EXPERIMENT - 6

AIM:

Java program to do String Manipulation using CharacterArray and perform the following string operations.

ALGORITHM:

Step 1: Start the Process.

Step 2: Import **Scanner** for reading user input.

Step 3: Create a **Scanner** object for user input.

Step 4: Prompt the user to enter the first string, read it into **firstString,** and convert it to a character array **firstCharArray.**

Step 5: Get the length of the **firstCharArray** and print the length of the first string.

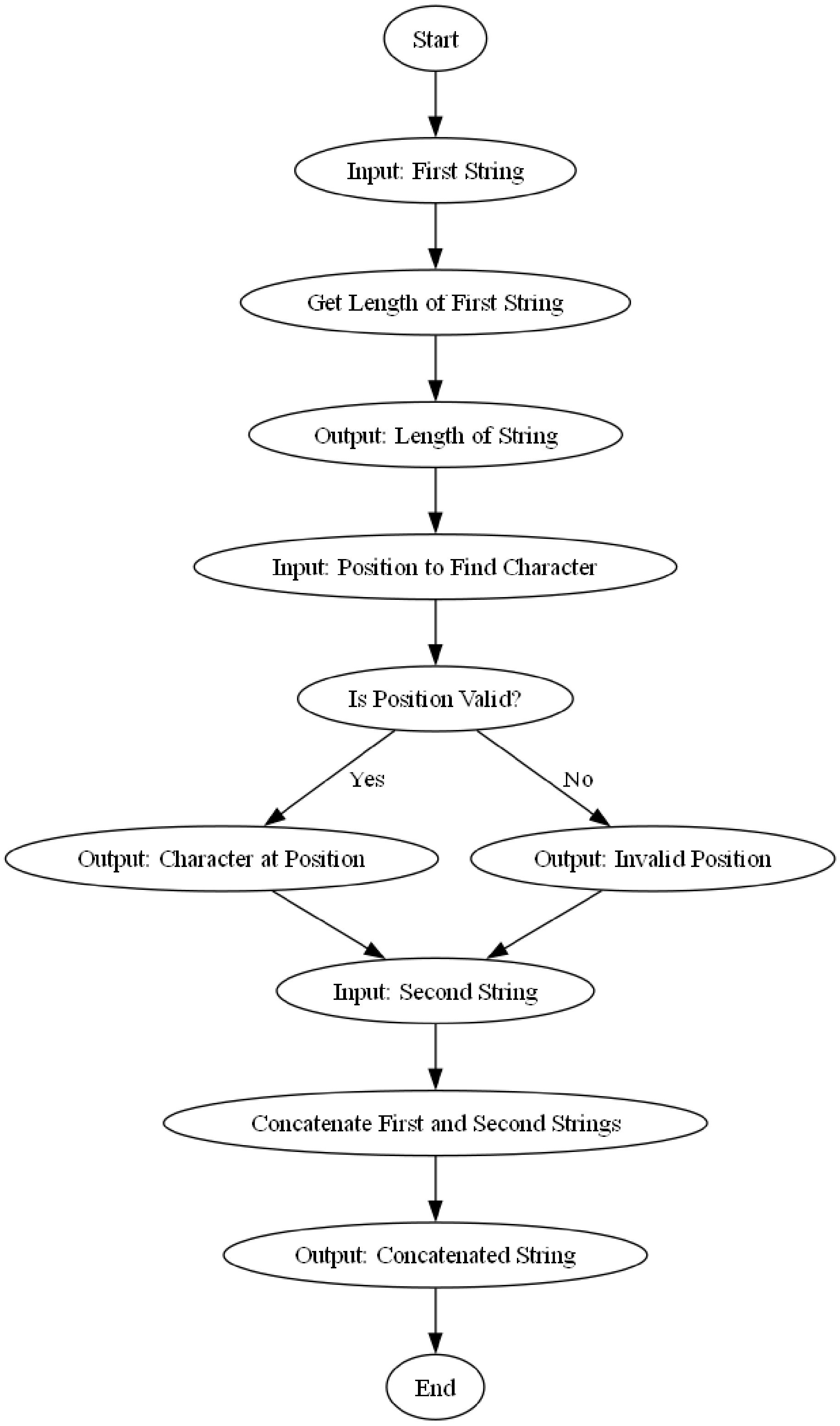
Step 6: Prompt the user to enter a position (index) to find the character, and read it into **position.** If valid **(0 to length - 1),** print **firstCharArray[position];** else print "Invalid position!"

Step 7: Clear the input buffer (optional), prompt the user to enter the second string, and read it into **secondString.**

Step 8: Concatenate **firstString** and **secondString** into a **concatenated string** and print it.

Step 9: End the Process.

FLOW CHART:


Start
Input: First String
Get Length of First String
Output: Length of String
Input: Position to Find Character
Is Position Valid?
Yes
No
Output: Character at Position
Output: Invalid Position
Input: Second String
Concatenate First and Second Strings
Output: Concatenated String
End

SOURCE CODE:

```
import java.util.Scanner;
public class StringManipulation {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Input for the first string
        System.out.print("Enter the first string: ");
        String firstString = scanner.nextLine();
        char[] firstCharArray = firstString.toCharArray(); // Convert to character array
        // a) Get the length of the string
        int length = firstCharArray.length;
        System.out.println("Length of the first string: " + length);
        // b) Finding a character at a particular position
        System.out.print("Enter a position to find the character (0 to " + (length - 1) + "): ");
        int position = scanner.nextInt();
        if (position >= 0 && position < length) {
            char characterAtPosition = firstCharArray[position];
            System.out.println("Character at position " + position + ": " + characterAtPosition);
        } else {
            System.out.println("Invalid position!");
        }
        // Input for the second string
        scanner.nextLine(); // Clear the buffer
        System.out.print("Enter the second string: ");
        String secondString = scanner.nextLine();
        // c) Concatenating two strings
        String concatenatedString = firstString + secondString;
        System.out.println("Concatenated string: " + concatenatedString);
    }
}
```

