Java Assignment

Day 18:

Task 1: Creating and Managing Threads

Write a program that starts two threads, where each thread prints numbers from 1 to 10 with a 1-second delay between each number

```
public class NumberPrinter extends Thread {
    private String threadName;

    public NumberPrinter(String name) {
        threadName = name;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println(threadName + ": " + i);
            try {
                Thread.sleep(1000); // Delay for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
    }

    public static void main(String[] args) {
        NumberPrinter thread1 = new NumberPrinter("Thread 1");
        NumberPrinter thread2 = new NumberPrinter("Thread 2");

        thread1.start();
        thread2.start();
    }
}
```

Task 2: States and Transitions

Create a Java class that simulates a thread going through different lifecycle states: NEW, RUNNABLE, WAITING, TIMED_WAITING, BLOCKED, and TERMINATED. Use methods like sleep(), wait(), notify(), and join() to demonstrate these states..

```
public class ThreadLifecycle {
    public static void main(String[] args) {
        Thread thread = new Thread(() -> {
            System.out.println("Thread is in NEW state");

            System.out.println("Thread is in RUNNABLE state");
```

```
Thread.sleep(1000);



System.out.println("Thread is in WAITING state");
synchronized (ThreadLifecycle.class) {
   try {
      ThreadLifecycle.class.wait();
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}



System.out.println("Thread is in TIMED_WAITING state");
Thread.sleep(1000);



System.out.println("Thread is in BLOCKED state");
synchronized (ThreadLifecycle.class) {
   try {
      ThreadLifecycle.class.notify();
   } catch (IllegalMonitorStateException e) {
      e.printStackTrace();
   }
```

```
            }


            System.out.println("Thread is in TERMINATED state");
        });

        thread.start();



        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Task 3: Synchronization and Inter-thread Communication

Implement a producer-consumer problem using wait() and notify() methods to handle the correct processing sequence between threads.

```
public class ProducerConsumer {
    private int[] buffer = new int[5];
    private int count = 0;
```

```
private int index = 0;

public void produce(int value) {
   synchronized (this) {
      while (count == buffer.length) {
         try {
            wait();
         } catch (InterruptedException e) {
            e.printStackTrace();
         }
      }
      buffer[index] = value;
      index = (index + 1) % buffer.length;
      count++;
      notify();
   }
}

public int consume() {
   synchronized (this) {
      while (count == 0) {
         try {
            wait();
         } catch (InterruptedException e) {
```

```
                e.printStackTrace();
            }
        }
        int value = buffer[0];
        buffer[0] = buffer[count - 1];
        index = (index - 1 + buffer.length) % buffer.length;
        count--;
        notify();
        return value;
    }
  }

  public static void main(String[] args) {
    ProducerConsumer producerConsumer = new ProducerConsumer();

    Thread producer = new Thread(() -> {
      for (int i = 0; i < 10; i++) {
        producerConsumer.produce(i);
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          e.printStackTrace();
        }
```

```
            }
        });

        Thread consumer = new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                System.out.println("Consumed: " +
producerConsumer.consume());
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        producer.start();
        consumer.start();
    }
}
```

Task 4: Synchronized Blocks and Methods

Write a program that simulates a bank account being accessed by multiple threads to perform deposits and withdrawals using synchronized methods to prevent race conditions.

```
public class BankAccount {
    private int balance = 0;

    public synchronized void deposit(int amount) {
        balance += amount;
        System.out.println("Deposited: " + amount);
        System.out.println("Balance: " + balance);
    }

    public synchronized void withdraw(int amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println("Withdrawn: " + amount);
            System.out.println("Balance: " + balance);
        } else {
            System.out.println("Insufficient balance");
        }
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Thread thread1 = new Thread(() -> {
            account.deposit(100);
```

```
            account.withdraw(50);
        });

        Thread thread2 = new Thread(() -> {
            account.deposit(200);
            account.withdraw(100);
        });

        thread1.start();
        thread2.start();
    }
}
```

Task 5: Thread Pools and Concurrency Utilities

Create a fixed-size thread pool and submit multiple tasks that perform complex calculations or I/O operations and observe the execution.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExample {
    public static void main(String[] args) {
```

```
ExecutorService executorService = Executors.newFixedThreadPool(3);

for (int i = 0; i < 5; i++) {
    int taskNumber = i;
    executorService.submit(() -> {

        System.out.println("Task " + taskNumber + " started");
        try {
            Thread.sleep(2000); // Simulate a 2-second operation
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Task " + taskNumber + " completed");
    });
}

executorService.shutdown();
try {
    executorService.awaitTermination(5, TimeUnit.MINUTES);
} catch (InterruptedException e) {
    e.printStackTrace();
```

```
        }

        System.out.println("All tasks completed");
    }
}
```

Task 6: Executors, Concurrent Collections, CompletableFuture

Use an ExecutorService to parallelize a task that calculates prime numbers up to a given number and then use CompletableFuture to write the results to a file asynchronously.

