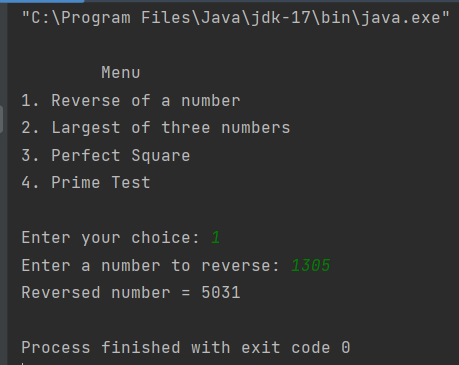
**Mukesh SA  
CB.EN.U4CYS21046**

**JAVA PROGRAMMING LAB  
ALGORITHMS**

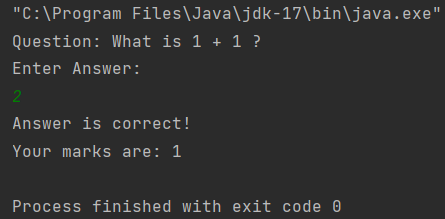
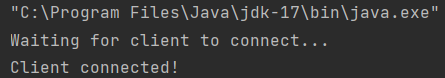
**P1:**1. Import the necessary libraries.  
2. Create a class named `Main` under the package `com.amrita.jpl.cys21046.P1`.  
3. Define the `main` function:  
 - Create a new `Scanner` object to read input from the user.  
 - Initialize the `choice` variable to an arbitrary value (1331 in this case).  
 - Display the menu options to the user.  
 - Prompt the user to enter their choice.  
 - Read the user's choice using `scanner.nextInt()`.  
4. Use a switch-case statement based on the user's choice:  
 - Case 1: Reverse of a number:  
 - Prompt the user to enter a number to reverse.  
 - Read the number using `scanner.nextInt()`.  
 - Call the `Functions.reverse\_num` method with the entered number as the argument.  
 - Print the reversed number.  
 - Break out of the switch-case.  
 - Case 2: Largest of three numbers:  
 - Prompt the user to enter three numbers separated by a space.  
 - Read the three numbers using `scanner.nextInt()`.  
 - Call the `Functions.large3num` method with the three numbers as arguments.  
 - Print the largest number.  
 - Break out of the switch-case.  
 - Case 3: Perfect Square:  
 - Prompt the user to enter a number to check for a perfect square.  
 - Read the number using `scanner.nextInt()`.  
 - Call the `Functions.perfect\_square\_check` method with the entered number as the argument.  
 - Print whether the number is a perfect square or not.  
 - Break out of the switch-case.  
 - Case 4: Prime Test:  
 - Prompt the user to enter a number to check for primality.  
 - Read the number using `scanner.nextInt()`.  
 - Call the `Functions.prime\_test` method with the entered number as the argument.  
 - Print whether the number is prime or not.  
 - Break out of the switch-case.  
 - Default case: Invalid choice:  
 - Print an error message for an invalid choice.  
 - Break out of the switch-case.  
5. Close the `Scanner` object using `scanner.close()`.  
6. Define the nested class `Functions` within the `Main` class:  
 - Define the `reverse\_num` method:  
 - Take an integer parameter `num`.  
 - Initialize a variable `reversedNum` to 0.  
 - Use a while loop to reverse the number:  
 - Extract the last digit of `num` using the modulo operator.  
 - Multiply `reversedNum` by 10 and add the extracted digit.  
 - Divide `num` by 10 to remove the last digit.  
 - Return the `reversedNum`.  
 - Define the `large3num` method:  
 - Take three integer parameters: `num1`, `num2`, and `num3`.  
 - Initialize a variable `largestNum` to `num1`.  
 - Use if statements to compare `num2` and `num3` with `largestNum` and update it if necessary.  
 - Return the `largestNum`.   
- Define the `perfect\_square\_check` method:  
 - Take an integer parameter `num`.  
 - Check if `num` is less than 0. If true, print an error message and return `false`.  
 - Calculate the square root of `num` using `Math.sqrt()` and cast it to an integer.  
 - Check if the square of the calculated square root is equal to `num`.  
 - Return `true` if the number is a perfect square, otherwise return `false`.  
- Define the `prime\_test` method:  
 - Take an integer parameter `num`.  
 - Check if `num` is less than 2. If true, print an error message and return `false`.  
 - Use a for loop to iterate from 2 to the square root of `num`.  
 - Check if `num` is divisible by any number in the range. If true, return `false`.  
 - If the loop completes without finding a divisor, return `true`.  
7. End of the program.

