Core Java 8:

- Concurrent Pattern in Java
- Concurrent Collections
- Lambda Expressions
- Stream API
- Introduction to Design Pattern
- GitHub
- Introduction to JDBC:
    - Connection, Statement,
    - PreparedS tatement,
    - ResultSet.

Q1:

1. **Explain Lambda Expression with a syntax and differentiate between Lambda Expression and over Regular Methods. Write JAVA Lambda Expression program using one parameter.**

Java Lambda Expressions

Lambda Expressions were added in Java 8.

A lambda expression is a short block of code which takes in parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and they can be implemented right in the body of a method.

Expressions are limited. They have to immediately return a value, and they cannot contain variables, assignments or statements such as if or for. In order to do more complex operations, a code block can be used with curly braces. If the lambda expression needs to return a value, then the code block should have a return statement.

Syntax:

The simplest lambda expression contains a single parameter and an expression:

parameter* -> *expression

To use more than one parameter, wrap them in parentheses:

(parameter1, parameter2) -> expression

Use a lambda expression in the ArrayList's forEach() method to print every item in the list:

```
import java.util.ArrayList;

public class p1 {
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    numbers.forEach( (n) -> { System.out.println(n); } );
  }
}
```

Lambda expressions can be stored in variables if the variable's type is an interface which has only one method. The lambda expression should have the same number of parameters and the same return type as that method. Java has many of these kinds of interfaces built in, such as the Consumer interface (found in the java.util package) used by lists.

```
import java.util.ArrayList;
import java.util.function.Consumer;

public class  p2{
  public static void main(String[] args) {
    ArrayList<Integer> numbers = new ArrayList<Integer>();
    numbers.add(5);
    numbers.add(9);
    numbers.add(8);
    numbers.add(1);
    Consumer<Integer> method = (n) -> { System.out.println(n); };
    numbers.forEach( method );
  }
}
```

To use a lambda expression in a method, the method should have a parameter with a single-method interface as its type. Calling the interface's method will run the lambda expression:

```
interface StringFunction {
    String run(String str);
  }

  public class p3 {
    public static void main(String[] args) {
      StringFunction exclaim = (s) -> s + "!";
      StringFunction ask = (s) -> s + "?";
      printFormatted("Hello", exclaim);
      printFormatted("Hello", ask);
```

```
    }
    public static void printFormatted(String str, StringFunction format) {
        String result = format.run(str);
        System.out.println(result);
    }
}
```

Lambda Expression in Java

Lambda expressions were introduced in Java 8 to provide a concise way to represent anonymous functions — functions without a name that can be passed around as arguments to methods or stored in variables. They are particularly useful in functional programming paradigms.

Syntax of Lambda Expression

The syntax of a lambda expression in Java has the following parts:

(parameters) -> expression
or
(parameters) -> { statements; }

Here's a breakdown:

- **Parameters**: These are similar to method parameters but without explicit types. If there's only one parameter, you can omit the parentheses. For example, x -> ... instead of (x) -> ....
- **Arrow (->)**: This separates parameters from the body of the lambda expression.
- **Expression/Body**: This is the code that implements the function. If it's a single expression, you can omit the braces {}. If it's a block of statements, you must use braces {}.

Differences Between Lambda Expressions and Regular Methods

1. **Syntax**: Lambda expressions are more concise and do not require a method name, return type declaration, and sometimes even parameter types (if they can be inferred).
2. **Purpose**: Lambda expressions are primarily used for implementing functional interfaces (interfaces with a single abstract method), while regular methods can have any access modifier, be static or instance methods, and can belong to any class.
3. **Flexibility**: Lambda expressions are more flexible in terms of where they can be used (like as arguments to methods or stored in variables), while regular methods are more rigidly defined in terms of where and how they can be called.