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.atomic.AtomicInteger;

public class PrimeNumberCalculator {
    public static void main(String[] args) {
        int maxNumber = 1000000;
        ExecutorService executorService = Executors.newFixedThreadPool(4);

        AtomicInteger primeCount = new AtomicInteger(0);
```

```
        CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
            for (int i = 2; i <= maxNumber; i++) {
                if (isPrime(i)) {
                    primeCount.incrementAndGet();
                }
            }
        }, executorService);

        future.thenAccept(result -> {
            System.out.println("Prime numbers found: " + primeCount.get());
            writeResultsToFile(primeCount.get());
        });

        executorService.shutdown();
    }

    private static boolean isPrime(int number) {
        if (number <= 1) {
            return false;
        }
        for (int i = 2; i * i <= number; i++) {
            if (number % i == 0) {
```

```
            return false;
        }
    }
    return true;
}

private static void writeResultsToFile(int primeCount) {
    try {

        CompletableFuture.runAsync(() -> {
            try {

                System.out.println("Writing results to file...");
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

Task 7: Writing Thread-Safe Code, Immutable Objects

Design a thread-safe Counter class with increment and decrement methods. Then demonstrate its usage from multiple threads. Also, implement and use an immutable class to share data between threads.

```
import java.util.concurrent.atomic.AtomicInteger;

public class ThreadSafeExample {

    public static class Counter {
        private AtomicInteger count;

        public Counter() {
            count = new AtomicInteger(0);
        }

        public void increment() {
            count.incrementAndGet();
        }

        public void decrement() {
            count.decrementAndGet();
        }

        public int getCount() {
```

```
        return count.get();
    }
}


public static class ImmutableData {
    private final int value;

    public ImmutableData(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}

public static void main(String[] args) {

    Counter counter = new Counter();

    Thread incrementThread = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
```

```
    }
});

Thread decrementThread = new Thread(() -> {
    for (int i = 0; i < 1000; i++) {
        counter.decrement();
    }
});

incrementThread.start();
decrementThread.start();

try {
    incrementThread.join();
    decrementThread.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

System.out.println("Final count: " + counter.getCount());


ImmutableData data = new ImmutableData(42);
System.out.println("Immutable data value: " + data.getValue());
```

```
    }
}
```

Day 19:

Task 1: Generics and Type Safety

Create a generic Pair class that holds two objects of different types, and write a method to return a reversed version of the pair.

```
public class Pair<T, U> {
    private T first;
    private U second;

    public Pair(T first, U second) {
        this.first = first;
        this.second = second;
    }

    public T getFirst() {
        return first;
    }

    public U getSecond() {
        return second;
```

```
    public Pair<U, T> reversed() {
        return new Pair<>(second, first);
    }
}

public class Main {
    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<>("Hello", 42);
        System.out.println("Original pair: (" + pair.getFirst() + ", " + pair.getSecond() + ")");

        Pair<Integer, String> reversedPair = pair.reversed();
        System.out.println("Reversed pair: (" + reversedPair.getFirst() + ", " + reversedPair.getSecond() + ")");
    }
}
```

Task 2: Generic Classes and Methods

Implement a generic method that swaps the positions of two elements in an array, regardless of their type, and demonstrate its usage with different object types.

```
public class GenericSwap {
    public static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
```

```
        array[i] = array[j];
        array[j] = temp;
    }

    public static void main(String[] args) {
        Integer[] integers = {1, 2, 3, 4, 5};
        swap(integers, 1, 3);
        System.out.println("Swapped integers: " +
Arrays.toString(integers));

        String[] strings = {"Hello", "World", "Java", "Programming"};
        swap(strings, 1, 3);
        System.out.println("Swapped strings: " +
Arrays.toString(strings));

        Double[] doubles = {1.0, 2.0, 3.0, 4.0, 5.0};
        swap(doubles, 1, 3);
        System.out.println("Swapped doubles: " +
Arrays.toString(doubles));
    }
}
```

Task 3: Reflection API

Use reflection to inspect a class's methods, fields, and constructors, and modify the access level of a private field, setting its value during runtime

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ReflectionExample {
    public static void main(String[] args) throws Exception {
        Class<?> clazz = MyClass.class;

        Method[] methods = clazz.getMethods();
        for (Method method : methods) {
            System.out.println("Method: " + method.getName());
        }


        Field[] fields = clazz.getDeclaredFields();
        for (Field field : fields) {
            System.out.println("Field: " + field.getName());
        }

        Constructor<?>[] constructors = clazz.getConstructors();
        for (Constructor<?> constructor : constructors) {
```

```
            System.out.println("Constructor: " + constructor.getName());
        }



        Field privateField = clazz.getDeclaredField("privateField");
        privateField.setAccessible(true);
        privateField.set(new MyClass(), "New Value");