**Output:  
**

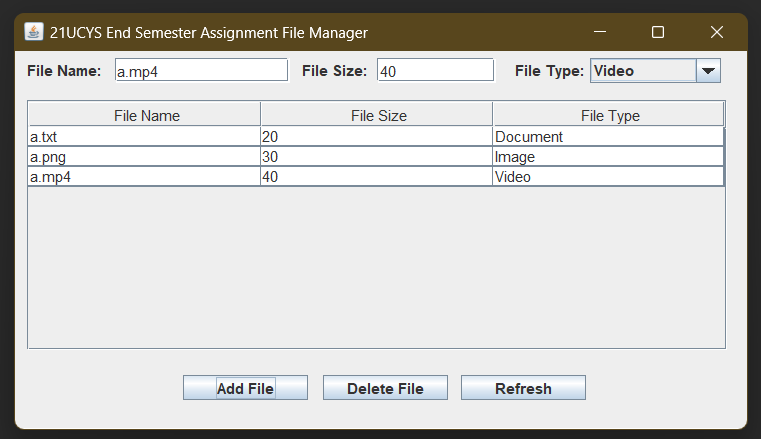
**P2:**  
1. Create an abstract class named QuizGame with the following methods:  
 - startGame(): Abstract method to start the quiz game.  
 - askQuestion(): Abstract method to ask a question during the quiz game.  
 - evaluateAnswer(): Abstract method to evaluate the answer provided during the quiz game.

2. Create an interface named QuizGameListener with the following methods:  
 - onQuestionAsked(question): Called when a question is asked in the quiz game.  
 - onAnswerEvaluated(isCorrect): Called when an answer is evaluated in the quiz game.

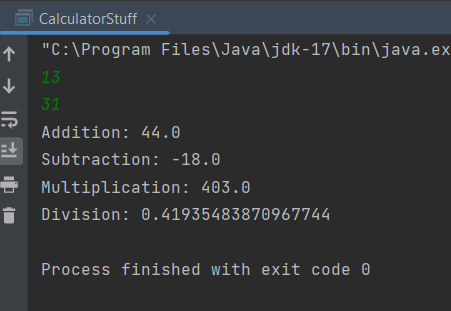
3. Create a class named QuizGameClient that extends QuizGame and implements QuizGameListener:  
 - Declare private variables outputStream and inputStream of type DataOutputStream and DataInputStream, respectively.  
 - Implement the startGame() method:  
 - Create a Socket object and connect to the server using the server's IP address and port.  
 - Initialize the outputStream and inputStream using the socket's getOutputStream() and getInputStream() methods.  
 - Read the first question from the inputStream using readUTF() method.  
 - Set the number of questions to n = 10.  
 - Run a loop until n becomes 0:  
 - Call the onQuestionAsked(question) method to notify the listener about the question asked.  
 - Call the evaluateAnswer() method to evaluate the answer provided by the user.  
 - Read the next question from the inputStream.  
 - Close the serverSocket, outputStream, and inputStream.  
 - Implement the evaluateAnswer() method:  
 - Create a Scanner object to read the user's answer from the console.  
 - Prompt the user to enter an answer.  
 - Read the answer from the scanner.  
 - Write the answer to the outputStream using the writeUTF() method.  
 - Read the evaluation result from the inputStream using the readBoolean() method.  
 - Call the onAnswerEvaluated(isCorrect) method to notify the listener about the evaluated answer.  
 - Implement the onQuestionAsked(question) method:  
 - Display the question to the user.  
 - Implement the onAnswerEvaluated(isCorrect) method:  
 - Display whether the answer is correct or incorrect.  
 - Read the marks from the inputStream using the readInt() method.  
 - Display the user's marks.  
 - Exit the program.  
4. Create a class named QuizGameServer that extends QuizGame and implements QuizGameListener:  
 - Declare private variables dops and dips of type DataOutputStream and DataInputStream, respectively.  
 - Declare a marks variable to store the user's marks.  
 - Implement the startGame() method:  
 - Create a ServerSocket object and listen for incoming connections on a specified port.  
 - Accept a client connection using the accept() method, which blocks until a client connects.  
 - Initialize the dops and dips using the clientSocket's getOutputStream() and getInputStream() methods.  
 - Set the number of questions to n = 10.  
 - Run a loop until n becomes 0:  
 - Call the askQuestion() method to ask a question.  
 - Call the evaluateAnswer() method to evaluate the client's answer.  
 - Close the serverSocket.  
 - Implement the askQuestion() method:  
 - Define a question as a string.  
 - Write the question to the client using the dops.writeUTF() method.  
 - Implement the evaluateAnswer() method:  
 - Read the answer from the client using the dips.readUTF() method.  
 - Compare the answer with the correct answer and set isCorrect accordingly.  
 - If the answer is correct, increment the marks variable.  
 - Write the evaluation result (isCorrect) to the client using the dops.writeBoolean() method.  
 - Write the marks to the client using the dops.writeInt() method.  
 - Implement the onQuestionAsked(question) method:  
 - Display the question to the server console.  
 - Implement the onAnswerEvaluated(isCorrect) method:  
 - Display whether the client's answer is correct or incorrect.  
 - Display the user's marks.  
5. Create a main class to start the Quiz Game server or client:  
 - Create an object of either QuizGameServer or QuizGameClient class.  
 - Call the startGame() method to start the quiz game.