Example Program: Lambda Expression with One Parameter

```
// Functional interface with one abstract method
interface MyFunctionalInterface {
    void sayMessage(String message);
}

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

```
        // Lambda expression to implement MyFunctionalInterface
        MyFunctionalInterface myLambda = (message) -> System.out.println("Hello, " + message);

        // Calling the lambda expression
        myLambda.sayMessage("Lambda Expression!");
    }
}
```

Example Program: Lambda exp With Multiple Parameters

```
// Functional interface with multiple parameters
interface MathOperation {
    int operate(int a, int b);
}

public class LambdaWithMultipleParameters {
    public static void main(String[] args) {
        // Lambda expression to implement MathOperation interface for addition
        MathOperation addition = (a, b) -> a + b;

        // Lambda expression to implement MathOperation interface for subtraction
        MathOperation subtraction = (a, b) -> a - b;

        // Lambda expression to implement MathOperation interface for multiplication
        MathOperation multiplication = (a, b) -> a * b;

        // Using the lambda expressions
        System.out.println("Addition: " + addition.operate(10, 5)); // Output: 15
        System.out.println("Subtraction: " + subtraction.operate(10, 5)); // Output: 5
        System.out.println("Multiplication: " + multiplication.operate(10, 5)); // Output: 50
    }
}
```

Explanation of the Example

- **MyFunctionalInterface**: This is a functional interface with a single abstract method sayMessage(String message).
- **Lambda Expression**: (message) -> System.out.println("Hello, " + message) is the lambda expression that implements the sayMessage method of MyFunctionalInterface. Here, message is the parameter passed to the lambda expression.
- **Main Method**: In the main method, we create an instance myLambda of MyFunctionalInterface using the lambda expression. Then, we invoke myLambda.sayMessage("Lambda Expression!"), which prints Hello, Lambda Expression!.

Design and implement a simple JDBC application program:

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

Register the Driver class

Create connection

Create statement

Execute queries

Close connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class JDBCTest {
            Class.forName("com.mysql.jdbc.Driver");
    // JDBC URL, username, and password of MySQL server
    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/testdb";
    private static final String JDBC_USER = "your_username";
    private static final String JDBC_PASSWORD = "your_password";

    // JDBC variables for opening and managing connection
    private static Connection connection;
    private static PreparedStatement preparedStatement;
    private static ResultSet resultSet;

    public static void main(String[] args) {
        try {
            // Connect to the MySQL database
            connection = DriverManager.getConnection(JDBC_URL, JDBC_USER, JDBC_PASSWORD);

            // Insert a new record into the employees table
            insertEmployee("John Doe", 30, "IT");

            // Print all records from the employees table
            printEmployees();

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Close JDBC objects in finally block to ensure they're closed even if an exception is thrown
            try {
```

```
            if (resultSet != null) {
                resultSet.close();
            }
            if (preparedStatement != null) {
                preparedStatement.close();
            }
            if (connection != null) {
                connection.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

private static void insertEmployee(String name, int age, String department) throws SQLException {
    // SQL query to insert a new record into the employees table
    String sql = "INSERT INTO employees (name, age, department) VALUES (?, ?, ?)";
    preparedStatement = connection.prepareStatement(sql);

    // Set parameters for the preparedStatement
    preparedStatement.setString(1, name);
    preparedStatement.setInt(2, age);
    preparedStatement.setString(3, department);

    // Execute the query
    int rowsInserted = preparedStatement.executeUpdate();
    if (rowsInserted > 0) {
        System.out.println("A new employee was inserted successfully!");
    }
}

private static void printEmployees() throws SQLException {
    // SQL query to select all records from the employees table
    String sql = "SELECT id, name, age, department FROM employees";
    preparedStatement = connection.prepareStatement(sql);

    // Execute the query and get the ResultSet
    resultSet = preparedStatement.executeQuery();

    // Iterate through the ResultSet and print each employee's information
    while (resultSet.next()) {
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        int age = resultSet.getInt("age");
        String department = resultSet.getString("department");

        System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age + ", Department: " + department);
    }
}
}
```