        MyClass myObject = new MyClass();

        System.out.println("Modified private field value: " +
privateField.get(myObject));
    }
}

class MyClass {
    private String privateField = "Initial Value";

    public MyClass() {
    }

    public void publicMethod() {
    }
```

```
    private void privateMethod() {
    }
}
```

Task 4: Lambda Expressions

Implement a Comparator for a Person class using a lambda expression, and sort a list of Person objects by their age..

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class LambdaExample {
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Alice", 25));
        people.add(new Person("Bob", 30));
        people.add(new Person("Charlie", 20));
        people.add(new Person("David", 35));

        Comparator<Person> ageComparator = (p1, p2) -> Integer.compare(p1.getAge(), p2.getAge());
        Collections.sort(people, ageComparator);
```

```
        for (Person person : people) {
            System.out.println(person);
        }
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
```

```
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}
```

Task 5: Functional Interfaces

Create a method that accepts functions as parameters using Predicate, Function, Consumer, and Supplier interfaces to operate on a Person object.

```
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.function.Supplier;

public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        Person person = new Person("John", 30);



        Predicate<Person> isAdult = p -> p.getAge() >= 18;
        if (isAdult.test(person)) {
            System.out.println("Person is an adult");
        } else {
            System.out.println("Person is not an adult");
```

```
        }


        Function<Person, String> getName = p -> p.getName();
        String name = getName.apply(person);
        System.out.println("Name: " + name);

        Consumer<Person> printName = p -> System.out.println("Name:
" + p.getName());
        printName.accept(person);



        Supplier<Person> personSupplier = () -> new Person("Jane", 25);
        Person newPerson = personSupplier.get();
        System.out.println("New person: " + newPerson);
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + "}";
    }
}
```

Day 20:

Task 1: Java IO Basics

Write a program that reads a text file and counts the frequency of each word using FileReader and FileWriter.

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
```

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class WordFrequencyCounter {
    public static void main(String[] args) {
        File file = new File("input.txt");
        Map<String, Integer> wordFrequencyMap = new HashMap<>();

        try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
            String line;
            while ((line = reader.readLine()) != null) {
                String[] words = line.split("\\s+");
                for (String word : words) {
                    word = word.toLowerCase();
                    if (wordFrequencyMap.containsKey(word)) {
                        wordFrequencyMap.put(word, wordFrequencyMap.get(word) + 1);
                    } else {
                        wordFrequencyMap.put(word, 1);
                    }
                }
```

```
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }

        try (FileWriter writer = new FileWriter("output.txt")) {
            for (Map.Entry<String, Integer> entry : wordFrequencyMap.entrySet()) {
                writer.write(entry.getKey() + ": " + entry.getValue() + "\n");
            }
        } catch (IOException e) {
            System.out.println("Error writing file: " + e.getMessage());
        }
    }
}
```

Task 2: Serialization and Deserialization

Serialize a custom object to a file and then deserialize it back to recover the object state.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
```

```
import java.util.Map;


public class WordFrequencyCounter {
   public static void main(String[] args) {
      try {
         File file = new File("input.txt");
         Map<String, Integer> wordFrequencyMap = new HashMap<>();


         try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String line;
            while ((line = reader.readLine()) != null) {
               String[] words = line.split("\\s+");
               for (String word : words) {
                  word = word.toLowerCase();
                  if (wordFrequencyMap.containsKey(word)) {
                     wordFrequencyMap.put(word,
wordFrequencyMap.get(word) + 1);
                  } else {
                     wordFrequencyMap.put(word, 1);
                  }
               }
            }
         }
```

```
        try (FileWriter writer = new FileWriter("output.txt")) {
            for (Map.Entry<String, Integer> entry : wordFrequencyMap.entrySet()) {
                writer.write(entry.getKey() + ": " + entry.getValue() + "\n");
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Task 3: New IO (NIO)

Use NIO Channels and Buffers to read content from a file and write to another file.

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class NIOFileTransfer {
```

```
    public static void main(String[] args) {
        Path sourceFile = Paths.get("source.txt");
        Path destinationFile = Paths.get("destination.txt");

        try (FileChannel sourceChannel = FileChannel.open(sourceFile,
StandardOpenOption.READ);
             FileChannel destinationChannel =
FileChannel.open(destinationFile, StandardOpenOption.WRITE,
StandardOpenOption.CREATE,
StandardOpenOption.TRUNCATE_EXISTING)) {

            ByteBuffer buffer = ByteBuffer.allocateDirect(1024);



            while (sourceChannel.read(buffer) != -1) {
                buffer.flip(); // Prepare the buffer for writing
                destinationChannel.write(buffer);
                buffer.clear();
            }

            System.out.println("File transfer complete.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
}
```

Task 4: Java Networking

Write a simple HTTP client that connects to a URL, sends a request, and displays the response headers and body.