**Output:**  
 

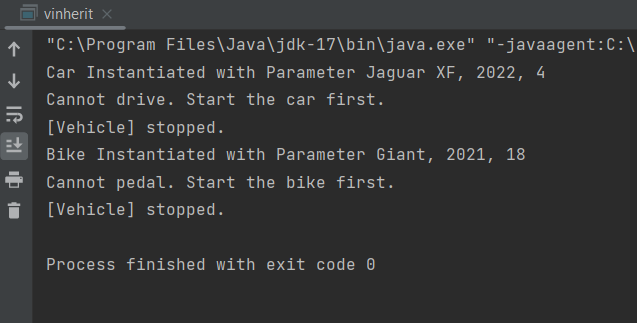
**EndSem:**1. Create a Java project and define the necessary packages.  
2. Create the `File` class:  
 - Define private variables `fileName` (String) and `fileSize` (int).  
 - Implement a constructor to initialize the `fileName` and `fileSize` variables.  
 - Implement getter and setter methods for `fileName` and `fileSize`.  
 - Implement a `displayFileDetails()` method to display the file name and file size.  
3. Create the `Document` class that extends the `File` class:  
 - Define a private variable `documentType` (String).  
 - Implement a constructor to initialize the `fileName`, `fileSize`, and `documentType` variables.  
 - Implement getter and setter methods for `documentType`.  
 - Override the `displayFileDetails()` method to display the document type in addition to the file details.  
4. Create the `Image` class that extends the `File` class:  
 - Define a private variable `resolution` (String).  
 - Implement a constructor to initialize the `fileName`, `fileSize`, and `resolution` variables.  
 - Implement getter and setter methods for `resolution`.  
 - Override the `displayFileDetails()` method to display the resolution in addition to the file details.  
5. Create the `Video` class that extends the `File` class:  
 - Define a private variable `duration` (String).  
 - Implement a constructor to initialize the `fileName`, `fileSize`, and `duration` variables.  
 - Implement getter and setter methods for `duration`.  
 - Override the `displayFileDetails()` method to display the duration in addition to the file details.  
6. Create the `FileManager` interface:  
 - Define the required methods: `addFile()`, `deleteFile()`, `displayAllFiles()`, `getAllFiles()`, `saveToFile()`, and `loadFromFile()`.  
7. Create the `FileManagerImpl` class that implements the `FileManager` interface:  
 - Define a private variable `files` (List of File objects) to store the files.  
 - Implement the methods defined in the `FileManager` interface:  
 - `addFile(File file)`: Add the file to the `files` list.  
 - `deleteFile(String fileName)`: Delete the file from the `files` list based on the file name.  
 - `displayAllFiles()`: Iterate through the `files` list and display the details of each file.  
 - `getAllFiles()`: Return the `files` list.  
 - `saveToFile(String fileName)`: Serialize and save the `files` list to a file.  
 - `loadFromFile(String fileName)`: Deserialize and load the `files` list from a file.  
8. Create the `FileManagementSystemUI` class:  
 - Define the necessary Swing components for the user interface (JFrame, JTextField, JComboBox, JTable, etc.).  
 - Implement methods to handle user interactions:  
 - `initialize()`: Set up the UI components and event listeners.  
 - `addFileButtonClicked()`: Retrieve the input from the UI components, create the appropriate file object (Document, Image, or Video), add the file to the file manager, and update the table.  
 - `deleteFileButtonClicked()`: Get the selected row from the table, retrieve the file name, delete the file from the file manager  
9. In the `FileManagementSystemUI` class, continue implementing the remaining methods:  
 - `displayAllFilesButtonClicked()`: Retrieve the files from the file manager, create a data model for the table, and display the file details in the table.  
 - `saveToFileButtonClicked()`: Retrieve the file name from the UI, call the `saveToFile()` method of the file manager to save the files to a file.  
 - `loadFromFileButtonClicked()`: Retrieve the file name from the UI, call the `loadFromFile()` method of the file manager to load the files from a file, update the table with the loaded files.  
10. Implement the serialization and deserialization methods in the `FileManagerImpl` class:  
 - `saveToFile(String fileName)`: Serialize the `files` list and save it to the specified file using ObjectOutputStream.  
 - `loadFromFile(String fileName)`: Deserialize the `files` list from the specified file using ObjectInputStream.  
11. Test the File Management System:  
 - Create an instance of the `FileManagementSystemUI` class.  
 - Initialize the UI using the `initialize()` method.  
 - Test adding files, deleting files, displaying all files, saving files to a file, and loading files from a file using the UI.  
12. Run the File Management System:  
 - Compile and run the Java program to launch the File Management System UI.  
 - Interact with the UI to perform file management operations and observe the results.