Explain the following with a program example: i. Statement ii. PreparedStatement

1. Statement Interface

The Statement interface in JDBC is used to execute a static SQL statement that is sent to the database. It doesn't allow parameters to be passed dynamically into the SQL query. Here's how you typically use Statement:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class StatementExample {
    // JDBC URL, username, and password of MySQL server
    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/testdb";
    private static final String JDBC_USER = "your_username";
    private static final String JDBC_PASSWORD = "your_password";

    public static void main(String[] args) {
        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try {
            // Connect to the MySQL database
            connection = DriverManager.getConnection(JDBC_URL, JDBC_USER, JDBC_PASSWORD);

            // Create a statement
            statement = connection.createStatement();

            // Execute a query
            String sql = "SELECT * FROM employees";
            resultSet = statement.executeQuery(sql);

            // Process the ResultSet
```

```
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String name = resultSet.getString("name");
                int age = resultSet.getInt("age");
                String department = resultSet.getString("department");

                System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age + ", Department: " + department);
            }

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

2. PreparedStatement Interface

The PreparedStatement interface extends Statement and provides more flexibility and performance improvements, especially when executing the same SQL statement multiple times with different parameters. It allows you to dynamically pass parameters into the SQL query. Here's how you typically use PreparedStatement:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class PreparedStatementExample {
    // JDBC URL, username, and password of MySQL server
    private static final String JDBC_URL = "jdbc:mysql://localhost:3306/testdb";
    private static final String JDBC_USER = "your_username";
    private static final String JDBC_PASSWORD = "your_password";

    public static void main(String[] args) {
        Connection connection = null;
        PreparedStatement preparedStatement = null;
        ResultSet resultSet = null;

        try {
            // Connect to the MySQL database
            connection = DriverManager.getConnection(JDBC_URL, JDBC_USER, JDBC_PASSWORD);

            // SQL query with parameters
            String sql = "SELECT * FROM employees WHERE department = ?";

            // Create a PreparedStatement with parameterized query
            preparedStatement = connection.prepareStatement(sql);

            // Set parameters dynamically
            preparedStatement.setString(1, "IT");
```

```
        // Execute the query
        resultSet = preparedStatement.executeQuery();

        // Process the ResultSet
        while (resultSet.next()) {
            int id = resultSet.getInt("id");
            String name = resultSet.getString("name");
            int age = resultSet.getInt("age");
            String department = resultSet.getString("department");

            System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age + ", Department: " + department);
        }

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

Define Streams. List out the difference between stream and collections with examples.

Definition of Streams:

In Java, streams represent a sequence of elements from a source that supports functional-style operations. Streams allow you to process collections of data in a declarative and functional way, without explicitly writing loops and iterations. They operate on the source data structure (like collections, arrays, or I/O channels) to produce a result.

Key Characteristics of Streams:

1. **Sequence of Elements**: Streams represent a sequence of elements that can be processed sequentially.
2. **Functional Operations**: Streams support functional-style operations such as filter, map, reduce, collect, etc., which can be chained together to form a pipeline of operations.
3. **Laziness**: Intermediate operations in a stream are lazily evaluated, meaning they are executed only when necessary (when a terminal operation is invoked).
4. **Internal Iteration**: Streams use internal iteration, where the iteration logic is handled by the Stream API itself, as opposed to external iteration (e.g., using for loops).

Differences Between Streams and Collections:

1. **Nature of Data Handling:**

- **Collections**: Store and manage elements directly. They allow for direct access, insertion, deletion, and modification of elements.
- **Streams**: Do not store elements themselves; they operate on a source (like a collection or array) and process elements from that source.

2. Mutability:

- **Collections**: Mutable data structures where elements can be added, removed, or modified directly.
- **Streams**: Typically immutable once created; operations on streams do not modify the original source but produce a new stream or result.

3. Execution:

- **Collections**: Operations are executed eagerly. For example, when iterating through a collection with a loop, each element is processed immediately.
- **Streams**: Operations are executed lazily. Intermediate operations (like filter or map) are deferred until a terminal operation (like forEach or collect) is invoked. This lazy evaluation allows for optimizations like short-circuiting.

4. Processing Paradigm:

- **Collections**: Emphasize external iteration, where you explicitly control the iteration process (e.g., using for or while loops).
- **Streams**: Emphasize internal iteration, where the Stream API handles the iteration process and you define operations on the elements.