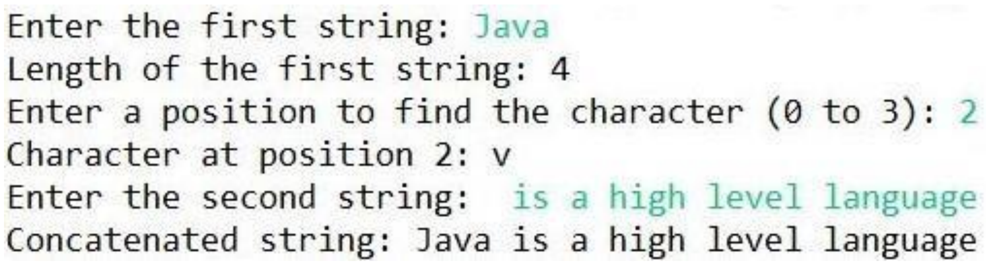
CODE EXPLANATION:

1. **import java.util.Scanner;**
   - This imports the Scanner class, which is used to read input from the user.
2. **public class StringManipulation {**
   - Declares a class named StringManipulation, which contains the program logic.
3. **public static void main(String[] args) {**
   - Defines the main method, which is the entry point of the program.
4. **Scanner scanner = new Scanner(System.in);**
   - Creates a Scanner object to read input from the console.
5. **System.out.print("Enter the first string: ");**
   - Prompts the user to enter the first string.
6. **String firstString = scanner.nextLine();**
   - Reads the first string input from the user, including spaces.
7. **char[] firstCharArray = firstString.toCharArray();**
   - Converts the first string into a character array for easier manipulation.
8. **int length = firstCharArray.length;**
   - Calculates the length of the first string using the length property of the character array.
9. **System.out.println("Length of the first string: " + length);**
   - Prints the length of the first string.
10. **System.out.print("Enter a position to find the character (0 to " + (length - 1) + "): ");**
    - Prompts the user to enter a position to retrieve a character from the first string.
11. **int position = scanner.nextInt();**
    - Reads the position input as an integer.
12. **if (position >= 0 && position < length) {**
    - Checks if the entered position is within the valid range (0 to length-1).
13. **char characterAtPosition = firstCharArray[position];**
    - Retrieves the character at the specified position from the character array.
14. **System.out.println("Character at position " + position + ": " + characterAtPosition);**
    - Prints the character found at the specified position.
15. **} else {**

- Executes if the position entered is invalid.

16. **System.out.println("Invalid position!");**
    - Prints a message indicating the entered position is not valid.
17. **scanner.nextLine();**
    - Clears the scanner buffer to avoid issues when switching between nextInt and nextLine.
18. **System.out.print("Enter the second string: ");**
    - Prompts the user to enter the second string.
19. **String secondString = scanner.nextLine();**
    - Reads the second string input from the user, including spaces.
20. **String concatenatedString = firstString + secondString;**
    - Concatenates the first and second strings using the + operator.
21. **System.out.println("Concatenated string: " + concatenatedString);**
    - Prints the concatenated result of the two strings.
22. **}**
    - Closes the main method.
23. **}**
    - Closes the class definition.

OUTPUT:



```
Enter the first string: Java
Length of the first string: 4
Enter a position to find the character (0 to 3): 2
Character at position 2: v
Enter the second string:  is a high level language
Concatenated string: Java is a high level language
```

RESULT:

Thus, the program had been successfully executed.

EXPERIMENT - 7

AIM:

To develop a Java program that demonstrates fundamental string operations, including String Concatenation, Substring Search, and Substring Extraction.

ALGORITHM:

Step 1: Start the Process.

Step 2: Initialize Scanner for user input.

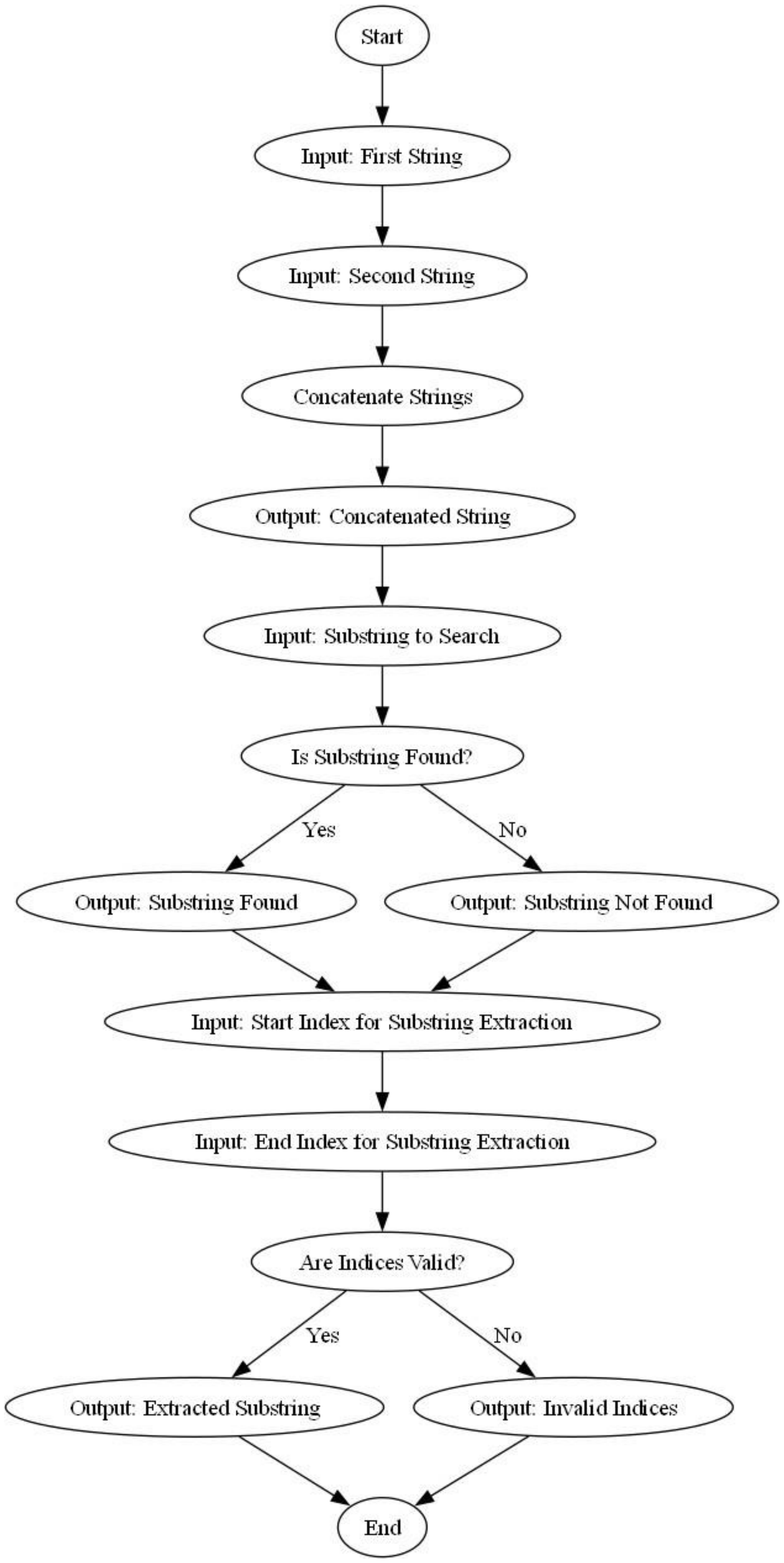
Step 3: Read the first string (str1) and second string (str2). Concatenate them into **concatenated** and display.

