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

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);  }  }); |
| **With Java 8: Using a lambda expression**  Runnable r2 = () -> System.out.println("Hello World!"); | **Before Java 8: Using an anonymous class**  Runnable r1 = new Runnable() {  @Override  public void run() {  System.out.println("Hello World!");  }  }; |

1. **Functional Interfaces:**

* 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.

1. **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.

1. **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.

1. **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:

|  |  |
| --- | --- |
| 1. Static Method Reference: 2. Instance Method Reference of a Particular Object: 3. Instance Method Reference of an Arbitrary Object of a Particular Type: 4. Constructor Reference: | 1. ClassName::staticMethodName 2. instance::instanceMethodName 3. ClassName::instanceMethodName 4. 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();

1. **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.

1. **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.

1. **Streams for Collections**

* 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);

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

@FunctionalInterface

public interface MyFunctionalInterface {

void execute();

}

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

func.execute();

**What are Default Methods in interfaces?**

* 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:

public interface MyInterface {

default void newMethod() {

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

}

}

**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));

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

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());

**What is difference between map() and flatmap()?**

1. **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**.

1. **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**.

|  |  |  |
| --- | --- | --- |
| **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 | It 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. | It 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. |

**What are Collectors in Java 8 Streams?**

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);