Example:

Example 1: Using Collections

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

public class CollectionsExampleforDiffinstrdcol{
    public static void main(String[] args) {
        // Creating a list of names using ArrayList
        List<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
```

```
        names.add("Charlie");

        // Direct access and modification
        System.out.println("Accessing element directly: " + names.get(1)); // Bob

        // Iterating through the collection
        System.out.println("Iterating through the collection:");
        for (String name : names) {
            System.out.println(name);
        }

        // Adding a new element
        names.add("David");
        System.out.println("After adding a new element:");
        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

Example Using Streams:

```
import java.util.Arrays;
import java.util.List;

public class StreamsExample {
    public static void main(String[] args) {
        // Creating a list of names using Arrays.asList
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

        // Using streams to process elements
        System.out.println("Using streams to process elements:");
        names.stream()
                .filter(name -> name.startsWith("A")) // Intermediate operation
(filtering)
                .forEach(System.out::println); // Terminal operation (printing)

        // Attempting to modify the stream (not possible, streams are immutable)
        names.stream().map(String::toUpperCase).forEach(System.out::println); //
Uncommenting this line will cause an exception

        // Adding a new element (not possible directly with streams)
        names.add("David"); // Uncommenting this line will cause an
UnsupportedOperationException
    }
}
```

Q2:

1. Explain how concurrency problems are overwhelmed in Collections with example.

Concurrency problems in collections arise when multiple threads access and modify a shared collection concurrently. These problems include issues like data inconsistency, race conditions, and even data corruption if not handled properly. Java provides several strategies and classes to address these concurrency challenges effectively.

Techniques to Overcome Concurrency Problems in Collections:

1. **Synchronization**: Use synchronized blocks or methods to ensure that only one thread can access the collection at a time.
2. **Concurrent Collections**: Java provides thread-safe collections in the java.util.concurrent package that handle synchronization internally for better performance in concurrent environments.
3. **Locks**: Explicitly use locks (ReentrantLock or ReadWriteLock) to control access to critical sections of code where collections are accessed or modified.
4. **Atomic Operations**: Use atomic classes (AtomicInteger, AtomicReference, etc.) for operations that need to be performed atomically without interference from other threads.

Example Using Synchronized Collection:

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

public class SynchronizedCollectionExample {
```

```
    public static void main(String[] args) {
        // Creating a synchronized list
        List<String> synchronizedList = Collections.synchronizedList(new ArrayList<>());

        // Creating and starting multiple threads to modify the synchronized list
concurrently
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                synchronizedList.add(Thread.currentThread().getName() + "-" + i);}};

        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);

        thread1.start();
        thread2.start();

        // Waiting for threads to complete
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();}

        // Printing the size of the synchronized list
        System.out.println("Size of synchronized list: " + synchronizedList.size());}}
```

Explanation of Synchronized Collection Example:

- **Synchronized List**: Collections.synchronizedList(new ArrayList<>()) creates a synchronized wrapper around an ArrayList, ensuring that all operations on the list are thread-safe.
- **Synchronization**: Inside the Runnable task, the synchronized block ensures that only one thread at a time can modify the list. This prevents concurrent modification exceptions and ensures data integrity.
- **Thread Safety**: By synchronizing access to the list, we prevent multiple threads from accessing or modifying the list simultaneously, avoiding concurrency issues.

Example Using Concurrent Collection:

```
import java.util.concurrent.CopyOnWriteArrayList;

public class ConcurrentCollectionExample {
    public static void main(String[] args) {
        // Creating a concurrent list
        CopyOnWriteArrayList<String> concurrentList = new CopyOnWriteArrayList<>();
```

```
        // Creating and starting multiple threads to modify the concurrent list
concurrently
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                concurrentList.add(Thread.currentThread().getName() + "-" + i);}};

        Thread thread1 = new Thread(task);
        Thread thread2 = new Thread(task);

        thread1.start();
        thread2.start();

        // Waiting for threads to complete
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();}