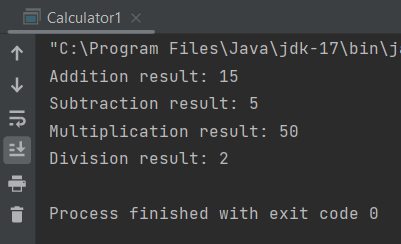
**Exercises:  
Basic Calculator:**1. Import the necessary classes: Import the `java.util.Scanner` class to read user input.  
2. Define the `Calculator` interface:  
 - Declare the following methods:  
 - `add(double a, double b)`: Adds two numbers and returns the sum.  
 - `subtract(double a, double b)`: Subtracts the second number from the first number and returns the difference.  
 - `multiply(double a, double b)`: Multiplies two numbers and returns the product.  
 - `divide(double a, double b)`: Divides the first number by the second number and returns the quotient or -1.0 if division by zero occurs.  
3. Define the `BasicCalculator` class that implements the `Calculator` interface:  
 - Implement the methods declared in the `Calculator` interface.  
 - In the `divide` method, check if the denominator (`b`) is not equal to zero. If it is not zero, perform the division and return the quotient. Otherwise, print a "Division by zero error!" message and return -1.0.  
4. Define the `CalculatorStuff` class:  
 - Define the `main` method, which is the entry point of the program.  
 - Create a `Scanner` object (`obj`) to read user input.  
 - Read two double numbers (`num1` and `num2`) from the user.  
 - Create an instance of the `BasicCalculator` class (`Obj1`).  
 - Perform the following operations using the `Obj1` instance:  
 - Add `num1` and `num2` and store the result in `result1`.  
 - Subtract `num2` from `num1` and store the result in `result2`.  
 - Multiply `num1` and `num2` and store the result in `result3`.  
 - Divide `num1` by `num2` and store the result in `result4`.  
 - Print the results of each operation.