```
import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.HttpURLConnection;

import java.net.URL;


public class SimpleHTTPClient {


   public static void main(String[] args) {

      String urlString = "http://example.com"; // Replace with the URL you want to connect to


      try {

         URL url = new URL(urlString);


         HttpURLConnection connection = (HttpURLConnection) url.openConnection();


         connection.setRequestMethod("GET");
```

```
            int responseCode = connection.getResponseCode();
            System.out.println("Response Code: " + responseCode);

            System.out.println("Response Headers:");
            connection.getHeaderFields().forEach((key, value) -> {
                if (key != null) {
                    System.out.println(key + ": " + value);
                }
            });

            System.out.println("\nResponse Body:");
            BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
            reader.close();

            connection.disconnect();

        } catch (IOException e) {
            e.printStackTrace();
```

```
        }
    }
}
```

Task 5: Java Networking and Serialization

Develop a basic TCP client and server application where the client sends a serialized object with 2 numbers and operation to be performed on them to the server, and the server computes the result and sends it back to the client. for eg, we could send 2, 2, ""+"" which would mean 2 + 2

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class Server {

    public static void main(String[] args) {
        final int PORT = 5000;

        try {
            ServerSocket serverSocket = new ServerSocket(PORT);
            System.out.println("Server started. Waiting for client...");
```

```
        while (true) {
            Socket socket = serverSocket.accept();
            System.out.println("Client connected: " + socket);

            ObjectInputStream objectInputStream = new
ObjectInputStream(socket.getInputStream());
            ObjectOutputStream objectOutputStream = new
ObjectOutputStream(socket.getOutputStream());

            try {

                Object obj = objectInputStream.readObject();

                if (obj instanceof OperationRequest) {
                    OperationRequest request = (OperationRequest) obj;

                    double result = 0;
                    switch (request.getOperation()) {
                        case "+":
                            result = request.getNumber1() +
request.getNumber2();
                            break;
                        case "-":
```

```
                    result = request.getNumber1() -
request.getNumber2();
                    break;
                case "*":
                    result = request.getNumber1() *
request.getNumber2();
                    break;
                case "/":
                    result = request.getNumber1() /
request.getNumber2();
                    break;
                default:
                    System.out.println("Unsupported operation: " +
request.getOperation());
                    break;
                }

                objectOutputStream.writeObject(result);
            } else {
                System.out.println("Received object is not of type
OperationRequest.");
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
```

```
            objectInputStream.close();
            objectOutputStream.close();
            socket.close();
        }

    } catch (IOException e) {
        e.printStackTrace();
    }
  }
}
```

Task 6: Java 8 Date and Time API

Write a program that calculates the number of days between two dates input by the user.

```
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.time.temporal.ChronoUnit;
import java.util.Scanner;

public class DateDifferenceCalculator {

  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
```

```
        System.out.print("Enter the first date (yyyy-MM-dd): ");
        String firstDateString = scanner.nextLine();

        System.out.print("Enter the second date (yyyy-MM-dd): ");
        String secondDateString = scanner.nextLine();

        DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");

        try {
            LocalDate firstDate = LocalDate.parse(firstDateString, dateFormatter);
            LocalDate secondDate = LocalDate.parse(secondDateString, dateFormatter);

            long daysDifference = Math.abs(ChronoUnit.DAYS.between(firstDate, secondDate));

            System.out.println("Number of days between " + firstDate + " and " + secondDate + ": " + daysDifference);

        } catch (Exception e) {
```

```
            System.out.println("Invalid date format. Please enter dates in yyyy-MM-dd format.");
        } finally {
            scanner.close();
        }
    }
}
```

Task 7: Timezone

Create a timezone converter that takes a time in one timezone and converts it to another timezone.

```
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
import java.time.format.DateTimeFormatter;
import java.util.Scanner;

public class TimezoneConverter {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        try {
```

```
        System.out.print("Enter a date and time (yyyy-MM-dd HH:mm:ss): ");
        String inputDateTimeString = scanner.nextLine();

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        LocalDateTime localDateTime = LocalDateTime.parse(inputDateTimeString, formatter);

        System.out.print("Enter the source timezone (e.g., Europe/Paris): ");
        String sourceTimeZone = scanner.nextLine();

        System.out.print("Enter the target timezone (e.g., America/New_York): ");
        String targetTimeZone = scanner.nextLine();

        ZonedDateTime sourceZonedDateTime = localDateTime.atZone(ZoneId.of(sourceTimeZone));
```

```
        ZonedDateTime targetZonedDateTime = sourceZonedDateTime.withZoneSameInstant(ZoneId.of(targetTimeZone));

        String formattedResult = targetZonedDateTime.format(formatter);

        System.out.println("Converted time in " + targetTimeZone + ": " + formattedResult);

    } catch (Exception e) {
        System.out.println("Invalid input. Please enter dates and time in yyyy-MM-dd HH:mm:ss format.");
    } finally {
        scanner.close();
    }
  }
}
```