Step 4: Read the substring to search. Check if **concatenated** contains it; display "Substring found!" or "Substring not found.".

Step 5: Read start and end indices. If valid, extract substring from **concatenated** and display; else, display "Invalid indices.".

Step 6: End the Process.

FLOW CHART:


Start
Input: First String
Input: Second String
Concatenate Strings
Output: Concatenated String
Input: Substring to Search
Is Substring Found?
Yes
No
Output: Substring Found
Output: Substring Not Found
Input: Start Index for Substring Extraction
Input: End Index for Substring Extraction
Are Indices Valid?
Yes
No
Output: Extracted Substring
Output: Invalid Indices
End

SOURCE CODE:

```
import java.util.Scanner;
public class SimpleStringOperations {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Concatenation
        System.out.print("Enter the first string: ");
        String str1 = scanner.nextLine();
        System.out.print("Enter the second string: ");
        String str2 = scanner.nextLine();
        String concatenated = str1 + str2;
        System.out.println("Concatenated String: " + concatenated);
        // Search for a substring
        System.out.print("Enter a substring to search: ");
        String substring = scanner.nextLine();
        if (concatenated.contains(substring)) {
            System.out.println("Substring found!");
        } else {
            System.out.println("Substring not found.");
        }
            // Extract a substring
        System.out.print("Enter start index: ");
        int start = scanner.nextInt();
        System.out.print("Enter end index: ");
        int end = scanner.nextInt();
        if (start >= 0 && end <= concatenated.length()) {
            String extracted = concatenated.substring(start, end);
            System.out.println("Extracted Substring: " + extracted);
        } else {
            System.out.println("Invalid indices.");
        }
    }
}
```

CODE EXPLANATION:

1. **import java.util.Scanner;**
    - Imports the Scanner class, which allows reading user input from the console.
2. **public class SimpleStringOperations {**
    - Declares the class SimpleStringOperations, where the program logic will be implemented.
3. **public static void main(String[] args) {**
    - Defines the main method, which is the entry point of the program.
4. **Scanner scanner = new Scanner(System.in);**
    - Creates a Scanner object named scanner to read input from the user.
5. **System.out.print("Enter the first string: ");**
    - Prompts the user to enter the first string.
6. **String str1 = scanner.nextLine();**
    - Reads the first string entered by the user and stores it in the variable str1.
7. **System.out.print("Enter the second string: ");**
    - Prompts the user to enter the second string.
8. **String str2 = scanner.nextLine();**
    - Reads the second string entered by the user and stores it in the variable str2.
9. **String concatenated = str1 + str2;**
    - Concatenates str1 and str2 using the + operator and stores the result in the variable concatenated.
10. **System.out.println("Concatenated String: " + concatenated);**
    - Prints the concatenated string to the console.
11. **System.out.print("Enter a substring to search: ");**
    - Prompts the user to enter a substring that they want to search for in the concatenated string.
12. **String substring = scanner.nextLine();**
    - Reads the substring entered by the user and stores it in the variable substring.

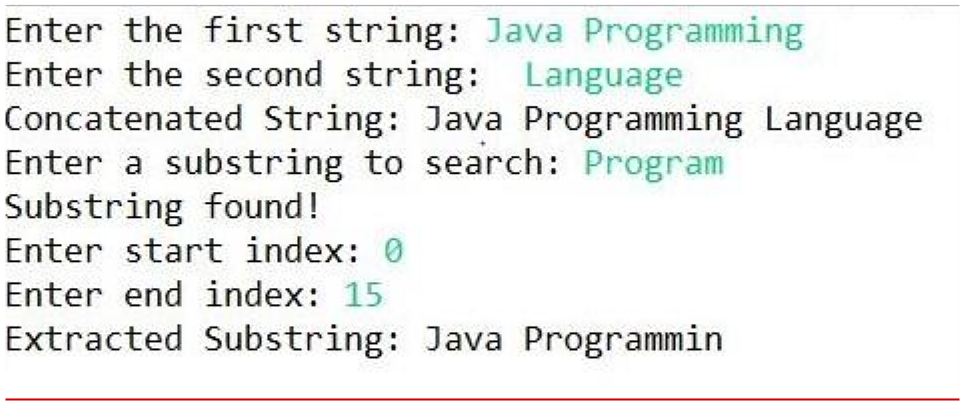
13. **if (concatenated.contains(substring)) {**
    - Checks if the concatenated string contains the entered substring using the contains method.
14. **System.out.println("Substring found!");**
    - If the substring is found, it prints "Substring found!".
15. **} else {**
    - If the substring is not found, it executes the code in this block.
16. **System.out.println("Substring not found.");**
    - Prints "Substring not found." if the substring is not found in the concatenated string.
17. **System.out.print("Enter start index: ");**
    - Prompts the user to enter the start index for the substring extraction.
18. **int start = scanner.nextInt();**
    - Reads the start index entered by the user and stores it in the variable start.
19. **System.out.print("Enter end index: ");**
    - Prompts the user to enter the end index for the substring extraction.
20. **int end = scanner.nextInt();**
    - Reads the end index entered by the user and stores it in the variable end.
21. **if (start >= 0 && end <= concatenated.length()) {**
    - Checks if the entered start and end indices are valid. Specifically, it ensures that the start index is greater than or equal to 0 and the end index is less than or equal to the length of the concatenated string.
22. **String extracted = concatenated.substring(start, end);**
    - If the indices are valid, it extracts the substring from the concatenated string using the substring method, from start to end index, and stores it in the extracted variable.
23. **System.out.println("Extracted Substring: " + extracted);**
    - Prints the extracted substring to the console.
24. **} else {**
    - If the indices are invalid, this block is executed.
25. **System.out.println("Invalid indices.");**
    - Prints an error message indicating the indices provided are invalid.
26. **}**