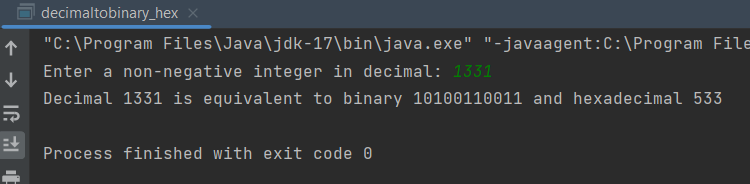
**Vinherit:**  
1. Define the `Vehicle` class:  
 - Declare an instance variable `run\_status` and initialize it to 0.  
 - Define a method `start` that sets `run\_status` to 1 and prints "[Vehicle] started."  
 - Define a method `stop` that sets `run\_status` to 0 and prints "[Vehicle] stopped."  
2. Define the `Car` class that extends the `Vehicle` class:  
 - Define a constructor that takes three parameters: `model\_name`, `Year`, and `no\_of\_wheels`.  
 - In the constructor, print "Car Instantiated with Parameter" followed by the values of the parameters.  
 - Define a method `drive` that takes a `gear\_position` parameter.  
 - If `run\_status` is 0, print "Cannot drive. Start the car first."  
 - Otherwise, print "Driving the car in gear position: " followed by the `gear\_position` value.  
3. Define the `Bike` class that extends the `Vehicle` class:  
 - Define a constructor that takes three parameters: `brand\_name`, `Year`, and `no\_of\_gears`.  
 - In the constructor, print "Bike Instantiated with Parameter" followed by the values of the parameters.  
 - Define a method `pedal` that takes a `pedal\_speed` parameter.  
 - If `run\_status` is 0, print "Cannot pedal. Start the bike first."  
 - Otherwise, print "Pedaling the bike at speed: " followed by the `pedal\_speed` value.  
4. Define the `vinherit` class:  
 - Define the `main` method, which is the entry point of the program.  
 - Create an instance of the `Car` class (`newCar`) with the parameters "Jaguar XF", 2022, and 4.  
 - Declare an integer variable `uu` and assign it the value 3.  
 - Call the `drive` method on `newCar` with `uu` as the argument.  
 - Call the `stop` method on `newCar`.  
 - Create an instance of the `Bike` class (`newBike`) with the parameters "Giant", 2021, and 18.  
 - Declare an integer variable `u` and assign it the value 10.  
 - Call the `pedal` method on `newBike` with `u` as the argument.  
 - Call the `stop` method on `newBike`.



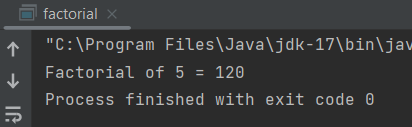
**Abstract Calculator:**1. Start the main method.  
2. Declare and initialize num1 and num2 with the desired values.  
3. Create instances of the calculator operations:  
 - Create an instance of Addition class, passing num1 and num2 as arguments.  
 - Create an instance of Subtraction class, passing num1 and num2 as arguments.  
 - Create an instance of Multiplication class, passing num1 and num2 as arguments.  
 - Create an instance of Division class, passing num1 and num2 as arguments.  
4. Perform calculations:  
 - Call calculate() method on addition object, store the result in result1.  
 - Print "Addition result: " concatenated with result1.  
 - Call calculate() method on subtraction object, store the result in result2.  
 - Print "Subtraction result: " concatenated with result2.  
 - Call calculate() method on multiplication object, store the result in result3.  
 - Print "Multiplication result: " concatenated with result3.  
 - Call calculate() method on division object, store the result in result4.  
 - Print "Division result: " concatenated with result4.  
5. End the main method.



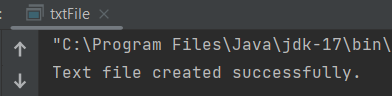
**Decimal to Binary:**1. Start the main method.  
2. Create a new instance of the Scanner class.  
3. Print "Enter a non-negative integer in decimal: ".  
4. Check if the user has entered an integer using hasNextInt().  
5. If an integer is entered, proceed to the next step. Otherwise, print "Error: Invalid input." and end the main method.  
6. Read the decimal number entered by the user using nextInt() and store it in decimal.  
7. Check if the entered decimal number is non-negative.  
8. If the number is non-negative, proceed to the next step. Otherwise, print "Error: Input must be a non-negative integer." and end the main method.  
9. Convert the decimal number to binary using toBinaryString() and store the result in binary.  
10. Convert the decimal number to hexadecimal using toHexString() and store the result in hexadecimal.  
11. Print "Decimal " concatenated with decimal, followed by " is equivalent to binary " concatenated with binary, and " and hexadecimal " concatenated with hexadecimal.  
12. End the main method.

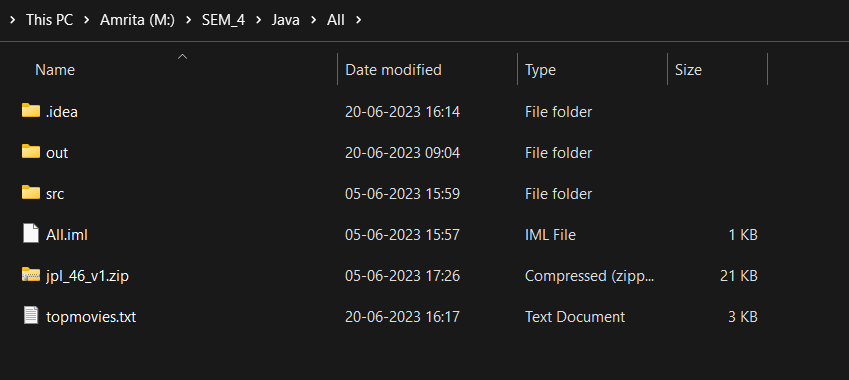


**Factorial:**1. Initialize num with the desired number  
2. Initialize factorial with 1  
3. for i = 1 to num do:  
 a. Multiply factorial by i and update factorial with the result  
4. Print "Factorial of num = factorial"