        // Printing the size of the concurrent list
        System.out.println("Size of concurrent list: " + concurrentList.size());}}
```

Explanation of Concurrent Collection Example:

- **CopyOnWriteArrayList**: CopyOnWriteArrayList is a concurrent collection that allows safe concurrent access for reads and writes. It achieves thread-safety by creating a new copy of the underlying array whenever an element is added, modified, or removed.
- **Concurrency Handling**: Threads thread1 and thread2 concurrently add elements to the CopyOnWriteArrayList. The collection internally handles synchronization, ensuring thread-safety without explicit synchronization on the client side.
- **Advantages**: CopyOnWriteArrayList is suitable for scenarios where reads are more frequent than writes, as it provides efficient read operations and maintains consistency during concurrent modifications.

2.Write note on Design Patterns in Java

A design patterns are **well-proved solution** for solving the specific problem/task.

Core Java Design Patterns

In core java, there are mainly three types of design patterns, which are further divided into their sub-parts:

1.Creational Design Pattern

1. Factory Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern.

2. Structural Design Pattern

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

3. Behavioural Design Pattern

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

1. Creational Design Patterns

1. Factory Pattern

- Purpose: Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

- Example: java.util.Calendar.getInstance() uses the Factory pattern to create a calendar instance based on locale.

2. Abstract Factory Pattern

- Purpose: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Example: javax.xml.parsers.DocumentBuilderFactory provides abstract factory implementation to create XML document builders.

3. Singleton Pattern

- Purpose: Ensures a class has only one instance and provides a global point of access to that instance.
- Example: java.lang.Runtime.getRuntime() returns the singleton instance of Runtime class.

4. Prototype Pattern

- Purpose: Creates new objects by copying an existing instance, thereby avoiding the need to use a constructor.
- Example: java.lang.Object.clone() method uses prototype pattern to create a copy of an object.

5. Builder Pattern

- Purpose: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.
- Example: java.lang.StringBuilder uses the builder pattern to construct strings efficiently.

2. Structural Design Patterns

1. Adapter Pattern

- Purpose: Allows objects with incompatible interfaces to work together by providing a wrapper that converts one interface to another.
- Example: java.util.Arrays.asList() adapts an array to a List interface.

2. Bridge Pattern

- Purpose: Decouples an abstraction from its implementation so that both can vary independently.
- Example: GUI frameworks often use the bridge pattern to separate platform-specific code from generic GUI classes.

3. Composite Pattern

- Purpose: Allows clients to treat individual objects and compositions of objects uniformly.
- Example: java.awt.Container uses composite pattern to treat components (buttons, panels) uniformly.

4. Decorator Pattern

- Purpose: Attaches additional responsibilities to an object dynamically. Provides a flexible alternative to subclassing for extending functionality.
- Example: java.io.BufferedInputStream decorates an input stream with buffering capabilities.

5. Facade Pattern

- Purpose: Provides a simplified interface to a complex subsystem of classes, making it easier to use.
- Example: javax.faces.context.FacesContext provides a facade for accessing various aspects of web application.

6. Flyweight Pattern

- Purpose: Minimizes memory usage by sharing common data among multiple objects.
- Example: java.lang.Integer.valueOf() caches and reuses Integer objects within a certain range.

7. Proxy Pattern

- Purpose: Provides a surrogate or placeholder for another object to control access to it.
- Example: java.lang.reflect.Proxy allows creation of dynamic proxy instances for interfaces at runtime.

3. Behavioural Design Patterns

1. Chain of Responsibility Pattern

- Purpose: Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Example: Java Servlet filters use chain of responsibility pattern for processing HTTP requests.

2. Command Pattern

- Purpose: Encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queuing, or logging of requests.

- Example: java.lang.Runnable interface uses command pattern for encapsulating actions that can be executed.

3. Interpreter Pattern

- Purpose: Defines a grammar for interpreting certain expressions and provides a way to evaluate those expressions.
- Example: java.util.Pattern uses interpreter pattern for regular expression matching.

4. Iterator Pattern

- Purpose: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Example: java.util.Iterator interface provides iterator pattern for traversing collections.

5. Mediator Pattern

- Purpose: Defines an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly.
- Example: GUI event handling often uses mediator pattern to decouple UI components.

6. Memento Pattern

- Purpose: Captures and externalizes an object's internal state so that the object can be restored to this state later.
- Example: java.io.Serializable interface uses memento pattern to serialize and deserialize objects.

7. Observer Pattern

- Purpose: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Example: java.util.Observer interface and java.util.Observable class implement observer pattern for event handling.

8. State Pattern

- Purpose: Allows an object to alter its behavior when its internal state changes. The object appears to change its class.
- Example: java.awt.Component uses state pattern for managing component state (enabled, disabled, etc.).

