.

**What is a flatMap operation in the Stream API?**

The flatMap operation is used to flatten the input Stream of Streams into a Stream of objects. It is a combination of map and flatten.

**What is the collect method in the Stream API?**

The collect method is a terminal operation that transforms the elements of the stream into a different kind of result, such as a List, Set, or Map.

Here's an example of using the Stream API to filter and transform a List:

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

List<String> result = names.stream()

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

.map(String::toUpperCase)

.collect(Collectors.toList());

In this example, the Stream API is used to filter the names that start with "J", convert them to uppercase, and collect the results into a new List.

**map and flatMap are operations on streams. They are used to transform the elements of the stream.**

map: This operation applies a given function to each item of a stream and returns a new stream containing the results. It does not alter the original stream.

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

List<Integer> squares = numbers.stream()

.map(x -> x \* x)

.collect(Collectors.toList()); // [1, 4, 9, 16, 25]

flatMap: This operation first applies a mapping function just like map does, but then it flattens the result. This is useful when each application of the map function produces multiple results (in the form of a stream) and you want to combine all these results into a single stream.

In this example, flatMap is used to flatten a stream of lists into a single stream.

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

Arrays.asList(1, 2, 3),

Arrays.asList(4, 5, 6),

Arrays.asList(7, 8, 9)

);

List<Integer> allNumbers = listOfLists.stream()

.flatMap(List::stream)

.collect(Collectors.toList()); // [1, 2, 3, 4, 5, 6, 7, 8, 9]

In summary, the main difference between map and flatMap in Java streams is that flatMap has an extra step where it flattens the output.

Both of the functions map() and flatMap are used for transformation and mapping operations. map() function produces one output for one input value, whereas flatMap() function produces an arbitrary no of values as output (ie zero or more than zero) for each input value.

**In the Stream API in Java, operations are divided into three categories:**

**Creation Operations:** These operations are used to create a stream. Examples include stream(), parallelStream(), of(), iterate(), generate(), etc.

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

Stream<String> stream = list.stream(); // Creation operation

**Intermediate Operations:** These operations transform a stream into another stream. They are always lazy, executing an intermediate operation such as filter() does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate. Examples include map(), filter(), sorted(), flatMap(), etc.

Stream<String> filteredStream = stream.filter(s -> s.startsWith("a")); // Intermediate operation

**Terminal Operations:** These operations produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. Examples include forEach(), reduce(), collect(), min(), max(), count(), anyMatch(), etc.

In the example above, stream() is a creation operation, filter() is an intermediate operation, and collect() is a terminal operation.

List<String> filteredList = filteredStream.collect(Collectors.toList()); // Terminal operation

**53 What is a Stream?**

An in-momery implementation of map/filter/reduce pattern. Stream API allows you to makes computation in parallel.

Stream consumes elopements from source and it can do several of these operations and are organisated in a pipeline where each element is passed from one step of the pipeline to the next one. this is used to filter based on the predicate. It may consists many things summing the elements extracting min and max or adding them to a list.

map, filter and reduce

map -> standardizeAvatars 👉 Accepts an array of users, and adds a base url to all of the avatar urls

filter -> getActiveUsers 👉 Accepts an array of users, and returns only the active ones.

reduce -> getAllAges 👉 Accepts an array of users, and gets the total age of everyone in the array

In Java's Stream API, map, filter, and reduce are three fundamental operations that you can perform on streams. Here's a brief explanation of each:

**map**: The map operation is used to transform the elements of the Stream. It takes a Function as an argument, which is applied to each element in the stream and produces the new value for each element.

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

List<String> upperCaseNames = names.stream()

.map(String::toUpperCase)

.collect(Collectors.toList());

// Result: ["JOHN", "JANE", "ADAM", "EVE"]

**filter**: The filter operation is used to select elements from the Stream, based on a predicate. It takes a Predicate as an argument and returns a stream including all elements that match the predicate.

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

List<String> namesStartingWithJ = names.stream()

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

.collect(Collectors.toList());

// Result: ["John", "Jane"]

**reduce**: The reduce operation is used to reduce the elements of a stream to a single value. The reduce method takes a BinaryOperator as a parameter and returns an Optional describing the reduced value, if any. This operation is terminal.

