Java 8 Features

Functional Interfaces

1. Define a functional interface `IntOperation` with a method `int apply(int x, int y)`. Implement this interface using lambda expressions to perform addition, subtraction, multiplication, and division.
   Approach: Create a functional interface with `int apply(int x, int y)`. Implement it using lambda expressions for each arithmetic operation. Test each implementation.
2. Create a functional interface `StringLength` with a method `int length(String s)`. Use a lambda expression to implement this interface to return the length of a given string.
   Approach: Define the interface with `int length(String s)`. Implement it using a lambda expression that returns `s.length()`. Test with different strings.
3. Implement a functional interface `Predicate<T>` to check if a string contains a given substring. Use a lambda expression for the implementation.
   Approach: Define a functional interface `Predicate<T>` with a method `boolean test(String s, String substring)`. Implement it using a lambda expression that checks substring presence.
4. Define a functional interface `Transformer` with a method `String transform(String input)`. Implement this interface using lambda expressions to convert the string to lowercase, uppercase, and title case.
   Approach: Create the interface with `String transform(String input)`. Use lambda expressions for lowercase, uppercase, and title case transformations.
5. Create a functional interface `MathOperation` with a method `double calculate(double a, double b)`. Implement this interface using lambda expressions to perform modulus and power operations.
   Approach: Define `MathOperation` with `double calculate(double a, double b)`. Implement using lambdas for modulus (`a % b`) and power (`Math.pow(a, b)`).

Stream API

6. Given a list of integers, use the Stream API to find the sum of all even numbers.
   Approach: Convert the list to a stream, filter even numbers, sum them using `mapToInt` and `sum()`.
7. Given a list of strings, use the Stream API to find the longest string.
   Approach: Convert the list to a stream, use `max()` with a comparator by length.
8. Given a list of names, use the Stream API to count the number of names starting with the letter "J".
   Approach: Convert the list to a stream, filter names starting with "J", and use `count()`.

9. Use the Stream API to find the average of a list of double values.
   Approach: Convert the list to a stream, use `mapToDouble`, and then `average()`.
10. Given a list of employees, use the Stream API to group employees by their department.
    Approach: Define an `Employee` class with `department`. Convert the list to a stream and use `collect(Collectors.groupingBy(Employee::getDepartment))`.

Lambda Expressions

11. Use lambda expressions to sort a list of integers in descending order.
    Approach: Convert the list to a stream, use `sorted` with a comparator for descending order.
12. Use a lambda expression to filter a list of strings to include only those that are non-empty.
    Approach: Convert the list to a stream, use `filter(s -> !s.isEmpty())`.
13. Create a lambda expression to find the maximum value in a list of integers.
    Approach: Convert the list to a stream, use `max(Integer::compareTo)`.
14. Implement a lambda expression to concatenate a list of strings with a comma separator.
    Approach: Convert the list to a stream, use `collect(Collectors.joining(","))`.
15. Use lambda expressions to map a list of integers to their squares.
    Approach: Convert the list to a stream, use `map(n -> n * n)`.

Optional Class

16. Create a method that returns an `Optional<String>` containing a value if a string is not null, otherwise return `Optional.empty()`.
    Approach: Use `Optional.ofNullable()` to wrap the string and return it.
17. Write a method that takes an `Optional<String>` and prints the string in uppercase if it is present, otherwise prints "No value".
    Approach: Use `ifPresentOrElse()` to handle both cases.
18. Given an `Optional<Double>`, return its value if present or a default value of 100.0 if absent.
    Approach: Use `orElse(100.0)` to provide the default value.
19. Implement a method that takes an `Optional<Integer>` and multiplies the value by 10 if present, otherwise returns 0.
    Approach: Use `map(n -> n * 10).orElse(0)`.
20. Create a method that takes an `Optional<List<String>>` and returns a list containing only non-empty strings.
    Approach: Use `flatMap` to handle the optional and filter non-empty strings.