Day 21:

Task 1: Establishing Database Connections

Write a Java program that connects to a SQLite database and prints out the connection object to confirm successful connection.

```
import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;


public class SQLiteConnectionExample {


   public static void main(String[] args) {

      Connection connection = null;


      try {

         String url = "jdbc:sqlite:/path/to/your/database.db"; // Replace with your SQLite database path


         connection = DriverManager.getConnection(url);


         System.out.println("Connection to SQLite database has been established.");


         System.out.println("Connection object: " + connection);


      } catch (SQLException e) {

         System.err.println(e.getMessage());
```

```
        } finally {
            try {
                if (connection != null) {
                    connection.close();
                }
            } catch (SQLException e) {
                System.err.println(e.getMessage());
            }
        }
    }
}
```

Task 2: SQL Queries using JDBC

Create a table 'User' with a following schema 'User ID' and 'Password' stored as hash format (note you have research on how to generate hash from a string), accept ""User ID"" and ""Password"" as input and check in the table if they match to confirm whether user access is allowed or not.

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.sql.*;

public class UserAuthentication {
```

```
    private static final String url = "jdbc:sqlite:users.db";

    public static void main(String[] args) {
        try (Connection conn = DriverManager.getConnection(url)) {
            if (conn != null) {
                DatabaseMetaData meta = conn.getMetaData();
                System.out.println("Connected to SQLite database.");

                String createTableSQL = "CREATE TABLE IF NOT EXISTS User
("
                        + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                        + "userId TEXT UNIQUE NOT NULL,"
                        + "passwordHash TEXT NOT NULL)";
                try (Statement stmt = conn.createStatement()) {
                    stmt.execute(createTableSQL);
                    System.out.println("User table created or already exists.");
                }

                insertSampleUsers(conn);

                String userId = "user1";
                String password = "password1";
                boolean loginSuccessful = verifyLogin(conn, userId,
password);
```

```
            if (loginSuccessful) {
                System.out.println("Login successful for user: " + userId);
            } else {
                System.out.println("Invalid credentials for user: " + userId);
            }

        } else {
            System.out.println("Failed to connect to SQLite database.");
        }
    } catch (SQLException e) {
        System.err.println("SQL Exception: " + e.getMessage());
    }
}

private static void insertSampleUsers(Connection conn) throws SQLException {
    String insertSQL = "INSERT INTO User (userId, passwordHash) VALUES (?, ?)";
    try (PreparedStatement pstmt = conn.prepareStatement(insertSQL)) {
        String userId1 = "user1";
        String password1 = hashPassword("password1");
        pstmt.setString(1, userId1);
        pstmt.setString(2, password1);
        pstmt.executeUpdate();
```

```
        String userId2 = "user2";
        String password2 = hashPassword("password2");
        pstmt.setString(1, userId2);
        pstmt.setString(2, password2);
        pstmt.executeUpdate();

        System.out.println("Sample users inserted into 'User' table.");
    }
}

private static boolean verifyLogin(Connection conn, String userId, String password) throws SQLException {
    String selectSQL = "SELECT * FROM User WHERE userId = ? AND passwordHash = ?";
    try (PreparedStatement pstmt = conn.prepareStatement(selectSQL)) {
        pstmt.setString(1, userId);
        pstmt.setString(2, hashPassword(password));
        ResultSet resultSet = pstmt.executeQuery();
        return resultSet.next();
    }
}
```

```
    private static String hashPassword(String password) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(password.getBytes());
            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                String hex = Integer.toHexString(0xff & b);
                if (hex.length() == 1) hexString.append('0');
                hexString.append(hex);
            }
            return hexString.toString();
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

