Practise Questions with Answers

Java Latest Features

Q1. You are writing a utility method that processes a collection of numeric values.

- How would you restrict the generic type to only numbers?

Answer :

Restricting a Generic Type to Only Numbers

To ensure that a generic type accepts only numeric types, use an upper bounded type parameter:

```
public static <T extends Number> double sum(List<T> list) {
    double total = 0;
    for (T num : list) {
        total += num.doubleValue();
    }
    return total;
}
```

Why this works

- T extends Number restricts T to:
    - Integer
    - Double
    - Float
    - Long, etc.
- Ensures access to methods of Number like doubleValue().

- Explain when to use <? extends Number> vs <? super Integer>.
- Give one practical use case for each

Answer :

A) <? extends Number> – Producer

List<? extends Number> numbers;

Meaning

- The list produces values of type Number
- Can hold List<Integer>, List<Double>, etc.

Rules:

- Can read elements as Number
- Cannot add elements (except null)

Ex: Number n = numbers.get(0); // Allowed

numbers.add(10); // Compile-time error

When to use-

- When you only read data
- When method should accept any numeric subtype

Practical Use Case-

```
public static void printNumbers(List<? extends Number> list) {
   for (Number n : list) {
      System.out.println(n);
   }
} Works for List<Integer>, List<Double>, etc.
```

B) <? super Integer> – Consumer

List<? super Integer> numbers;

Meaning

- The list consumes Integer values
- Can be:
    - List<Integer>
    - List<Number>
    - List<Object>

Rules:

- Can add Integer
- Reading gives Object type

Code - numbers.add(10); // Allowed

Object obj = numbers.get(0); // Only Object

When to use

- When you add/write elements
- When inserting Integer values into a broader collection

Practical Use Case

```
public static void addIntegers(List<? super Integer> list) {
   list.add(1);
   list.add(2);
}
```

Can add integers safely into multiple compatible list

Q2. Java introduced **records** to simplify data-carrying classes.

- What problems do records solve compared to traditional POJOs?

Answer : Records (introduced in Java 16) are designed for simple data-carrier classes.
They solve several common issues with traditional POJOs.

Problems with POJOs

A typical POJO requires a lot of boilerplate:

```
class Employee {
    private final int id;
    private final String name;

    // constructor
    public Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    // getters
    public int getId() { return id; }
    public String getName() { return name; }

    // equals, hashCode, toString
}
```

Problems:

- Too much boilerplate code

- Easy to forget equals(), hashCode(), or toString()
- Intention (just holding data) is not clear
- Error-prone and verbose

How Records Solve These Problems

public record Employee(int id, String name) {}

Records automatically provide:

- private final fields
- Canonical constructor
- Getters (id(), name())
- equals(), hashCode(), toString()
- Clear intent: immutable data holder

- When would you **not** use a record?

Answer : Records are not a replacement for all classes.

Do NOT use records when:

1. You Need Mutability

Records are immutable.

record User(String name) {}

// name cannot be changed

If setters are required → use a class

2. You Need Inheritance

Records:

- Cannot extend other classes
- Implicitly extend java.lang.Record
- record A(int x) extends B {} // Not allowed

3. You Have Complex Business Logic

Records are meant for data, not behavior-heavy objects.

Bad fit:

- Service classes
- Domain models with changing state
- Objects with many validation rules

4. You Need No-Args Constructor

Records must define all components.

```
record Product(int id) {}
// No default constructor allowed
```

5. Framework Limitations

Some frameworks expect:

- Mutable fields
- Proxy creation
- No-args constructor

Examples:

- Older versions of JPA / Hibernate
- Serialization frameworks without record support

Q3. What is a **functional interface**?

Answer :A functional interface is an interface that has exactly one abstract method.

It may have:

- Any number of default methods
- Any number of static methods
  It is the foundation for lambda expressions and method references (Java 8+)

```
@FunctionalInterface
interface Calculator {
   int add(int a, int b);
}
```

@FunctionalInterface is optional but helps the compiler catch errors.

- Name two built-in functional interfaces.

Answer : Java provides many functional interfaces in java.util.function.

1. Predicate<T>

- Takes one argument
- Returns boolean
- Used for conditions / filtering

Predicate<Integer> isEven = n -> n % 2 == 0;

2. Function<T, R>

- Takes one argument
- Returns a result
- Used for data transformation

Function<String, Integer> length = s -> s.length();

- Show how a lambda expression improves readability compared to an anonymous class.

Answer :

Using an Anonymous Class (Before Java 8)

```
Runnable task = new Runnable() {
    @Override
    public void run() {
        System.out.println("Task is running");
    }
};
```

Problems:

- Too much boilerplate code

- new, class name, and @Override hide the actual logic
- Harder to quickly understand intent

Using a Lambda Expression (Java 8+)

```
Runnable task = () -> System.out.println("Task is running");
```

Improvements:

- No class name or method declaration
- Focuses directly on the behavior
- Short, clean, and expressive

Another Common Example – Comparator

Anonymous Class

```
Comparator<String> comp = new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return a.length() - b.length();
    }
};
```

Lambda Expression

```
Comparator<String> comp = (a, b) -> a.length() - b.length();
```

Lambda expressions remove boilerplate code from anonymous classes, making the code shorter, clearer, and easier to read.