   - Closes the if-else block for extracting the substring.

27. }
   - Closes the main method.

28. }
   - Closes the class definition.

OUTPUT:



```
Enter the first string: Java Programming
Enter the second string:  Language
Concatenated String: Java Programming Language
Enter a substring to search: Program
Substring found!
Enter start index: 0
Enter end index: 15
Extracted Substring: Java Programmin
```

RESULT:

Thus, the program had been successfully executed.

EXPERIMENT - 8

AIM:

To create a Java program that utilizes the StringBuffer class to perform string operations: determine string length, reverse the string, and delete a specified substring.

ALGORITHM:

Step 1: Start the Process.

Step 2: Initialize Scanner for user input.

Step 3: Read a string to create a StringBuffer object (sb).

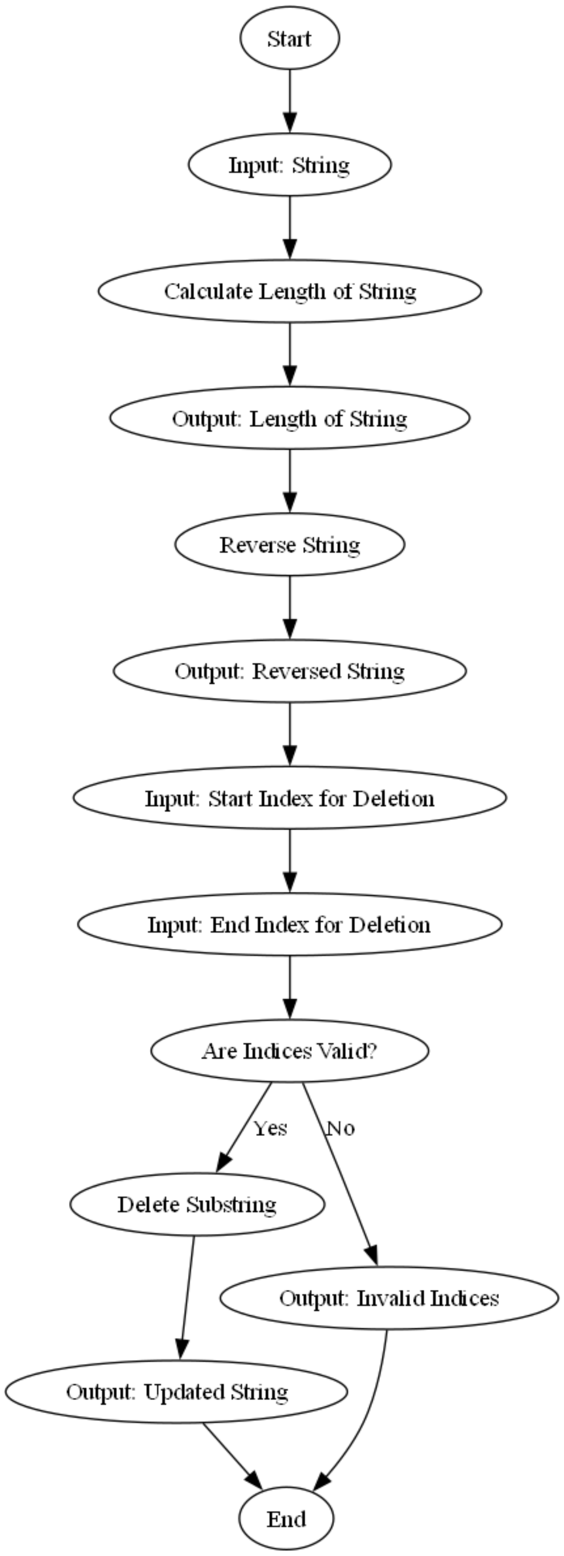
Step 4: Find and display the length of sb using the length() method.

Step 5: Reverse sb using the reverse() method and display it.

Step 6: Read start and end indices for deletion. If valid (start >= 0, end <= length of sb, start < end), delete the substring using delete(start, end) and display the modified string; else, display "Invalid indices.".

Step 7: End the Process.

FLOW CHART:


Start
Input: String
Calculate Length of String
Output: Length of String
Reverse String
Output: Reversed String
Input: Start Index for Deletion
Input: End Index for Deletion
Are Indices Valid?
Yes
No
Delete Substring
Output: Invalid Indices
Output: Updated String
End

SOURCE CODE:

```
import java.util.Scanner;
public class SimpleStringBuffer {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Input string
        System.out.print("Enter a string: ");
        StringBuffer sb = new StringBuffer(scanner.nextLine());
        // Length of the string
        System.out.println("Length: " + sb.length());
        // Reverse the string
        System.out.println("Reversed: " + sb.reverse());
        // Delete a substring
        System.out.print("Start index to delete: ");
        int start = scanner.nextInt();
        System.out.print("End index to delete: ");
        int end = scanner.nextInt();
        if (start >= 0 && end <= sb.length() && start < end) {
            sb.delete(start, end);
            System.out.println("After deletion: " + sb);
        } else {
            System.out.println("Invalid indices.");
        }
    }
}
```

CODE EXPLANATION:

1. **import java.util.Scanner;**
   - Imports the Scanner class to read user input from the console.
2. **public class SimpleStringBuffer {**

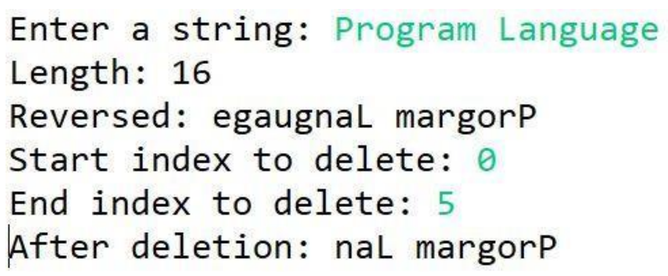
- Defines the class SimpleStringBuffer, where the program logic will be implemented.