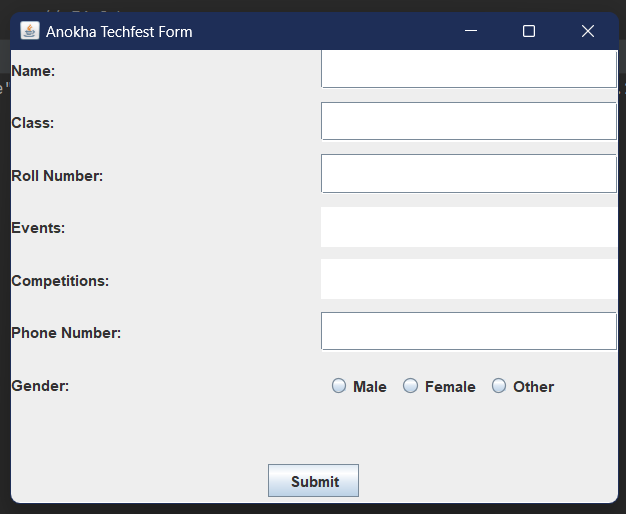


**File Handling:**1. Import the required classes.  
2. Define the main method.  
3. Try:  
 a. Create a BufferedWriter instance with a new FileWriter for the file "topmovies.txt".  
 b. Create an ArrayList named "movies" and populate it with the desired movie titles.  
 c. Loop through each movie in the "movies" ArrayList:  
 - Get the current movie using the index "i".  
 - Create a String variable named "movie" with the formatted movie entry.  
 - Write the "movie" String to the BufferedWriter.  
 d. Close the BufferedWriter.  
 e. Print a success message.  
4. Catch any IOException:  
 - Print a failure message.  
 - Print the stack trace.





**Form:**  
1. Import the required classes.  
2. Define the Form class that extends JFrame.  
3. Declare private fields for each component used in the form.  
4. Define the constructor for the Form class.  
 a. Set the title of the JFrame.  
 b. Set the size of the JFrame.  
 c. Set the default close operation.  
 d. Create the mainPanel using JPanel with BorderLayout and add it to the JFrame.  
 e. Create the formPanel using JPanel with GridLayout and add it to the mainPanel.  
 f. Create the required components for the form.  
 g. Add the components to the formPanel.  
 h. Create a radioButtonPanel using JPanel with FlowLayout and add the radio buttons to it.  
 i. Add the radioButtonPanel to the formPanel.  
 j. Create a submitButtonPanel using JPanel with FlowLayout and add the submit button to it.  
 k. Add the submitButtonPanel to the mainPanel.  
 l. Set the visibility of the JFrame to true.  
5. Define an anonymous ActionListener for the submitButton.  
 a. Retrieve the form data from the text fields, text areas, and radio buttons.  
 b. Print the form data.  
6. Define the main method.  
 a. Create an instance of the Form class.

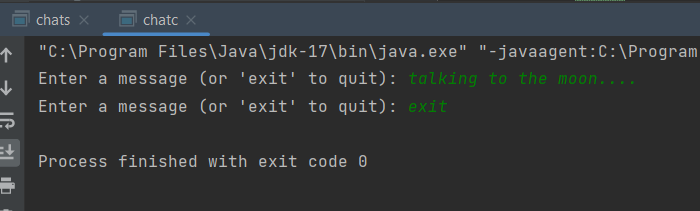


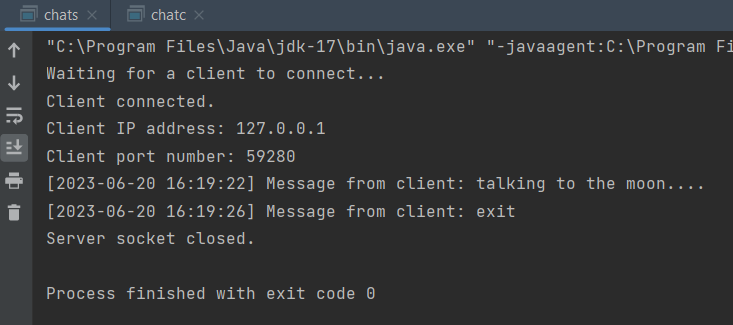
**Client Server**Client:

1. Start the main method.  
2. Create a socket object and establish a connection with the server using the hostname "localhost" and port number 2444.  
3. Create a DataOutputStream object called `dataOutputStream` to send messages to the server using the socket's output stream.  
4. Create a Scanner object called `scanner` to read user input from the console.  
5. Initialize an empty string variable called `message`.  
6. Enter a loop that continues until the user enters "exit":  
 a. Print the message "Enter a message (or 'exit' to quit):" to the console.  
 b. Read a line of input from the user using the `scanner.nextLine()` method and store it in the `message` variable.  
 c. Send the message to the server by writing it as a UTF string to the `dataOutputStream` using the `writeUTF(message)` method.  
 d. Flush the `dataOutputStream` to ensure the message is sent immediately.  
7. Exit the loop when the user enters "exit".  
8. Close the `dataOutputStream` using the `close()` method to release any system resources associated with it.  
9. Close the socket using the `close()` method to release any system resources associated with it.  
10. Handle any IOException that occurs during socket operations:  
 a. If an IOException is thrown, catch it and print an error message like "An error occurred: " followed by the exception message.

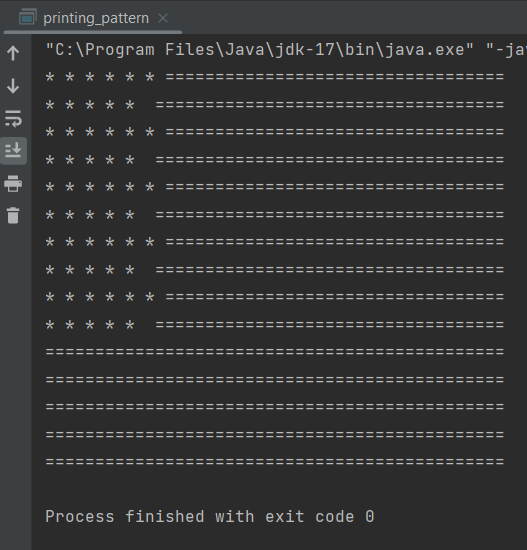
Server:

1. Import the required packages: `java.io.\*`, `java.net.\*`, and `java.time.\*`.  
2. Declare a public class named `chats` within the package `com.amrita.jpl.cys21046.practice.net`.  
3. Define the `main` method within the `chats` class, which is the entry point of the program. The method takes an array of strings `args` as a parameter.  
4. Within the `main` method, enclose the code in a try-catch block to handle any potential IOException that may occur.  
5. Inside the try block, create a new `ServerSocket` object named `serverSocket` and pass the port number `2444` as an argument.  
6. Print "Waiting for a client to connect..." to the console.  
7. Call the `accept()` method on the `serverSocket` object, which will block until a client connects to the server.  
8. Once a client connects, create a new `Socket` object named `clientSocket` to handle communication with the client.  
9. Print "Client connected." to the console.  
10. Retrieve the client details by calling `getInetAddress()` and `getPort()` methods on the `clientSocket` object. Store the client's IP address in an `InetAddress` object named `clientAddress` and the port number in an `int` variable named `clientPort`.  
11. Print the client's IP address and port number to the console.  
12. Create a new `DataInputStream` object named `dataInputStream` and initialize it with the input stream from the `clientSocket`.  
13. Declare a string variable named `message` and set it to an empty string.  
14. Start a while loop that continues until the `message` is "exit". The loop condition checks if `message` is not equal to "exit" using the `equalsIgnoreCase()` method.  
15. Inside the loop, read the next UTF-encoded string from the `dataInputStream` using the `readUTF()` method and assign it to the `message` variable.  
16. Get the current timestamp by creating a `LocalDateTime` object named `currentTime` using `LocalDateTime.now()`.  
17. Create a `DateTimeFormatter` object named `formatter` and initialize it with the pattern "yyyy-MM-dd HH:mm:ss".  
18. Format the `currentTime` object using the `format()` method of the `formatter` object and store it in a string variable named `formattedTime`.  
19. Print the received message along with the timestamp to the console, using the `System.out.println()` method.  
20. End the while loop.  
21. After the while loop, close the `serverSocket` using the `close()` method.  
22. Print "Server socket closed." to the console.  
23. If an IOException occurs in the try block, catch it in the catch block, and print the error message to the console.

