Question 1

In a log parser, you write this inside a method: var logLines = Files.readAllLines(path); What is true?

1) Java decides the type of logLines at runtime
2) logLines can later become a different type (like a String)
3) The compiler infers a fixed type for logLines at compile time
4) var can only be used for fields, not inside methods

Reply format: 1 / 2 / 3 / 4

Answer 1

In a log parser, you write this inside a method: var logLines = Files.readAllLines(path); What is true?

1) Java decides the type of logLines at runtime
2) logLines can later become a different type (like a String)
3) The compiler infers a fixed type for logLines at compile time
4) var can only be used for fields, not inside methods

Correct answer: 3
Why: var is compile-time type inference for local variables; Java remains statically typed.

```
import java.nio.file.*;
import java.util.*;

class LogFileReader {
    List<String> read(Path path) throws Exception {
        var logLines = Files.readAllLines(path); // inferred as List<String>
        // logLines = "oops"; // does not compile
        return logLines;
    }
}
```

Reply format: 1 / 2 / 3 / 4

Question 2

You are mapping payment statuses to UI labels. What is a key advantage of switch expressions?

1) They are only allowed with int
2) They must always use break
3) They can return a value directly
4) They cannot have a default case

Reply format: 1 / 2 / 3 / 4

Answer 2

You are mapping payment statuses to UI labels. What is a key advantage of switch expressions?

1) They are only allowed with int
2) They must always use break
3) They can return a value directly
4) They cannot have a default case

Correct answer: 3

Why: A switch expression returns a value, which makes mapping logic concise and less error prone.

```
class PaymentUiMapper {
    String label(String status) {
        return switch (status) {
            case "SUCCESS" -> "Paid";
            case "PENDING" -> "Processing";
            case "FAILED"  -> "Failed";
            default        -> "Unknown";
        };
    }
}
```

Reply format: 1 / 2 / 3 / 4

Question 3

You model an immutable API response as record TransactionSummary(...). Which statement is true?

1) A record can never have additional methods

2) A record automatically generates equals, hashCode, toString, and accessor methods

3) A record must extend another class

4) A record allows multiple inheritance

Reply format: 1 / 2 / 3 / 4

Answer 3

You model an immutable API response as record TransactionSummary(...). Which statement is true?

1) A record can never have additional methods

2) A record automatically generates equals, hashCode, toString, and accessor methods

3) A record must extend another class

4) A record allows multiple inheritance

Correct answer: 2

Why: Records generate common boilerplate automatically and can still include your own methods.

```
record TransactionSummary(String txnId, long amountPaise, String status) {
    boolean isSuccess() {
        return "SUCCESS".equals(status);
    }
}

class RecordDemo {
    void demo() {
        var t1 = new TransactionSummary("TXN123", 15000, "SUCCESS");
        System.out.println(t1.txnId());     // accessor
        System.out.println(t1);             // toString auto generated
        System.out.println(t1.isSuccess()); // custom method
    }
}
```

Reply format: 1 / 2 / 3 / 4

Question 4

You model outcomes as sealed interface ImportResult permits Success, Skipped, Failed. Why is this useful?

1) It makes all methods static by default
2) It restricts who can implement or extend the type
3) It automatically logs errors
4) It prevents exceptions from being thrown

Reply format: 1 / 2 / 3 / 4

Answer 4

You model outcomes as sealed interface ImportResult permits Success, Skipped, Failed. Why is this useful?

1) It makes all methods static by default

2) It restricts who can implement or extend the type

3) It automatically logs errors

4) It prevents exceptions from being thrown

Correct answer: 2

Why: Sealed types prevent unexpected implementations and pair well with switch for exhaustive handling.

```
sealed interface ImportResult permits Success, Skipped, Failed {}

record Success(int rowsImported) implements ImportResult {}
record Skipped(String reason) implements ImportResult {}
record Failed(String error) implements ImportResult {}

class ImportHandler {
    String summary(ImportResult r) {
        return switch (r) {
            case Success s -> "Imported " + s.rowsImported() + " rows";
            case Skipped s -> "Skipped: " + s.reason();
            case Failed f  -> "Failed: " + f.error();
        };
    }
}
```

Reply format: 1 / 2 / 3 / 4

Question 5

You are storing a multi line JSON template in code for a config export. Why use text blocks?