Task 3: PreparedStatement

Modify the SELECT query program to use PreparedStatement to parameterize the query and prevent SQL injection.

```
import java.security.MessageDigest;
```

```
import java.security.NoSuchAlgorithmException;
import java.sql.*;

public class UserAuthentication {

   private static final String url = "jdbc:sqlite:users.db";

   public static void main(String[] args) {

      try (Connection conn = DriverManager.getConnection(url)) {
         if (conn != null) {
            DatabaseMetaData meta = conn.getMetaData();
            System.out.println("Connected to SQLite database.");



            String createTableSQL = "CREATE TABLE IF NOT EXISTS User
("
                  + "id INTEGER PRIMARY KEY AUTOINCREMENT,"
                  + "userId TEXT UNIQUE NOT NULL,"
                  + "passwordHash TEXT NOT NULL)";
            try (Statement stmt = conn.createStatement()) {
               stmt.execute(createTableSQL);
               System.out.println("User table created or already exists.");
            }
```

```
            insertSampleUsers(conn);
            String userId = "user1";
            String password = "password1";
            boolean loginSuccessful = verifyLogin(conn, userId, password);
            if (loginSuccessful) {
                System.out.println("Login successful for user: " + userId);
            } else {
                System.out.println("Invalid credentials for user: " + userId);
            }

        } else {
            System.out.println("Failed to connect to SQLite database.");
        }
    } catch (SQLException e) {
        System.err.println("SQL Exception: " + e.getMessage());
    }
}

private static void insertSampleUsers(Connection conn) throws SQLException {
    String insertSQL = "INSERT INTO User (userId, passwordHash) VALUES (?, ?)";
```

```
    try (PreparedStatement pstmt = conn.prepareStatement(insertSQL)) {
        String userId1 = "user1";
        String password1 = hashPassword("password1");
        pstmt.setString(1, userId1);
        pstmt.setString(2, password1);
        pstmt.executeUpdate();

        String userId2 = "user2";
        String password2 = hashPassword("password2");
        pstmt.setString(1, userId2);
        pstmt.setString(2, password2);
        pstmt.executeUpdate();

        System.out.println("Sample users inserted into 'User' table.");
    }
  }


  private static boolean verifyLogin(Connection conn, String userId, String password) throws SQLException {
    String selectSQL = "SELECT * FROM User WHERE userId = ? AND passwordHash = ?";
    try (PreparedStatement pstmt = conn.prepareStatement(selectSQL)) {
```

```
            pstmt.setString(1, userId);
            pstmt.setString(2, hashPassword(password));
            ResultSet resultSet = pstmt.executeQuery();
            return resultSet.next(); // True if a row with matching credentials exists
        }
    }
```

Day 22:

Task 1: Write a set of JUnit tests for a given class with simple mathematical operations (add, subtract, multiply, divide) using the basic @Test annotation.

MathematicalOperations.java :

```
public class MathematicalOperations {

    public int add(int a, int b) {
        return a + b;
    }

    public int subtract(int a, int b) {
        return a - b;
    }

    public int multiply(int a, int b) {
```

```
        return a * b;
    }

    public int divide(int a, int b) {
        if (b == 0) {
            throw new IllegalArgumentException("Cannot divide by zero");
        }
        return a / b;
    }
}
```

MathematicalOperationsTest.java :

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MathematicalOperationsTest {

    private MathematicalOperations mathOps = new MathematicalOperations();

    @Test
    public void testAdd() {
        assertEquals(5, mathOps.add(2, 3));
        assertEquals(-1, mathOps.add(2, -3));
        assertEquals(0, mathOps.add(0, 0));
```

```
}

@Test
public void testSubtract() {
   assertEquals(-1, mathOps.subtract(2, 3));
   assertEquals(5, mathOps.subtract(8, 3));
   assertEquals(0, mathOps.subtract(5, 5));
}

@Test
public void testMultiply() {
   assertEquals(6, mathOps.multiply(2, 3));
   assertEquals(-6, mathOps.multiply(2, -3));
   assertEquals(0, mathOps.multiply(0, 5));
}

@Test
public void testDivide() {
   assertEquals(2, mathOps.divide(6, 3));
   assertEquals(-2, mathOps.divide(6, -3));
   assertEquals(0, mathOps.divide(0, 5));

   try {
      mathOps.divide(6, 0);
```

```
            fail("Expected IllegalArgumentException for divide by zero");
        } catch (IllegalArgumentException e) {
            assertEquals("Cannot divide by zero", e.getMessage());
        }
    }
}
```

Task 2: Extend the above JUnit tests to use @Before, @After, @BeforeClass, and @AfterClass annotations to manage test setup and teardown.

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

public class MathematicalOperationsTest {

    private static MathematicalOperations mathOps;

    @BeforeClass
    public static void setUpClass() {
        System.out.println("Setting up class...");
```

```
    mathOps = new MathematicalOperations();
}

@AfterClass
public static void tearDownClass() {

    System.out.println("Tearing down class...");
    mathOps = null;
}

@Before
public void setUp() {
    System.out.println("Setting up for test...");
}

@After
public void tearDown() {
    System.out.println("Tearing down after test...");
}

@Test
public void testAdd() {
    assertEquals(5, mathOps.add(2, 3));
    assertEquals(-1, mathOps.add(2, -3));
```

```
    assertEquals(0, mathOps.add(0, 0));
}

@Test
public void testSubtract() {
    assertEquals(-1, mathOps.subtract(2, 3));
    assertEquals(5, mathOps.subtract(8, 3));
    assertEquals(0, mathOps.subtract(5, 5));
}

@Test
public void testMultiply() {
    assertEquals(6, mathOps.multiply(2, 3));
    assertEquals(-6, mathOps.multiply(2, -3));
    assertEquals(0, mathOps.multiply(0, 5));
}

@Test
public void testDivide() {
    assertEquals(2, mathOps.divide(6, 3));
    assertEquals(-2, mathOps.divide(6, -3));
    assertEquals(0, mathOps.divide(0, 5));

    try {
```

```
            mathOps.divide(6, 0);
            fail("Expected IllegalArgumentException for divide by zero");
        } catch (IllegalArgumentException e) {
            assertEquals("Cannot divide by zero", e.getMessage());
        }
    }
}
```