3. **public static void main(String[] args) {**
    - Declares the main method, which serves as the entry point of the program.
4. **Scanner scanner = new Scanner(System.in);**
    - Creates a Scanner object called scanner to read input from the user.
5. **System.out.print("Enter a string: ");**
    - Prompts the user to enter a string.
6. **StringBuffer sb = new StringBuffer(scanner.nextLine());**
    - Reads the string entered by the user and stores it in a StringBuffer object named sb. StringBuffer allows us to modify the string.
7. **System.out.println("Length: " + sb.length());**
    - Prints the length of the string using the length() method of StringBuffer.
8. **System.out.println("Reversed: " + sb.reverse());**
    - Reverses the string using the reverse() method of StringBuffer and prints the reversed string.
9. **System.out.print("Start index to delete: ");**
    - Prompts the user to enter the start index for deleting a substring.
10. **int start = scanner.nextInt();**
    - Reads the start index entered by the user and stores it in the variable start.
    -
11. **System.out.print("End index to delete: ");**
    - Prompts the user to enter the end index for deleting a substring.
12. **int end = scanner.nextInt();**
    - Reads the end index entered by the user and stores it in the variable end.
13. **if (start >= 0 && end <= sb.length() && start < end) {**
    - Checks if the entered indices are valid. The start index must be greater than or equal to 0, the end index must be less than or equal to the length of the string, and the start index must be less than the end index.
14. **sb.delete(start, end);**

  - If the indices are valid, it deletes the substring from the start index to the end index using the delete() method of StringBuffer.

15. **System.out.println("After deletion: " + sb);**
    - Prints the modified string after the deletion.
16. **} else {**
    - If the indices are invalid, this block is executed.
17. **System.out.println("Invalid indices.");**
    - Prints an error message indicating that the indices provided are invalid.
18. **}**
    - Closes the if-else block.
19. **}**
    - Closes the main method.
20. **}**
    - Closes the class definition.

OUTPUT:



```
Enter a string: Program Language
Length: 16
Reversed: egaugnaL margorP
Start index to delete: 0
End index to delete: 5
After deletion: naL margorP
```

RESULT:

Thus, the program had been successfully executed.

EXPERIMENT - 9

AIM:

To implement a multi-threaded Java application that generates random integers and computes their squares if even or cubes if odd, using three separate threads.

ALGORITHM:

Step 1: Start the Process.

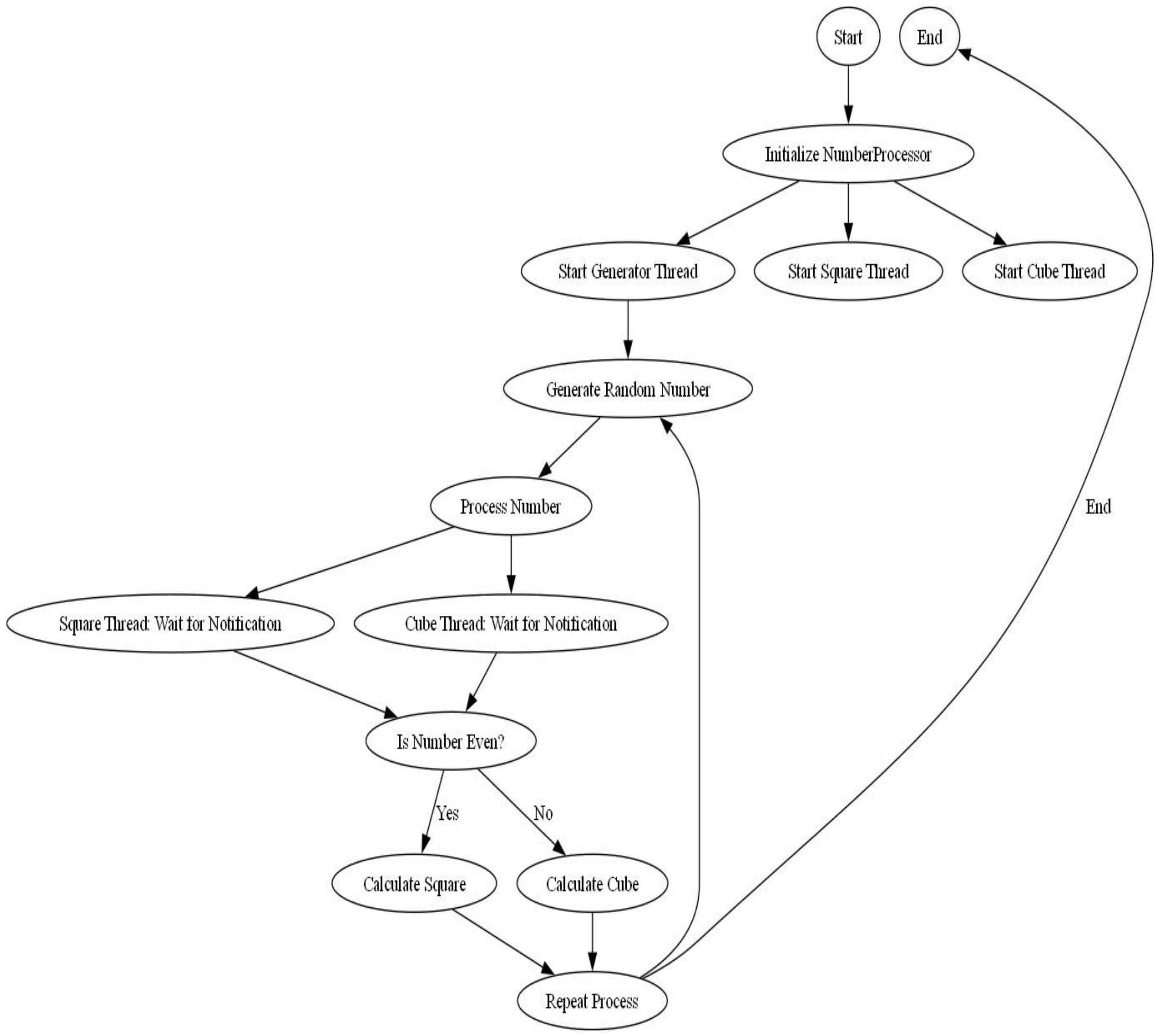
Step 2: Create the **RandomNumberGenerator** Class. Create a **NumberProcessor** object. In the **run()** method, generate a random integer between 0 and 99, print it, call **processNumber(number),** and sleep for 1 second.