Combining Concepts

21. Use the Stream API and Optional to safely calculate the average salary from a list of employee salaries, handling cases where the list might be empty.
    Approach: Convert the list to a stream, use `filter` and `mapToDouble`, and handle the optional result of `average()`.
22. Given a list of integers, use a combination of functional interfaces and streams to find the highest number that is divisible by 5.
    Approach: Define a functional interface for checking divisibility, use streams to filter, and find the maximum.
23. Create a method that takes a list of strings and returns a new list with all strings that have a length greater than 5, using the Stream API.
    Approach: Convert the list to a stream, filter based on length, and collect the results.
24. Use Optional and Lambda expressions to provide a default greeting message if a name is not provided.
    Approach: Use `Optional.ofNullable(name).orElse("Guest")` to handle the name and provide a default message.
25. Given a list of integers, use streams to calculate the product of all odd numbers.
    Approach: Convert the list to a stream, filter odd numbers, and use `reduce` to calculate the product.

Exception Handling

1. **Complex Arithmetic Operations**
    - **Description:** Create a program that performs various arithmetic operations (e.g., addition, subtraction, multiplication, division). Handle exceptions for invalid input and division by zero.
    - **Approach:** Use `try-catch` blocks to handle `ArithmeticException` for division by zero and `NumberFormatException` for invalid input.
2. **File Reading and Writing**
    - **Description:** Write a program that reads from one file and writes to another. Handle exceptions related to file operations such as file not found and IO errors.
    - **Approach:** Use `try-catch` blocks to handle `FileNotFoundException` and `IOException`. Ensure proper resource management using `try-with-resources`.
3. **Data Parsing and Validation**
    - **Description:** Parse data from a string (e.g., converting a comma-separated string into integers) and validate the data. Handle exceptions for parsing errors and invalid data formats.
    - **Approach:** Use `try-catch` blocks to handle `NumberFormatException` and implement custom validation exceptions if necessary.
4. **Nested Method Calls**

- **Description:** Create a program with multiple methods that throw exceptions. Demonstrate how exceptions are propagated through nested method calls and how to handle them at the top level.
- **Approach:** Throw exceptions from nested methods and handle them in the main method or calling method using `try-catch` blocks.

5. **Custom Exception Handling**
   - **Description:** Implement a custom exception class and use it to handle specific error conditions in your application. For example, create a `CustomDataException` for invalid data.
   - **Approach:** Define a custom exception class extending `Exception` or `RuntimeException`. Use this exception in your code and handle it with `try-catch`.
6. **Exception Handling in Collections**
   - **Description:** Write a program that manipulates a collection (e.g., `List` or `Map`) and handles exceptions related to collection operations such as out-of-bounds or null keys.
   - **Approach:** Use `try-catch` blocks to handle exceptions like `IndexOutOfBoundsException` and `NullPointerException` when accessing or modifying the collection.
7. **Exception Handling in Multiple Threads**
   - **Description:** Create a multi-threaded application where each thread performs a task that may throw exceptions. Ensure that exceptions are properly handled and reported for each thread.
   - **Approach:** Use `try-catch` blocks within each thread's `run` method. Aggregate and handle exceptions in the main thread or a central location.
8. **Exception Handling in Recursive Methods**
   - **Description:** Implement a recursive method (e.g., calculating factorial or Fibonacci numbers) and handle exceptions that may occur during recursion.
   - **Approach:** Use `try-catch` blocks to handle potential exceptions such as `StackOverflowError` or invalid input in the recursive method.
9. **Simulating a Banking System**
   - **Description:** Create a simple banking system that handles operations like deposits and withdrawals. Handle exceptions for invalid transactions (e.g., withdrawing more than the balance).
   - **Approach:** Use `try-catch` blocks to handle exceptions such as `IllegalArgumentException` for invalid transactions and custom exceptions for specific banking errors.
10. **Exception Handling in JSON Parsing**
    - **Description:** Parse JSON data from a string or file. Handle exceptions related to JSON parsing errors and invalid JSON format.

- **Approach:** Use `try-catch` blocks to handle `JsonParseException` or `JsonMappingException` when parsing JSON data using a library like Jackson or Gson.