Task 3: Create test cases with assertEquals, assertTrue, and assertFalse to validate the correctness of a custom String utility class.

```
import org.junit.Test;
import static org.junit.Assert.*;

public class StringUtilsTest {

    @Test
    public void testIsEmpty() {
        assertTrue(StringUtils.isEmpty(null));
        assertTrue(StringUtils.isEmpty(""));
        assertTrue(StringUtils.isEmpty("   "));
        assertFalse(StringUtils.isEmpty("hello"));
    }

    @Test
```

```
    public void testReverse() {
        assertNull(StringUtils.reverse(null));
        assertEquals("", StringUtils.reverse(""));
        assertEquals("olleh", StringUtils.reverse("hello"));
        assertEquals("racecaR", StringUtils.reverse("Racecar"));
    }

    @Test
    public void testStartsWithIgnoreCase() {
        assertTrue(StringUtils.startsWithIgnoreCase("hello world",
"HELLO"));
        assertTrue(StringUtils.startsWithIgnoreCase("Hello world",
"helLo"));
        assertFalse(StringUtils.startsWithIgnoreCase("Hello world",
"world"));
        assertFalse(StringUtils.startsWithIgnoreCase(null, "hello"));
        assertFalse(StringUtils.startsWithIgnoreCase("hello", null));
        assertFalse(StringUtils.startsWithIgnoreCase(null, null));
    }
}
public class StringUtils {

    public static boolean isEmpty(String str) {
        return str == null || str.trim().isEmpty();
    }
```

```
    public static String reverse(String str) {
        if (str == null) {
            return null;
        }
        return new StringBuilder(str).reverse().toString();
    }

    public static boolean startsWithIgnoreCase(String str, String prefix)
{
        if (str == null || prefix == null) {
            return false;
        }
        if (prefix.length() > str.length()) {
            return false;
        }
        return str.substring(0, prefix.length()).equalsIgnoreCase(prefix);
    }
}
```

Task 4: Research and present a comparison of different garbage collection algorithms (Serial, Parallel, CMS, G1, ZGC) in Java.

Garbage collection (GC) algorithms in Java play a crucial role in managing memory efficiently by reclaiming unused objects, thus

preventing memory leaks and improving overall application performance.

1. Serial Garbage Collector

Description: The Serial GC is a simple, single-threaded collector suited for small-scale applications or environments with limited resources.

Usage: Typically used for client-side applications or environments where low memory footprint and simplicity are prioritized.

Performance: Pause times can be noticeable as it operates using a single thread for both garbage collection and application execution.

2. Parallel Garbage Collector

Description: The Parallel GC (also known as throughput collector) uses multiple threads to perform garbage collection, which reduces pause times by parallelizing garbage collection tasks.

Usage: Suitable for applications that prioritize throughput and can tolerate short pauses for garbage collection.

Performance: Provides improved throughput by leveraging multiple threads, but may introduce longer pause times during full GC cycles.

3. Concurrent Mark-Sweep (CMS) Garbage Collector

Description: CMS GC minimizes pause times by performing most of its work concurrently with the application threads.

Usage: Suitable for applications that prioritize responsiveness and low pause times, as it significantly reduces pause times compared to the Parallel GC.

Performance: While it reduces pause times, CMS GC can suffer from fragmentation issues and may not perform well with very large heap sizes.

4. G1 (Garbage First) Garbage Collector

Description: G1 GC is designed to provide both low pause times and high throughput by dividing the heap into regions and performing garbage collection on the regions with the least live data first (hence "Garbage First").

Usage: Suitable for large heap applications where responsiveness and predictable pause times are important.

Performance: Generally offers better performance compared to CMS for large heap sizes, as it can avoid the fragmentation issues of CMS.

5. ZGC (Z Garbage Collector)

Description: ZGC is a low-latency garbage collector designed to keep pause times consistently low even with very large heap sizes (multi-terabyte heaps).

Usage: Ideal for applications requiring very low pause times (in the millisecond range) regardless of heap size.

Performance: Offers excellent performance in terms of low pause times but may have slightly lower throughput compared to other collectors like G1.