1) They make strings encrypted
2) They compress the string to save memory
3) They improve readability and reduce escaping and concatenation
4) They prevent null values

Reply format: 1 / 2 / 3 / 4

Answer 5

You are storing a multi line JSON template in code for a config export. Why use text blocks?

1) They make strings encrypted
2) They compress the string to save memory
3) They improve readability and reduce escaping and concatenation
4) They prevent null values

Correct answer: 3

Why: Text blocks keep multi-line strings readable and avoid messy escaping and plus concatenations.

```
class ConfigExporter {
    String jsonTemplate(String env) {
        return """
            {
              "service": "reporting",
              "environment": "%s",
              "enabled": true
            }
            """.formatted(env);
    }
}
```

Reply format: 1 / 2 / 3 / 4

Question 6

You add @Override on a method in FileReportGenerator implementing an interface. What does it protect you from most?

1) Slow performance
2) Accidentally not matching the parent method signature
3) Memory leaks
4) Compilation becoming slower

Reply format: 1 / 2 / 3 / 4

Answer 6

You add @Override on a method in FileReportGenerator implementing an interface. What does it protect you from most?

1) Slow performance

2) Accidentally not matching the parent method signature

3) Memory leaks

4) Compilation becoming slower

Correct answer: 2

Why: @Override makes the compiler catch signature mistakes early.

```
interface ReportGenerator {
    String generate(String fileName);
}

class FileReportGenerator implements ReportGenerator {
    @Override
    public String generate(String fileName) {
        return "Report for: " + fileName;
    }

    // If the signature is wrong, @Override causes a compile error.
}
```

Reply format: 1 / 2 / 3 / 4

Question 7

You are reading multiple log files and opening a reader. What is the main benefit of try with resources?

1) It makes the file reads parallel automatically
2) It guarantees the resource closes even if an exception happens
3) It disables checked exceptions
4) It prevents file not found errors

Reply format: 1 / 2 / 3 / 4

Answer 7

You are reading multiple log files and opening a reader. What is the main benefit of try with resources?

1) It makes the file reads parallel automatically

2) It guarantees the resource closes even if an exception happens

3) It disables checked exceptions

4) It prevents file not found errors

Correct answer: 2

Why: The resource is closed automatically, even if an exception happens inside the try block.

```
import java.nio.file.*;

class SafeFileRead {
    int countLines(Path path) throws Exception {
        try (var br = Files.newBufferedReader(path)) { // auto closed
            int count = 0;
            while (br.readLine() != null) count++;
            return count;
        }
    }
}
```

Reply format: 1 / 2 / 3 / 4

Question 8

A class implements Auditable and Trackable, and both define the same default method getAuditTag(). What must the class do?

1) Nothing; Java will pick one automatically
2) Remove one interface from the class
3) Override the method to resolve the conflict
4) Mark the method final inside the interfaces

Reply format: 1 / 2 / 3 / 4

Answer 8

A class implements Auditable and Trackable, and both define the same default method getAuditTag(). What must the class do?

1) Nothing; Java will pick one automatically

2) Remove one interface from the class

3) Override the method to resolve the conflict

4) Mark the method final inside the interfaces

Correct answer: 3

Why: When two interfaces provide the same default method, the class must override to remove ambiguity.

```
interface Auditable {
    default String getAuditTag() { return "AUDIT"; }
}
interface Trackable {
    default String getAuditTag() { return "TRACK"; }
}

class OrderEvent implements Auditable, Trackable {
    @Override
    public String getAuditTag() {
        return Auditable.super.getAuditTag() + "+" + Trackable.super.getAuditTag();
    }
}
```