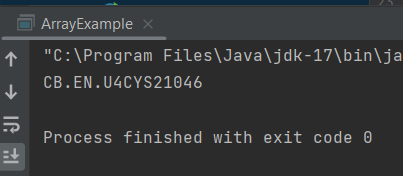




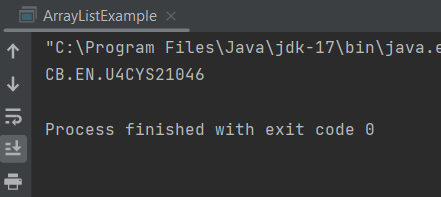
**Pattern:**1. Declare a public class named `printing\_pattern` within the package `com.amrita.jpl.cys21046.practice.pattern`.  
2. Define the `main` method within the `printing\_pattern` class, which is the entry point of the program. The method takes an array of strings `args` as a parameter.  
3. Inside the `main` method, use a `for` loop with the control variable `i` initialized to 1 and the loop condition `i <= 5`. This loop controls the number of rows in the pattern.  
4. Within the `for` loop, print the pattern line containing asterisks and equal signs using the `System.out.println()` method. The pattern line consists of six asterisks followed by 34 equal signs.  
5. After the first `System.out.println()` statement, write another `System.out.println()` statement to print the pattern line containing five asterisks followed by 35 equal signs.  
6. End the first `for` loop.  
7. Start a new `for` loop with the control variable `i` initialized to 1 and the loop condition `i <= 5`. This loop controls the number of rows in the last part of the pattern.  
8. Within this `for` loop, print a pattern line consisting of 46 equal signs using the `System.out.println()` method.  
9. End the second `for` loop.



**Data structures:  
1) Array example**1. Start the ArrayExample class.  
2. Print any relevant introductory comments or documentation.  
3. Start the main method.  
4. Declare and initialize an array of strings named rollNumber with a specified size of 1.  
5. Assign the value "CB.EN.U4CYS21046" to the first element of the rollNumber array.  
6. Start a for loop with the loop variable i initialized to 0 and the loop condition i < rollNumber.length.  
7. Within the loop, print the value of the current element rollNumber[i].  
8. End the for loop.  
9. End the main method.  
10. End the ArrayExample class.

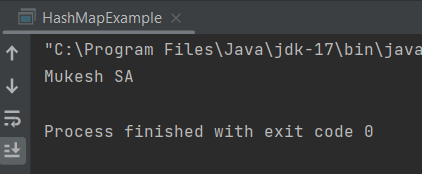
**  
  
2) Array list example**

1. Start the ArrayListExample class.  
2. Print any relevant introductory comments or documentation.  
3. Start the main method.  
4. Create an ArrayList of strings named u21cys to represent student roll numbers.  
5. Add the value "CB.EN.U4CYS21046" to the u21cys ArrayList.  
6. Get an iterator from the u21cys ArrayList using the iterator() method and assign it to a variable named it.  
7. Print the value of the first element in the u21cys ArrayList using the next() method of the iterator it.  
8. End the main method.  
9. End the ArrayListExample class.



**3) Hashmap example**

1. Start the HashMapExample class.  
2. Print any relevant introductory comments or documentation.  
3. Start the main method.  
4. Create a HashMap object named rollName to store roll numbers and names.  
5. Add a key-value pair to the rollName HashMap using the put() method, where the key is the roll number "CB.EN.U4CYS21046" and the value is the corresponding name "Mukesh SA".  
6. Retrieve the value associated with a specific key from the rollName HashMap using the get() method, where the key is "CB.EN.U4CYS21046".  
7. Print the retrieved name.  
8. End the main method.  
9. End the HashMapExample class.



**4) linked list example**

1. Start the LinkedListExample class.  
2. Print any relevant introductory comments or documentation.  
3. Start the main method.  
4. Create a LinkedList object named u21cys to store student roll numbers.  
5. Add elements to the u21cys LinkedList using the add() method.  
 - Add the value "CB.EN.U4CYS22046" to the u21cys LinkedList.  
 - Add the value "CB.EN.U4CYS22047" to the u21cys LinkedList.  
6. Print the contents of the u21cys LinkedList.  
7. End the main method.  
8. End the LinkedListExample class.