Day 23:

Task 1: Singleton

Implement a Singleton class that manages database connections. Ensure the class adheres strictly to the singleton pattern principles.

```
public class DatabaseSingleton {


    private static DatabaseSingleton instance;
```

```
private DatabaseSingleton() {
    System.out.println("Initializing database connection...");
}

public static DatabaseSingleton getInstance() {
    if (instance == null) {
        instance = new DatabaseSingleton();
    }
    return instance;
}

public void executeQuery(String query) {
    System.out.println("Executing query: " + query);
}



public void closeConnection() {
    System.out.println("Closing database connection...");
}

public static void main(String[] args) {
```

```
        DatabaseSingleton dbSingleton = DatabaseSingleton.getInstance();


        dbSingleton.executeQuery("SELECT * FROM users");
        dbSingleton.executeQuery("INSERT INTO products VALUES (...)");

        dbSingleton.closeConnection();
    }
}
```

Task 2: Factory Method

Create a ShapeFactory class that encapsulates the object creation logic of different Shape objects like Circle, Square, and Rectangle."

```
public class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType == null) {
            return null;
        }
        if (shapeType.equalsIgnoreCase("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equalsIgnoreCase("SQUARE")) {
            return new Square();
```

```
      } else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
         return new Rectangle();
      }
      return null;
   }
}
public interface Shape {
   void draw();
}
public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}

public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}

public class Rectangle implements Shape {
```

```
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

Task 3: Proxy

Create a proxy class for accessing a sensitive object that contains a secret key. The proxy should only allow access to the secret key if a correct password is provided.

```
interface SensitiveObject {
    void accessSecretKey();
}

class RealSensitiveObject implements SensitiveObject {
    private String secretKey = "mySuperSecretKey123";

    @Override
    public void accessSecretKey() {
        System.out.println("Accessing secret key: " + secretKey);
    }
}

class SensitiveObjectProxy implements SensitiveObject {
```

```
    private RealSensitiveObject realObject = new RealSensitiveObject();
    private String password;

    public SensitiveObjectProxy(String password) {
        this.password = password;
    }

    @Override
    public void accessSecretKey() {
        if (authenticate()) {
            realObject.accessSecretKey();
        } else {
            System.out.println("Access denied! Incorrect password.");
        }
    }

    private boolean authenticate() {
        return password.equals("correctPassword");
    }
}

public class Main {
    public static void main(String[] args) {
```

```
        SensitiveObject proxy = new SensitiveObjectProxy("correctPassword");

        proxy.accessSecretKey();

        SensitiveObject proxyWrong = new SensitiveObjectProxy("wrongPassword");
        proxyWrong.accessSecretKey();
    }
}
```

Task 4: Strategy

Develop a Context class that can use different SortingStrategy algorithms interchangeably to sort a collection of numbers

```
interface SortingStrategy {
    void sort(int[] numbers);
}

class BubbleSortStrategy implements SortingStrategy {
    @Override
    public void sort(int[] numbers) {
        int n = numbers.length;
        for (int i = 0; i < n-1; i++) {
            for (int j = 0; j < n-i-1; j++) {
```

```
            if (numbers[j] > numbers[j+1]) {
                int temp = numbers[j];
                numbers[j] = numbers[j+1];
                numbers[j+1] = temp;
            }
        }
    }
    System.out.println("Sorting using Bubble Sort");
  }
}
class SelectionSortStrategy implements SortingStrategy {
  @Override
  public void sort(int[] numbers) {
    int n = numbers.length;
    for (int i = 0; i < n-1; i++) {
      int minIndex = i;
      for (int j = i+1; j < n; j++) {
        if (numbers[j] < numbers[minIndex]) {
          minIndex = j;
        }
      }
      int temp = numbers[minIndex];
      numbers[minIndex] = numbers[i];
      numbers[i] = temp;
```

```
        }
        System.out.println("Sorting using Selection Sort");
    }
}

class Context {
    private SortingStrategy sortingStrategy;

    public Context(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void setSortingStrategy(SortingStrategy sortingStrategy) {
        this.sortingStrategy = sortingStrategy;
    }

    public void performSort(int[] numbers) {
        sortingStrategy.sort(numbers);
    }
}

public class Main {
    public static void main(String[] args) {
        int[] numbers = {5, 1, 4, 2, 8};
```

```
        SortingStrategy bubbleSort = new BubbleSortStrategy();
        Context context = new Context(bubbleSort);
        context.performSort(numbers.clone()); // Clone to preserve original array

        SortingStrategy selectionSort = new SelectionSortStrategy();
        context.setSortingStrategy(selectionSort);
        context.performSort(numbers.clone()); // Clone to preserve original array
    }
}
```

Day 24:.

Task 1: Build Lifecycle

Demonstrate the use of Maven lifecycle phases (clean, compile, test, package, install, deploy) by executing them on a sample project and documenting what happens in each phase.

```
package com.example;

public class App {
    public static void main(String[] args) {
        System.out.println("Hello, Maven!");
    }
```

}

Maven Lifecycle Phases:

Clean: mvn clean

Compile: mvn compile

Test: mvn test

Package: mvn package

Install: mvn install

Deploy: mvn deploy