In this example, the reduce method is used to calculate the sum of the numbers in the list. The Integer::sum method reference is used as the BinaryOperator.

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> sum = numbers.stream()

.reduce(Integer::sum);

// Result: Optional[15]

Remember, map and filter are intermediate operations which return a new Stream, while reduce is a terminal operation which returns a result or a side-effect.

In Java, a **side effect** is an action that changes the state of the program outside of the current function or operation. In the context of the Stream API, side effects are changes that occur when a function (like a lambda expression) modifies the state of variables outside its own scope.

For example, consider the following code:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = 0;

numbers.stream().forEach(n -> sum += n); // This will not compile

In this example, the lambda expression n -> sum += n inside the forEach operation is trying to modify the sum variable, which is outside its scope. This is a side effect. However, this code will not compile in Java, because variables in the outer scope that are accessed from a lambda expression must be effectively final.

A correct way to sum the numbers using the Stream API would be to use the reduce operation, which does not have side effects:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

Optional<Integer> sum = numbers.stream().reduce(Integer::sum);

In general, you should avoid side effects when working with the Stream API in Java, because they can lead to unpredictable results, especially when working with parallel streams. The Stream API is designed to work correctly without the need for side effects.

115. **What are the four fundamental functional interfaces in Java?**

Consumer<String> consumer = s -> System.out.println(s); take a argument and return nothing

Suppiler<Double> supplier = () -> Math.PI; doesn't take anything and return something

Function<String, Integer> length = s-> s.length(); takes something and return something else

Predicate<Integer> predicate = i -> i% 2 == 0; takes argument and return Boolean

Runnable task = () -> "I am the fifth"; doesn't take any argument and doesn't return anything

BiConsumer<String, String> biCons = (s1,s2) -> System.out.println(s1+s2)

BiFunction<String, String, Integer>

BiPredicate takes two arguments

UnaryOperator<String> unary =

BinaryOperator<String> binary = (S1,s2) -> s1 + "" + s2

specilization for premitive type

To create an immutable class in Java, you need to follow these steps:

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don't provide setter methods for variables.
4. Make all mutable fields final so that their value can be assigned only once.
5. Initialize all fields via a constructor.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

Comparable and Comparator interfaces are used to sort collections, but there is some differences b/w them:

* Comparable:
  + It is a part of java.lang package.
  + It provides a single sorting sequence. In other words, we can sort the collection on the basis of a single element such as id, name, and price.
  + The compareTo(Object obj) method is used for sorting elements.
  + We can sort the list elements of Comparable type by Collections.sort(List) method.
  + A class needs to implement Comparable interface and override compareTo() method.
* Comparator:
  + It is a part of java.util package.
  + It provides multiple sorting sequences. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price.
  + The compare(Object obj1, Object obj2) method is used for sorting elements.
  + We can sort the list elements of Comparator type by Collections.sort(List, Comparator) method.
  + Comparator is typically used when we want to provide multiple ways to sort objects, or when we want to sort objects of a class that did not implement Comparable.

In summary, if sorting of objects needs to be based on natural order then use Comparable whereas if you need to sort objects based on different attributes then use Comparator in Java.

### ****1. What are Generics in Java?****

Generics are a feature in Java that allows you to define classes, interfaces, and methods with type parameters. This enables you to create code that is more reusable and type-safe. For example, you can create a List that can hold any type of objects, but it will ensure that only objects of a specified type are added.

**Example:**

public class Box<T> {

private T value;

public void setValue(T value) {

this.value = value;

}

public T getValue() {

return value;

}

}

In this example, T is a type parameter that will be replaced with a concrete type when an instance of Box is created.

### ****2. Why use Generics?****

Generics offer several advantages:

* **Type Safety**: Generics enforce type constraints at compile-time, reducing runtime errors caused by type mismatches.
* **Code Reusability**: Generics allow you to write more flexible and reusable code. You can use the same class, interface, or method with different types.
* **Elimination of Casts**: With generics, you don't need explicit type casts when retrieving items from a collection.

### ****3. How do Generics Improve Type Safety?****

Generics improve type safety by ensuring that only the specified type can be used with a class or method. The compiler enforces this constraint, reducing the likelihood of ClassCastException at runtime.

