**Java 8 Features and Interview Questions**

### ****Java 8 Features:****

**Lambda Expressions** - Enables functional programming by allowing concise syntax for anonymous functions.

**Detailed Explanation:**

* + Lambda expressions provide a clear and concise way to implement functional interfaces.
  + They allow passing behavior as parameters and reduce boilerplate code.
  + A lambda expression has three main parts:
    - Parameters: (parameter1, parameter2, ...)
    - Arrow Token: ->
    - Body: { statement(s) }

**Real-time Example:** Suppose we are developing a system where we need to sort a list of employees based on their salary.

import java.util.\*;

class Employee {

String name;

int salary;

public Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

public String toString() {

return name + " - " + salary;

}

}

public class LambdaRealTimeExample {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", 50000),

new Employee("Bob", 60000),

new Employee("Charlie", 40000)

);

// Using lambda expression to sort employees by salary

employees.sort((e1, e2) -> Integer.compare(e1.salary, e2.salary));

// Display sorted employees

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

}

}

**Output:**

Charlie - 40000

Alice - 50000

Bob - 60000

* + Here, the lambda expression (e1, e2) -> Integer.compare(e1.salary, e2.salary) is used to define a custom comparator for sorting employees by salary.
  + This eliminates the need for creating an anonymous inner class, making the code more readable and concise.

**Functional Interfaces** - Introduces @FunctionalInterface annotation and built-in interfaces like Predicate, Consumer, and Supplier.

**Detailed Explanation:**

* + A functional interface is an interface with a single abstract method.
  + It can have multiple default and static methods.
  + Common built-in functional interfaces:
    - 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.
    - Supplier<T>: Represents a supplier of results.

**Real-time Example:** Suppose we need to filter out employees who earn more than 50,000 using Predicate.

import java.util.Arrays;

import java.util.List;

import java.util.function.Predicate;

class Employee {

String name;

int salary;

public Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

public String toString() {

return name + " - " + salary;

}

}

public class FunctionalInterfaceExample {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", 50000),

new Employee("Bob", 60000),

new Employee("Charlie", 40000)

);

Predicate<Employee> highSalaryPredicate = emp -> emp.salary > 50000;

employees.stream()

.filter(highSalaryPredicate)

.forEach(System.out::println);

}

}

**Output:**

Bob - 60000

* + Here, we use the Predicate functional interface to filter employees based on salary.
  + The stream().filter(highSalaryPredicate) method processes the list and filters the employees who meet the criteria.

**Streams API** - Facilitates bulk operations on collections using functional programming.

**Detailed Explanation:**

* + The Stream API provides a functional approach to processing collections.
  + It allows operations such as filtering, mapping, sorting, and reducing collections efficiently.
  + Streams support parallel execution to improve performance.
  + Streams do not modify the original collection but produce a new one.

**Real-time Example:** Suppose we need to filter and sort a list of employees based on salary.

import java.util.\*;

import java.util.stream.Collectors;

class Employee {

String name;

int salary;

public Employee(String name, int salary) {

this.name = name;

this.salary = salary;

}

public String toString() {

return name + " - " + salary;

}

}

public class StreamExample {

public static void main(String[] args) {

List<Employee> employees = Arrays.asList(

new Employee("Alice", 50000),

new Employee("Bob", 60000),

new Employee("Charlie", 40000),

new Employee("David", 70000)

);

// Filtering employees with salary greater than 50,000 and sorting them

List<Employee> filteredEmployees = employees.stream()

.filter(e -> e.salary > 50000)

.sorted(Comparator.comparingInt(e -> e.salary))

.collect(Collectors.toList());

// Display the filtered and sorted employees

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

}

}

**Output:**

Bob - 60000

David - 70000

* + Here, stream().filter(e -> e.salary > 50000) filters employees earning more than 50,000.
  + sorted(Comparator.comparingInt(e -> e.salary)) sorts them by salary.
  + collect(Collectors.toList()) collects the results into a new list.

**Default and Static Methods in Interfaces** - Allows defining method implementations inside interfaces.

**Detailed Explanation:**

* + Before Java 8, interfaces could only have abstract methods.
  + Java 8 introduced default and static methods, allowing method implementations within interfaces.
  + Default methods help in extending interfaces without breaking existing implementations.
  + Static methods allow utility methods to be added directly in interfaces.

**Real-time Example:** Suppose we have an interface with a default method and a static method.

interface MyInterface {

// Default method

default void show() {

System.out.println("Default Method in Interface");

}

// Static method

static void staticMethod() {

System.out.println("Static Method in Interface");

}

}

public class DefaultStaticMethodExample implements MyInterface {

public static void main(String[] args) {

DefaultStaticMethodExample obj = new DefaultStaticMethodExample();

obj.show(); // Calls default method

MyInterface.staticMethod(); // Calls static method

}

}

**Output:**

Default Method in Interface

Static Method in Interface

**Method References** - Provides a shorthand for lambda expressions referring to methods by their names.

**Detailed Explanation:**

* + Method references simplify lambda expressions by directly referring to existing methods.
  + They can be used to refer to static methods, instance methods, or constructors.
  + They improve readability and make the code more concise.