Step 3: Create the **NumberProcessor** Class. Declare an integer called **number.** In the **processNumber(int number)** method, set **number** and notify waiting threads. In the **square()** method, wait for a number; if it's even, print its square. In the **cube()** method, wait for a number; if it's odd, print its cube.

Step 4: Create the **MultiThreadedRandomNumber** Class. In the **main()** method**,** create a **NumberProcessor** instance. Start the **RandomNumberGenerator** thread, the **square()** thread, and the **cube()** thread.

Step 5: End the Process.

FLOW CHART:



SOURCE CODE:

```
import java.util.Random;
class RandomNumberGeneratorEx extends Thread {
   private final NumberProcessor processor;
   public RandomNumberGeneratorEx(NumberProcessor processor) {
      this.processor = processor;
   }
```

```
    @Override
    public void run() {
        Random random = new Random();
        while (true) {
            int number = random.nextInt(100);
            System.out.println("Generated: " + number);
            processor.processNumber(number);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                break;
            }
        }
    }
}
class NumberProcessor {
    private int number;
    public synchronized void processNumber(int number) {
        this.number = number;
        notifyAll();
    }
    public void square() {
        while (true) {
            synchronized (this) {
                try {
                    wait();
                    if (number % 2 == 0) {
                        System.out.println("Square: " + (number * number));
                    }
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    }
    public void cube() {
        while (true) {
            synchronized (this) {
                try {
                    wait();
                    if (number % 2 != 0) {
                        System.out.println("Cube: " + (number * number * number));
                    }
                } catch (InterruptedException e) {
                    break;
                }
            }
        }
    }
}

public class MultiThreadedRandomNumber {
    public static void main(String[] args) {
        NumberProcessor processor = new NumberProcessor();
        RandomNumberGeneratorEx generator = new RandomNumberGeneratorEx(processor);
        Thread squareThread = new Thread(processor::square);
        Thread cubeThread = new Thread(processor::cube);
        generator.start();
        squareThread.start();
        cubeThread.start();
    }
}
```

CODE EXPLANATION:

1. **RandomNumberGeneratorEx extends Thread {**
   - The RandomNumberGeneratorEx class is a subclass of the Thread class, meaning it will execute in its own thread of execution.
2. **private final NumberProcessor processor;**
   - A private member processor of type NumberProcessor is declared. This will be used to process numbers.
3. **public RandomNumberGeneratorEx(NumberProcessor processor) {**
   - Constructor to initialize the processor object. This is called when a RandomNumberGeneratorEx object is created.
4. **this.processor = processor;**
   - Inside the constructor, the processor parameter is assigned to the class's processor variable.
5. **@Override public void run() {**
   - The run() method is overridden. This is the entry point for the thread execution.
6. **Random random = new Random();**
   - A Random object is created to generate random numbers.
7. **while (true) {**
   - Starts an infinite loop where random numbers will continuously be generated.
8. **int number = random.nextInt(100);**
   - Generates a random integer between 0 and 99.
9. **System.out.println("Generated: " + number);**
   - Prints the generated random number to the console.
10. **processor.processNumber(number);**
    - Passes the generated number to the processNumber() method of NumberProcessor.
11. **try { Thread.sleep(1000); }**
    - The thread sleeps for 1000 milliseconds (1 second), creating a pause before generating the next random number.
    -
12. **catch (InterruptedException e) { break; }**

- If the thread is interrupted, it will break out of the loop and stop execution.

13. **}**
    - Ends the run() method.

NumberProcessor Class:

14. **class NumberProcessor {**
    - The NumberProcessor class is responsible for processing the generated numbers.
15. **private int number;**
    - A private number variable that stores the current number being processed.
16. **public synchronized void processNumber(int number) {**
    - This method is synchronized to ensure that only one thread at a time can execute it.
    - It takes the number as input and processes it.
17. **this.number = number;**
    - The input number is assigned to the class's number variable.
18. **notifyAll();**
    - This call notifies all waiting threads that they can proceed (i.e., they are allowed to process the number).
19. **}**
    - Ends the processNumber() method.

Square Method:

20. **public void square() {**
    - Defines the square() method to process the number and compute its square.
21. **while (true) {**
    - Starts an infinite loop that continues to process numbers.
22. **synchronized (this) {**
    - This block is synchronized to ensure mutual exclusion while accessing the shared number variable.
23. **try { wait(); }**
    - The wait() method causes the thread to wait until notified by another thread.

24. **if (number % 2 == 0) {**
    - Check if the current number is even.
25. **System.out.println("Square: " + (number * number));**
    - If the number is even, it calculates the square and prints it.
26. **} catch (InterruptedException e) { break; }**
    - If the thread is interrupted, it breaks out of the loop and stops execution.
27. **}**
    - Ends the synchronized block.
28. **}**
    - Ends the square() method.