**Example:**

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

list.add("Hello");

String str = list.get(0); // No cast needed

### ****4. What is Type Erasure?****

Type erasure is a process used by the Java compiler to implement generics. During compilation, generic type information is erased and replaced with raw types, and generic type parameters are replaced with their bounds or Object if unbounded.

**Example:**

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

At runtime, the List<String> is treated as a List, and the compiler inserts type casts where necessary.

**5. What is a Wildcard in Generics?**

Wildcards are used to represent an unknown type in generics. They are useful for writing flexible and reusable code. There are three types of wildcards:

**Unbounded Wildcard**: <?> - Represents any type.

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

**Bounded Wildcard**: <? extends T> - Represents a type that is a subclass of T (inclusive).

List<? extends Number> list = new ArrayList<Integer>();

**Lower Bounded Wildcard**: <? super T> - Represents a type that is a superclass of T (inclusive).

List<? super Integer> list = new ArrayList<Number>();

### ****6. Can Generics be used with Primitive Types?****

No, generics in Java cannot be used with primitive types (e.g., int, char). Instead, you should use their wrapper classes (e.g., Integer, Character).

**Example:**

List<Integer> intList = new ArrayList<>();

### ****7. What is a Generic Method?****

A generic method is a method that has its own type parameters. The type parameters are declared before the return type of the method.

**Example:**

public <T> void printArray(T[] array) {

for (T element : array) {

System.out.println(element);

}

}

### ****8. Can Generic Methods be Static?****

Yes, generic methods can be static. The type parameters are specified in the method declaration and are not tied to the instance of the class.

**Example:**

public static <T> void printArray(T[] array) {

for (T element : array) {

System.out.println(element);

}

}

### ****9. What is a Generic Class?****

A generic class is a class that can operate on objects of various types while providing compile-time type safety. The type parameter is specified in the class definition.

**Example:**

public class Pair<T, U> {

private T first;

private U second;

public Pair(T first, U second) {

this.first = first;

this.second = second;

}

public T getFirst() {

return first;

}

public U getSecond() {

return second;

}

}

### ****10. Can a Generic Class Implement an Interface with Generics?****

Yes, a generic class can implement an interface that has generics. You need to specify the type parameters in the implementation.

**Example:**

public class MyList<T> implements List<T> {

// Implement methods here

}

### ****11. What is the Difference Between**** List<T> ****and**** List<?>****?****

* List<T>: A list that holds elements of type T. You can add and retrieve elements of type T.
* List<?>: A list with an unknown type. You can retrieve elements from it, but you cannot add elements (except null).

**Example:**

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

List<?> unknown = strings;

String s = (String) unknown.get(0); // Allowed

unknown.add("Hello"); // Not allowed

### ****12. Can Generics be Used with Constructors?****

Yes, you can use generics in constructors. The type parameters of the class can be used within the constructor.

**Example**

public class Box<T> {

private T value;

public Box(T value) {

this.value = value;

}

public T getValue() {

return value;

}

}

### ****13. What are Type Parameters?****

Type parameters are the placeholders for types used in generic classes, interfaces, and methods. They are specified within angle brackets (<T>) and replaced with actual types when an instance is created.

**Example:**

public class MyClass<T> {

private T data;

public MyClass(T data) {

this.data = data;

}

}

### ****14. What are Generic Constraints?****

Generic constraints are used to restrict the types that can be used as type arguments. You can specify bounds on type parameters using the extends keyword.

**Example:**

public <T extends Number> void process(T number) {

// T must be a subclass of Number

}

### ****15. Can You Create Arrays of Generic Types?****

No, you cannot directly create arrays of generic types due to type erasure and the limitations of generic arrays. You can use collections instead.

**Example:**

List<String>[] stringLists = new ArrayList[10]; // Unsafe

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

### ****Summary****

Generics in Java enhance code reusability, type safety, and readability. Understanding how to use them effectively involves knowing how to define and use generic classes, methods, and interfaces, as well as understanding limitations and how to work with wildcards and type bounds.

Java 17, released in September 2021, is a Long-Term Support (LTS) release and includes several important features and improvements over previous versions. Here’s a summary of the key features and enhancements introduced in Java 17:

### 1. ****Sealed Classes (JEP 409)****

* **Concept**: Sealed classes restrict which other classes or interfaces can extend or implement them. This provides more control over the class hierarchy and can improve security and maintainability.
* **Example**:

java