9. Strategy Pattern

- Purpose: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Allows algorithms to vary independently from clients using them.
- Example: java.util.Comparator interface uses strategy pattern for defining custom sorting strategies.

10. Template Pattern

- Purpose: Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Allows subclasses to redefine certain steps of an algorithm without changing its structure.
- Example: java.io.InputStream and java.io.OutputStream use template pattern for reading from and writing to streams.

11. Visitor Pattern

- Purpose: Defines a new operation to a collection of objects without changing the objects themselves. Encapsulates the logic for operation in a separate visitor object.
- Example: javax.lang.model.element.ElementVisitor uses visitor pattern for visiting elements in a Java source file.

Define Streams. List out the different Stream API Operations and explain with an example.

Streams in Java

Streams in Java represent a sequence of elements and provide a way to operate on them sequentially or in parallel. They allow for functional-style operations on collections of elements, facilitating concise and readable code.

Characteristics of Streams:

- **Sequence of Elements**: Streams provide an interface to a sequence of elements.
- **Aggregate Operations**: Operations such as filter, map, reduce, etc., can be performed on streams.
- **Laziness**: Intermediate operations are lazy, meaning they do not process elements until a terminal operation is invoked.
- **Possibility of Parallelism**: Streams can leverage parallelism for faster execution on multicore processors.

Stream API Operations

The Stream API provides two types of operations:

1. **Intermediate Operations**: These operations are lazy and return a new stream. They are typically used for filtering, transforming, or manipulating the elements of the stream.
2. **Terminal Operations**: These operations are eager and produce a result or a side-effect. They trigger the processing of elements in the stream pipeline.

Intermediate operations:

- filter: Keeps only words longer than 5 characters.
- map: Converts each word to uppercase.
- distinct: Removes duplicate words.
- sorted: Sorts the words alphabetically.
- limit: Takes only the first two words.
- collect: Collects the processed words into a list.

Terminal operations:

- forEach: Prints each word in the filtered list.
- findFirst: Finds the first word starting with "b" and prints it if present.
- mapToInt and sum: Calculates the total length of all words.

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamOperationsExample {

    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "grape", "orange",
"peach", "banana", "apple");

        // Intermediate Operations

        // 1. filter: Keeps only words longer than 5 characters.
        List<String> filteredWords = words.stream()
                                          .filter(word -> word.length() > 5)
                                          .collect(Collectors.toList());
        System.out.println("Filtered words (length > 5): " + filteredWords);

        // 2. map: Converts each word to uppercase.
        List<String> uppercaseWords = words.stream()
```

```
                                        .map(String::toUpperCase)
                                        .collect(Collectors.toList());
        System.out.println("Uppercase words: " + uppercaseWords);

        // 3. distinct: Removes duplicate words.
        List<String> distinctWords = words.stream()
                                        .distinct()
                                        .collect(Collectors.toList());
        System.out.println("Distinct words: " + distinctWords);

        // 4. sorted: Sorts the words alphabetically.
        List<String> sortedWords = words.stream()
                                    .sorted()
                                    .collect(Collectors.toList());
        System.out.println("Sorted words: " + sortedWords);

        // 5. limit: Takes only the first two words.
        List<String> limitedWords = words.stream()
                                        .limit(2)
                                        .collect(Collectors.toList());
        System.out.println("Limited words (first two): " + limitedWords);

        // Terminal Operations

        // 6. collect: Collects the processed words into a list.
        List<String> collectedWords = words.stream()
                                        .filter(word -> word.length() > 5)
                                        .map(String::toUpperCase)
                                        .distinct()
                                        .sorted()
                                        .limit(2)
                                        .collect(Collectors.toList());
        System.out.println("Collected words: " + collectedWords);

        // 7. forEach: Prints each word in the filtered list.
        words.stream()
             .filter(word -> word.length() > 5)
             .forEach(System.out::println);

        // 8. findFirst: Finds the first word starting with "b" and prints it if
present.
        String firstBWord = words.stream()
                                 .filter(word -> word.startsWith("b"))
                                 .findFirst()
                                 .orElse("No word found starting with 'b'");
        System.out.println("First word starting with 'b': " + firstBWord);

        // 9. mapToInt and sum: Calculates the total length of all words.
        int totalLength = words.stream()
                               .mapToInt(String::length)
                               .sum();
        System.out.println("Total length of all words: " + totalLength);
    }
```

```
}
```