**Real-time Example:** Suppose we want to print a list of names using method references.

import java.util.Arrays;

import java.util.List;

public class MethodReferenceExample {

public static void main(String[] args) {

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

// Using lambda expression

names.forEach(name -> System.out.println(name));

// Using method reference

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

}

}

**Output:**

Alice

Bob

Charlie

* + System.out::println is a method reference that replaces name -> System.out.println(name).
  + It directly refers to the println method of System.out, making the code more readable.

**Optional Class** - Helps in handling NullPointerException issues in Java.

**Detailed Explanation:**

* + The Optional class was introduced to handle null values safely.
  + It helps avoid NullPointerException by providing methods to check the presence of a value.
  + Optional.of(value), Optional.empty(), and Optional.ofNullable(value) are commonly used methods.

**Real-time Example:** Suppose we need to safely retrieve an employee's name.

import java.util.Optional;

class Employee {

String name;

public Employee(String name) {

this.name = name;

}

public String getName() {

return name;

}

}

public class OptionalExample {

public static void main(String[] args) {

Employee emp = new Employee("Alice");

Employee empNull = null;

// Using Optional to avoid NullPointerException

Optional<Employee> optionalEmp = Optional.ofNullable(empNull);

String name = optionalEmp.map(Employee::getName).orElse("Unknown");

System.out.println("Employee Name: " + name);

}

}

**Output:**

Employee Name: Unknown

* + Optional.ofNullable(empNull) avoids null reference errors.
  + map(Employee::getName) safely extracts the value.
  + orElse("Unknown") provides a default value when the optional is empty.

**New Date and Time API** - Introduces java.time package for better date-time handling.

**Real-time Example:**

import java.time.LocalDate;

import java.time.LocalTime;

import java.time.LocalDateTime;

public class DateTimeExample {

public static void main(String[] args) {

LocalDate date = LocalDate.now();

LocalTime time = LocalTime.now();

LocalDateTime dateTime = LocalDateTime.now();

System.out.println("Current Date: " + date);

System.out.println("Current Time: " + time);

System.out.println("Current Date and Time: " + dateTime);

}

}

**Output:**

Current Date: 2025-03-05

Current Time: 14:30:15.123

Current Date and Time: 2025-03-05T14:30:15.123

* + LocalDate.now() fetches the current date.
  + LocalTime.now() fetches the current time.
  + LocalDateTime.now() fetches both date and time.

### Collectors and Parallel Streams - Allows efficient data collection and parallel processing

**Detailed Explanation:**

* Collectors provide a mechanism to accumulate the results of stream operations into collections (e.g., List, Set, Map) or other data structures.
* Parallel Streams enable processing of large datasets across multiple CPU cores, potentially improving performance for computationally intensive tasks.
* A stream pipeline typically includes a source, intermediate operations (e.g., filter, map), and a terminal operation (e.g., collect or sum).
* Parallel streams are invoked using parallelStream() instead of stream(), but care must be taken as they introduce overhead for small datasets.

**Real-time Example:** Suppose we are developing a system to process a large list of sales transactions and calculate the total sales amount for transactions above $100, both sequentially and in parallel.

import java.util.Arrays;

import java.util.List;

import java.util.stream.Collectors;

class Transaction {

String id;

double amount;

public Transaction(String id, double amount) {

this.id = id;

this.amount = amount;

}

public double getAmount() {

return amount;

}

public String toString() {

return id + " - $" + amount;

}

}

public class CollectorsParallelExample {

public static void main(String[] args) {

List<Transaction> transactions = Arrays.asList(

new Transaction("T1", 50.0),

new Transaction("T2", 150.0),

new Transaction("T3", 200.0),

new Transaction("T4", 75.0)

);

// Sequential stream: Filter and collect transactions > $100

List<Transaction> highValueSequential = transactions.stream()

.filter(t -> t.getAmount() > 100).collect(Collectors.toList());

System.out.println("Sequential High-Value Transactions:");

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

// Parallel stream: Calculate total sales for transactions > $100

double totalSalesParallel = transactions.parallelStream()

.filter(t -> t.getAmount() > 100).mapToDouble(Transaction::getAmount).sum();

System.out.println("Total Sales (Parallel): $" + totalSalesParallel);

}

}

**Output:**  
Sequential High-Value Transactions:  
T2 - $150.0  
T3 - $200.0  
Total Sales (Parallel): $350.0

* Here, stream().filter(t -> t.getAmount() > 100).collect(Collectors.toList()) filters and collects high-value transactions sequentially.
* parallelStream().filter(...).mapToDouble(...).sum() computes the total sales amount in parallel, leveraging multiple cores for larger datasets.

### Improved Type Inference - Enhancements in type inference for better lambda expression usage

**Detailed Explanation:**

* Improved type inference in Java 8 allows the compiler to deduce parameter types in lambda expressions based on the context (e.g., functional interface).
* This reduces boilerplate code, making lambda expressions more concise and readable.
* Type inference applies to generic methods and lambda expressions, enhancing their usability in functional programming.