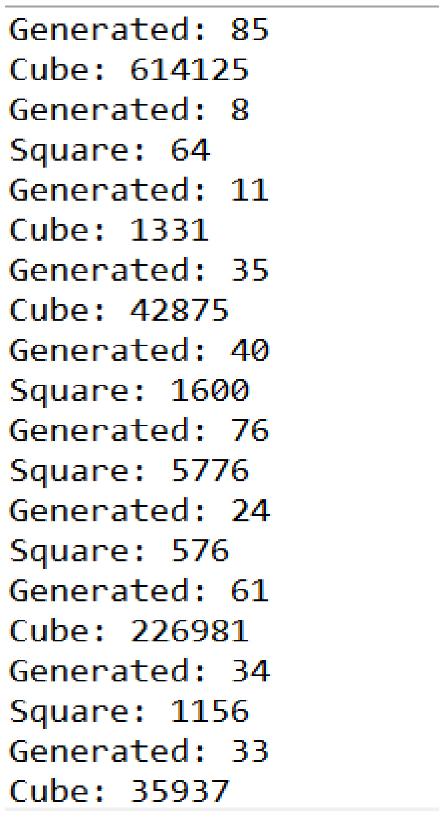
Cube Method:

29. **public void cube() {**
    - Defines the cube() method to process the number and compute its cube.
30. **while (true) {**
    - Starts an infinite loop to process numbers.
31. **synchronized (this) {**
    - The synchronized block ensures that only one thread can process the number at a time.
32. **try { wait(); }**
    - Causes the thread to wait until it is notified by another thread.
33. **if (number % 2 != 0) {**
    - Check if the current number is odd.
34. **System.out.println("Cube: " + (number * number * number));**
    - If the number is odd, it calculates and prints the cube.
35. **} catch (InterruptedException e) { break; }**
    - If the thread is interrupted, it exits the loop and stops execution.
36. **}**
    - Ends the synchronized block.
37. **}**
    - Ends the cube() method.

MultiThreadedRandomNumber (Main Class):

38. **public class MultiThreadedRandomNumber {**
    - The main class where the program execution begins.
39. **public static void main(String[] args) {**
    - The main() method is the entry point of the program.
40. **NumberProcessor processor = new NumberProcessor();**
    - Creates a new instance of NumberProcessor, which will process the random numbers.
41. **RandomNumberGeneratorEx generator = new RandomNumberGeneratorEx(processor);**
    - Creates a new RandomNumberGeneratorEx object, passing the processor to it.
42. **Thread squareThread = new Thread(processor::square);**
    - Creates a new thread that will run the square() method from NumberProcessor.
43. **Thread cubeThread = new Thread(processor::cube);**
    - Creates a new thread that will run the cube() method from NumberProcessor.
44. **generator.start();**
    - Starts the RandomNumberGeneratorEx thread, which begins generating random numbers.
45. **squareThread.start();**
    - Starts the thread that calculates the square of even numbers.
46. **cubeThread.start();**
    - Starts the thread that calculates the cube of odd numbers.
47. **}**
    - Ends the main() method.
48. **}**
    - Ends the MultiThreadedRandomNumber class.

OUTPUT:



```
Generated: 85
Cube: 614125
Generated: 8
Square: 64
Generated: 11
Cube: 1331
Generated: 35
Cube: 42875
Generated: 40
Square: 1600
Generated: 76
Square: 5776
Generated: 24
Square: 576
Generated: 61
Cube: 226981
Generated: 34
Square: 1156
Generated: 33
Cube: 35937
```

RESULT:

Thus, the program had been successfully executed.

EXPERIMENT - 10

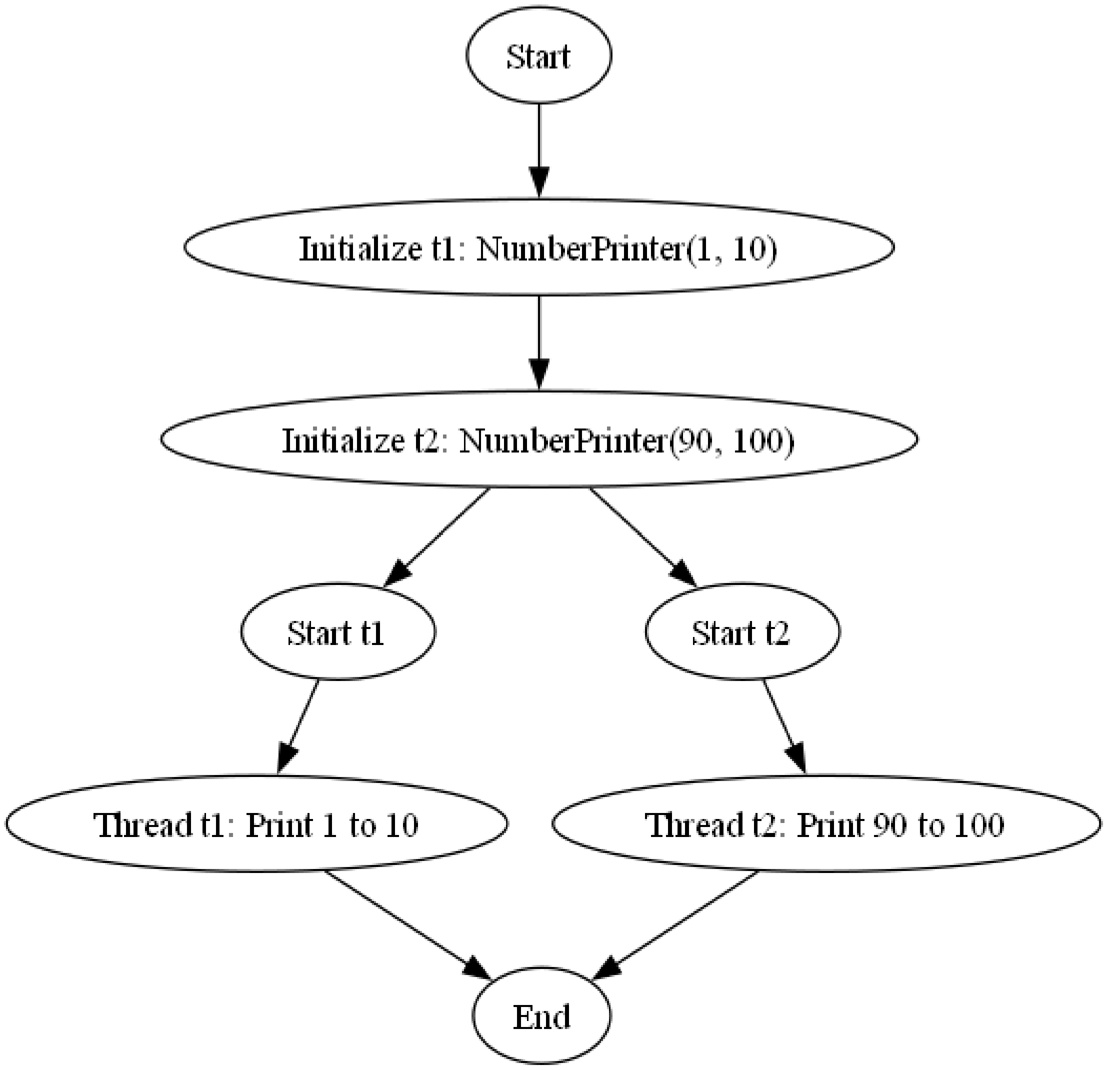
AIM:

To implement a Java program that uses multi-threading to print numbers from 1 to 10 and from 90 to 100 asynchronously using the same method.