11. **Handling Null References**
    - **Description:** Develop a program that performs operations on objects and handles exceptions related to null references.
    - **Approach:** Use `try-catch` blocks to handle `NullPointerException` when working with potentially null objects or performing operations that assume non-null values.
12. **Exception Handling in Resource Allocation**
    - **Description:** Allocate resources such as network connections or file handles and handle exceptions related to resource allocation and management.
    - **Approach:** Use `try-catch` blocks to handle exceptions such as `IOException` and ensure resources are properly released using `finally` or `try-with-resources`.
13. **Exception Handling in Thread Execution**
    - **Description:** Create a multi-threaded application where each thread performs a task that may throw exceptions (e.g., dividing by zero). Ensure that exceptions are handled properly in each thread.
    - **Approach:** Implement a `Runnable` or `Callable` class where the `run` or `call` method includes `try-catch` blocks for exception handling. Use a `Thread` or `ExecutorService` to run threads and handle exceptions in the main thread or a central location.
14. **Handling InterruptedException in Threads**
    - **Description:** Write a program where threads perform tasks that might be interrupted. Handle `InterruptedException` and ensure proper thread interruption handling.
    - **Approach:** Use `try-catch` blocks to handle `InterruptedException` in `sleep`, `wait`, or other interruptible operations. Ensure that the thread's interrupted status is set correctly.
15. **Synchronous Operations with Exception Handling**
    - **Description:** Implement a system that performs synchronous operations with possible exceptions (e.g., processing multiple files). Ensure proper handling of exceptions during each operation.
    - **Approach:** Use `try-catch` blocks around synchronous operations. Implement exception handling for various scenarios like file reading or data processing.
16. **Thread Pool Exception Handling**
    - **Description:** Use a thread pool to execute multiple tasks. Handle exceptions that occur in tasks and ensure that they are properly reported or logged.
    - **Approach:** Submit tasks to an `ExecutorService` and handle exceptions using `Future.get()` with `try-catch`. Optionally, use a custom `ThreadPoolExecutor` to log or handle exceptions.

17. **Exception Handling in Producer-Consumer Model**
    - **Description:** Implement a producer-consumer model using threads where producers and consumers may throw exceptions (e.g., during data production or consumption). Handle these exceptions gracefully.
    - **Approach:** Use `try-catch` blocks within producer and consumer threads. Handle exceptions that may occur during data production or consumption and ensure proper synchronization.
18. **Exception Handling in Asynchronous Callbacks**
    - **Description:** Implement asynchronous callbacks using `CompletableFuture` or similar APIs. Handle exceptions that occur during asynchronous operations and callbacks.
    - **Approach:** Use `CompletableFuture.exceptionally` or `handle` methods to manage exceptions in asynchronous tasks and callbacks.
19. **Handling Resource Leaks in Multi-threaded Applications**
    - **Description:** Ensure that resources (e.g., file handles, network connections) are properly managed and closed in a multi-threaded application. Handle exceptions related to resource management.
    - **Approach:** Use `try-with-resources` within threads to manage resources and handle exceptions. Ensure that resources are closed even if exceptions occur.
20. **Exception Handling in Fork/Join Framework**
    - **Description:** Use the Fork/Join framework to parallelize tasks. Handle exceptions that occur during the execution of tasks and manage the results or exceptions.
    - **Approach:** Implement `RecursiveTask` or `RecursiveAction` and handle exceptions in the `compute` method. Manage results and exceptions in the main thread or combining results.
21. **Handling Exceptions in Parallel Streams**
    - **Description:** Use parallel streams to process data concurrently. Handle exceptions that occur during the processing of elements in parallel streams.
    - **Approach:** Use `try-catch` blocks within the stream operations. Handle exceptions using stream operators like `flatMap` and manage them appropriately.
22. **Exception Handling in Thread Pools with Timeout**
    - **Description:** Use thread pools with tasks that have timeouts. Handle exceptions related to timeout and task execution.
    - **Approach:** Submit tasks to a thread pool with a timeout. Use `Future.get(long timeout, TimeUnit unit)` and handle `TimeoutException` and other exceptions.