**Real-time Example:** Suppose we are developing a system to sort a list of products by price.

import java.util.Arrays;

import java.util.List;

class Product {

String name;

double price;

public Product(String name, double price) {

this.name = name;

this.price = price;

}

public double getPrice() {

return price;

}

public String toString() {

return name + " - $" + price;

}

}

public class TypeInferenceExample {

public static void main(String[] args) {

List<Product> products = Arrays.asList(

new Product("Laptop", 999.99),

new Product("Phone", 499.99),

new Product("Tablet", 299.99)

);

// Using lambda with type inference to sort by price

products.sort((p1, p2) -> Double.compare(p1.getPrice(), p2.getPrice()));

// Display sorted products

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

}

}

**Output:**  
Tablet - $299.99  
Phone - $499.99  
Laptop - $999.99

* Here, (p1, p2) -> Double.compare(p1.getPrice(), p2.getPrice()) uses type inference; the compiler infers p1 and p2 as Product types based on the Comparator<Product> context in sort.
* This eliminates the need to explicitly write (Product p1, Product p2).

### Nashorn JavaScript Engine - A new engine to execute JavaScript code within Java applications

**Detailed Explanation:**

* Nashorn is a lightweight JavaScript engine introduced in Java 8, replacing the older Rhino engine.
* It integrates with the javax.script API, enabling Java applications to execute JavaScript dynamically.
* Useful for scripting, prototyping, or embedding JavaScript logic within Java programs.

**Real-time Example:** Suppose we are developing a system that evaluates a JavaScript function to calculate discounts.

import javax.script.ScriptEngine;

import javax.script.ScriptEngineManager;

import javax.script.ScriptException;

public class NashornExample {

public static void main(String[] args) throws ScriptException {

ScriptEngineManager manager = new ScriptEngineManager();

ScriptEngine engine = manager.getEngineByName("nashorn");

String script = "function calculateDiscount(price) { return price \* 0.9; }; calculateDiscount(100);";

Object result = engine.eval(script);

System.out.println("Discounted Price: $" + result);

}

}

**Output:**  
Discounted Price: $90.0

* Here, the JavaScript function calculateDiscount is defined and executed using Nashorn, returning a 10% discount on $100.
* engine.eval(script) runs the JavaScript code, integrating it seamlessly with Java.

### Base64 Encoding and Decoding - Utility class for encoding and decoding Base64 data

**Detailed Explanation:**

* The java.util.Base64 class in Java 8 provides methods to encode binary data into Base64 strings and decode them back.
* Base64 is widely used for data transmission (e.g., in emails, APIs) to represent binary data as ASCII text.
* Methods include Base64.getEncoder() for encoding and Base64.getDecoder() for decoding.

**Real-time Example:** Suppose we are developing a system to encode and decode user credentials for secure transmission.

import java.util.Base64;

public class Base64Example {

public static void main(String[] args) {

String credentials = "username:password";

// Encoding credentials to Base64

String encoded = Base64.getEncoder().encodeToString(credentials.getBytes());

System.out.println("Encoded Credentials: " + encoded);

// Decoding Base64 back to original string

byte[] decodedBytes = Base64.getDecoder().decode(encoded);

String decoded = new String(decodedBytes);

System.out.println("Decoded Credentials: " + decoded);

}

}

**Output:**  
Encoded Credentials: dXNlcm5hbWU6cGFzc3dvcmQ=  
Decoded Credentials: username:password

* Here, Base64.getEncoder().encodeToString() converts the string to Base64, and Base64.getDecoder().decode() reverses the process.
* This ensures secure and portable data transmission.

### Enhanced Concurrency API - Improvements in concurrency with new features like CompletableFuture

**Detailed Explanation:**

* Java 8 introduced CompletableFuture to the java.util.concurrent package, enhancing asynchronous programming.
* It allows chaining operations, handling results or exceptions, and executing tasks concurrently.
* Useful for non-blocking operations, such as fetching data from multiple sources simultaneously.

**Real-time Example:** Suppose we are developing a system to fetch and process stock prices asynchronously.

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutionException

public class CompletableFutureExample {

public static void main(String[] args) throws ExecutionException, InterruptedException {

CompletableFuture<Double> stockPriceFuture = CompletableFuture.supplyAsync(() -> {

try {

Thread.sleep(1000); // Simulate API call delay

return 150.75; // Simulated stock price

} catch (InterruptedException e) {

return 0.0;

}

});

System.out.println("Fetching stock price...");

stockPriceFuture.thenAccept(price -> System.out.println("Stock Price: $"+price));

// Wait for completion to see the output

stockPriceFuture.get();

}

}

**Output:**  
Fetching stock price...  
Stock Price: $150.75

* Here, CompletableFuture.supplyAsync() runs the task asynchronously, simulating a delay for fetching a stock price.
* thenAccept() processes the result once available, demonstrating non-blocking execution.