ALGORITHM:

Step 1: Start.
Step 2: Define the NumberPrinter Class.
Step 3: Declare attributes int start for the starting number and int end for the ending number.
Step 4: Create a constructor to initialize start and end with given values.
Step 5: In the run() method, loop from start to end (inclusive) and print the current number. Sleep for 500 milliseconds and handle any InterruptedException by restoring the thread's interrupted status.
Step 6: Define the AsyncNumberPrinting Class. In the main() method, create a Thread object t1 for printing numbers from 1 to 10 using NumberPrinter.
Step 7: Create another Thread object t2 for printing numbers from 90 to 100 using NumberPrinter. Start both threads using t1.start() and t2.start().
Step 8: End the Process.

FLOW CHART:


Start
Initialize t1: NumberPrinter(1, 10)
Initialize t2: NumberPrinter(90, 100)
Start t1
Start t2
Thread t1: Print 1 to 10
Thread t2: Print 90 to 100
End

SOURCE CODE:

```
class NumberPrinter extends Thread {
    private final int start;
    private final int end;
    public NumberPrinter(int start, int end) {
        this.start = start;
        this.end = end;
    }
    @Override
    public void run() {
        for (int i = start; i <= end; i++) {
            System.out.println(i);
            try {
                Thread.sleep(500); // Pause for visibility
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
public class AsyncNumberPrinting {
    public static void main(String[] args) {
        Thread t1 = new NumberPrinter(1, 10);
        Thread t2 = new NumberPrinter(90, 100);
        t1.start();
        t2.start();
    }
}
```

CODE EXPLANATION:

NumberPrinter Class:

1. **class NumberPrinter extends Thread {**
   - Defines the NumberPrinter class, which extends the Thread class. This means the class will run in its own separate thread of execution.
2. **private final int start;**
   - Declares a private member variable start of type int. This will store the starting number from which the printing begins.

3. **private final int end;**
   - Declares a private member variable end of type int. This will store the ending number up to which the numbers will be printed.
4. **public NumberPrinter(int start, int end) {**
   - The constructor for the NumberPrinter class, which takes two parameters start and end. These parameters define the range of numbers to print.
5. **this.start = start;**
   - Assigns the start parameter value to the start member variable of the NumberPrinter class.
6. **this.end = end;**
   - Assigns the end parameter value to the end member variable of the NumberPrinter class.
7. **@Override public void run() {**
   - Overrides the run() method from the Thread class. This is the method that gets executed when the thread starts.
8. **for (int i = start; i <= end; i++) {**
   - Starts a for loop that begins at the start value and runs until i equals end. Each iteration prints a number.
9. **System.out.println(i);**
   - Prints the current value of i to the console, which is the current number in the loop.
10. **try { Thread.sleep(500); }**
    - Pauses the execution of the current thread for 500 milliseconds (half a second) to allow for better visibility of the printed numbers.
11. **catch (InterruptedException e) {**
    - Catches any InterruptedException that may occur during the Thread.sleep() operation. This exception occurs if the thread is interrupted while sleeping.
12. **Thread.currentThread().interrupt();**
    - Restores the interrupted status of the current thread after the exception is caught. This ensures that the interruption is not lost, and can be handled appropriately if needed.
13. **}**

- Ends the try-catch block.

14. **}**
    - Ends the for loop.
15. **}**
    - Ends the run() method.
16. **}**
    - Ends the NumberPrinter class.

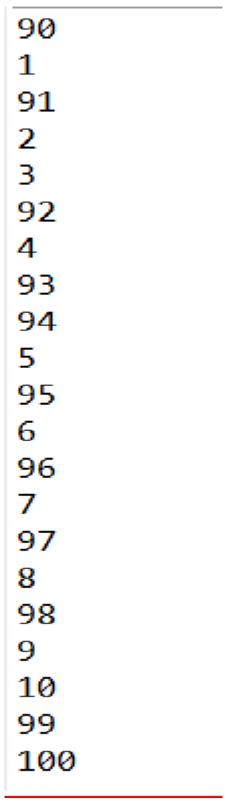
AsyncNumberPrinting (Main Class):

17. **public class AsyncNumberPrinting {**
    - Defines the AsyncNumberPrinting class, which contains the main() method where the execution begins.
18. **public static void main(String[] args) {**
    - The main() method is the entry point of the program. It's where the execution starts when you run the program.
19. **Thread t1 = new NumberPrinter(1, 10);**
    - Creates a new instance of the NumberPrinter class called t1, and initializes it with a range from 1 to 10. This thread will print the numbers from 1 to 10.
20. **Thread t2 = new NumberPrinter(90, 100);**
    - Creates a new instance of the NumberPrinter class called t2, and initializes it with a range from 90 to 100. This thread will print the numbers from 90 to 100.
21. **t1.start();**
    - Starts the thread t1, which invokes the run() method of NumberPrinter. This will begin printing the numbers from 1 to 10.

22. **t2.start();**
    - Starts the thread t2, which also invokes the run() method of NumberPrinter. This will begin printing the numbers from 90 to 100.
23. **}**
    - Ends the main() method.
24. **}**

- Ends the AsyncNumberPrinting class.

OUTPUT:



```
90
1
91
2
3
92
4
93
94
5
95
6
96
7
97
8
98
9
10
99
100
```

RESULT:

Thus, the program had been successfully executed.