Note on GitHub

GitHub is a web-based platform used for version control, collaboration, and project management. It is built around the Git version control system, allowing developers to track and manage changes to their codebase efficiently. Here are some key aspects and features of GitHub:

Key Features

1. **Repositories**:
    - A repository (or repo) is a central location where all files and their revision history are stored. It can be either public or private.
2. **Branches**:
    - Branching allows you to diverge from the main codebase to develop features, fix bugs, or experiment. The main branch is typically called main or master.
3. **Commits**:
    - A commit is a snapshot of changes made to the codebase. Each commit has a unique identifier and includes a message describing the changes.
4. **Pull Requests**:
    - Pull requests (PRs) are a way to propose changes to the codebase. They allow team members to review and discuss changes before merging them into the main branch.
5. **Issues**:
    - Issues are used to track tasks, enhancements, and bugs. They provide a way to discuss and manage work within a repository.
6. **Actions**:
    - GitHub Actions enable CI/CD workflows. You can automate tasks such as testing, building, and deploying code with custom scripts.
7. **Wiki**:
    - Each repository can have a wiki to document the project. It's useful for maintaining documentation, guides, and tutorials.
8. **Projects**:
    - Projects are used for project management. They provide Kanban-style boards for tracking progress on issues, pull requests, and other tasks.
9. **Forking**:
    - Forking creates a personal copy of someone else's repository. This is useful for contributing to open-source projects or creating separate versions of a project.
10. **Collaboration**:

- GitHub provides features for team collaboration, such as code review, commenting, and @mention notifications. It also integrates with various tools and services to enhance productivity.

Common GitHub Workflow

1. **Cloning a Repository**:
   - Clone a repository to your local machine using git clone [URL].
2. **Creating a Branch**:
   - Create a new branch for your work: git checkout -b feature-branch.
3. **Making Changes**:
   - Make changes to the codebase and commit them: git add . and git commit -m "Commit message".
4. **Pushing Changes**:
   - Push your changes to GitHub: git push origin feature-branch.
5. **Opening a Pull Request**:
   - On GitHub, open a pull request to propose merging your changes into the main branch.
6. **Reviewing and Merging**:
   - Team members review the pull request, provide feedback, and eventually merge the changes.
7. **Syncing with Upstream**:
   - Keep your branch up-to-date with the main branch: git fetch upstream and git rebase upstream/main.

Best Practices

1. **Write Clear Commit Messages**:
   - Use descriptive commit messages to explain what changes were made and why.
2. **Use Branches**:
   - Use branches for different features, bug fixes, or experiments to keep the main branch stable.
3. **Review Code**:
   - Perform code reviews to ensure code quality, share knowledge, and catch potential issues early.
4. **Automate Testing**:
   - Use GitHub Actions or other CI/CD tools to automate testing and deployment processes.
5. **Document Your Code**:
   - Maintain clear documentation in the README file and use comments to explain complex code sections.

6. **Follow Community Guidelines**:
    - Adhere to community guidelines and code of conduct when contributing to open-source projects.