Q4. What is a **Stream** in Java?

Answer: A Stream in Java (introduced in Java 8) is a sequence of elements from a data source that supports functional-style operations such as filtering, mapping, and reducing.

```
list.stream()
   .filter(n -> n > 10)
   .map(n -> n * 2)
   .forEach(System.out::println);
```

- Focuses on what to do, not how to do it
- Supports declarative programming
- Can be sequential or parallel

- How is it different from a collection?

Answer: A Collection in Java is primarily a data structure whose main purpose is to store and manage elements in memory, such as a List, Set, or Map.
A Stream, on the other hand, is not a data structure and does not store elements. Instead, it represents a pipeline for processing data obtained from a data source like a collection, array, or I/O channel.

Collections hold data eagerly, can be iterated multiple times, and allow direct modification of elements (add, remove, update).

Streams process data in a lazy, functional, and declarative way, where operations such as filter, map, and reduce describe what should be done rather than how to do it.

Iteration over a collection is usually external, meaning the developer controls how the iteration happens using loops.

Stream operations are executed only when a terminal operation is invoked, making them efficient and optimized through short-circuiting. Unlike collections, streams are single-use, support internal iteration, and emphasize immutability and readability over data storage.

Example-

```
// Collection: stores data

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



// Stream: processes data

numbers.stream()

    .filter(n -> n % 2 == 0)

    .map(n -> n * 2)

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

Here, the collection numbers holds the data, while the stream processes that data lazily—filtering and transforming elements only when forEach() is called.

- Explain the difference between **intermediate** and **terminal**operations.

Answer:

A) Intermediate Operations

- Return another Stream
- Are lazy
- Do not execute until a terminal operation is called

Examples:

filter()

map()

sorted()

distinct()

limit()

```
Stream<Integer> s = list.stream()
    .filter(n -> n > 5)
    .map(n -> n * 2);
```

No execution yet

B) Terminal Operations

- Produce a result or side effect
- Trigger stream processing
- End the stream

Examples:

forEach()

collect()

reduce()

count()

findFirst()

```
List<Integer> result = list.stream()
    .filter(n -> n > 5)
    .map(n -> n * 2)
```

.collect(Collectors.toList());

Execution happens here.

- Why are streams considered lazy?

Answer: Streams are lazy because:

- Intermediate operations are not executed immediately
- Elements are processed only when needed
- Processing happens one element at a time, not step-by-step per operation

Example:

```
list.stream()
   .filter(n -> {
      System.out.println("Filtering " + n);
      return n > 5;
   })
   .map(n -> {
      System.out.println("Mapping " + n);
      return n * 2;
   })
   .findFirst();
```

Execution stops as soon as findFirst() gets one match.

- Improves performance
- Avoids unnecessary computation

- Enables short-circuiting operations (findFirst, anyMatch, limit)

Q5. Given a list of employee objects:

- Which stream operations would you use to filter employees by department?

Answer:

Stream Operations on Employee Objects

Assume we have an Employee class:

```
class Employee {
   private String name;
   private String department;

   // constructors, getters
   public String getName() { return name; }
   public String getDepartment() { return department; }
}
```

Filter Employees by Department

Use the filter() intermediate operation.

```
List<Employee> itEmployees =
        employees.stream()
                .filter(e ->
"IT".equalsIgnoreCase(e.getDepartment()))
                .toList();   // Java 16+
```

- filter() is used to select elements based on a condition.

- How would you transform employee names to uppercase?

Answer:

Transform Employee Names to Uppercase

- Use the map() intermediate stream operation.
- map() is used to transform each element in a stream.
- It does not modify the original list.
- Returns a new stream with transformed elements.
- Commonly used for data formatting and normalization.

Example:

```
List<String> upperCaseNames =
        employees.stream()
                .map(emp -> emp.getName().toUpperCase())
                .toList();   // Java 16+
```

Explanation:

- employees.stream() creates a stream from the collection.
- map() converts each employee's name to uppercase.
- toList() collects the result into a new list.

- Which collector would you use to group employees by department?

Answer: Collector used: Collectors.groupingBy()

- groupingBy() is used to group stream elements based on a key
- It returns a Map<K, List<V>>
- The key is derived using a classifier function
- Commonly used for categorizing data
- Supports downstream collectors (like mapping, counting, averaging)

Basic Example:

```
Map<String, List<Employee>> employeesByDept =
    employees.stream()
        .collect(Collectors.groupingBy(Employee::getDepartment));
```

- Groups employees by department
- Department name becomes the key
- List of employees becomes the value