1. **File Reading and Writing with IO**
    - **Description:** Create a program that reads from one file and writes the content to another using `FileInputStream` and `FileOutputStream`. Handle exceptions related to file operations.
    - **Approach:** Use `try-catch` blocks to handle `FileNotFoundException` and `IOException`. Ensure proper resource management using `try-with-resources`.
2. **Handling IOException in Buffered Streams**
    - **Description:** Implement a program that uses `BufferedReader` and `BufferedWriter` for file operations. Handle exceptions such as file not found and IO errors.
    - **Approach:** Use `try-catch` blocks for `IOException` and `FileNotFoundException`. Implement `try-with-resources` to manage the buffered streams.
3. **Handling in NIO File Operations**
    - **Description:** Use Java NIO's `Files` and `Paths` classes to read from and write to files. Handle exceptions related to file operations.
    - **Approach:** Use `try-catch` blocks to handle `IOException` and `NoSuchFileException`. Handle file operations using `Files.readAllLines()` and `Files.write()`.
4. **Handling Directory Traversal with NIO**
    - **Description:** Implement a program to traverse directories and list files using NIO's `DirectoryStream`. Handle exceptions related to directory access and iteration.
    - **Approach:** Use `try-catch` blocks to handle `IOException` and `DirectoryIteratorException`. Implement directory traversal with `DirectoryStream`.
5. **Exception Handling in Asynchronous File Channels**
    - **Description:** Use `AsynchronousFileChannel` to perform asynchronous file operations. Handle exceptions related to asynchronous IO operations.
    - **Approach:** Use `Future` or `CompletionHandler` to manage asynchronous file operations. Handle exceptions such as `IOException` and `CompletionException`.
6. **Exception Handling in NIO Buffer Operations**
    - **Description:** Work with NIO `ByteBuffer` for reading and writing data. Handle exceptions related to buffer operations and ensure proper buffer management.
    - **Approach:** Use `try-catch` blocks to handle exceptions like `BufferOverflowException` and `BufferUnderflowException` during buffer operations.
7. **Exception Handling in Network Communication**

- **Description:** Implement network communication using `Socket` and `ServerSocket` for client-server interactions. Handle network-related exceptions.
- **Approach:** Use `try-catch` blocks to handle `IOException`, `SocketException`, and `UnknownHostException`. Manage network communication and exceptions properly.

8. **Handling File Locking with NIO**
    - **Description:** Use NIO's `FileChannel` to lock and unlock files. Handle exceptions related to file locking operations.
    - **Approach:** Use `try-catch` blocks to handle `IOException` and `OverlappingFileLockException` when working with file locks.
9. **Exception Handling in Random Access Files**
    - **Description:** Create a program using `RandomAccessFile` to read and write data at specific file positions. Handle exceptions related to file access and positioning.
    - **Approach:** Use `try-catch` blocks to handle `IOException` and `EOFException`. Ensure proper file access and handling.
10. **Handling Path Operations with NIO**
    - **Description:** Perform various path operations using NIO's `Path` and `Paths` classes (e.g., copying, moving, deleting files). Handle exceptions related to path operations.
    - **Approach:** Use `try-catch` blocks to handle `IOException`, `NoSuchFileException`, and `FileAlreadyExistsException` during path operations.
11. **Exception Handling in File I/O with Non-Blocking NIO**
    - **Description:** Implement non-blocking file operations using NIO's `FileChannel`. Handle exceptions related to non-blocking I/O.
    - **Approach:** Use `try-catch` blocks to handle exceptions such as `IOException` and `ClosedChannelException` during non-blocking file operations.
12. **Handling Large Files with IO and NIO**
    - **Description:** Create a program to handle large files efficiently using both IO and NIO techniques. Handle exceptions related to large file processing.
    - **Approach:** Use `try-catch` blocks to manage `IOException`, `FileNotFoundException`, and `OutOfMemoryError`. Implement efficient reading and writing strategies.
13. **Exception Handling in File Attribute Operations**
    - **Description:** Use NIO's `Files` class to read and write file attributes (e.g., size, last modified time). Handle exceptions related to file attributes.
    - **Approach:** Use `try-catch` blocks to handle `IOException` and `UnsupportedOperationException` when accessing or modifying file attributes.
14. **Exception Handling in Char Streams**

- **Description:** Implement a program using character-based streams (`FileReader`, `FileWriter`) for file operations. Handle exceptions related to character stream operations.
- **Approach:** Use `try-catch` blocks to handle `IOException` and `FileNotFoundException`. Ensure proper handling and management of character streams.

15. Handling Buffer Overflows with NIO Channels

- **Description:** Use NIO channels to read and write data with buffers. Handle exceptions related to buffer overflows and other channel operations.
- **Approach:** Use `try-catch` blocks to handle `BufferOverflowException`, `BufferUnderflowException`, and `IOException` during channel operations.