1. Write a program that compares two strings and checks for substring presence.

Aim:

To compare two strings for equality using various methods and to check for the presence of a substring in one of the strings.

Objectives:

1. To demonstrate the use of **equals(), equalsIgnoreCase()**, and **compareTo()** methods for string comparison.
2. To check if one string contains another as a substring.

Theory:

- **String Comparison:**
  - **equals()**: Compares two strings for exact match (case-sensitive).
  - **equalsIgnoreCase()**: Compares two strings for equality (case-insensitive).
  - **compareTo()**: Lexicographically compares two strings based on Unicode value.
- **Substring Check:** The **contains()** method checks if a string contains a specified sequence of characters.

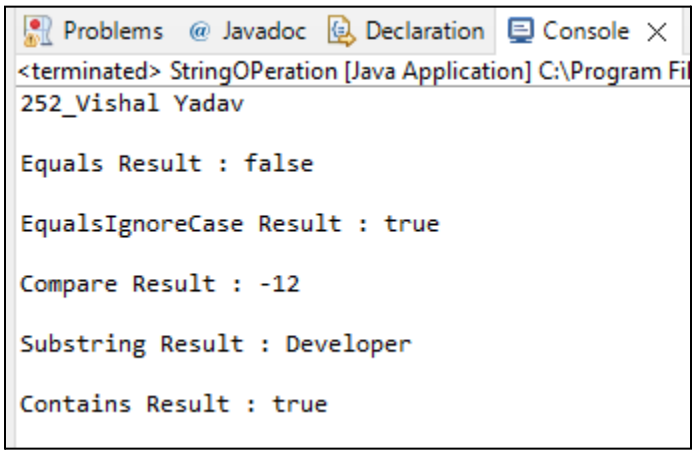
Code:

StringOperation.java

```
package MyFirstPackage;

public class StringOPeration {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		String str1 = "Java Developer";
		String str2 = "java Developer";
		String str3 = new String("Java Programming");
		boolean equalsresult = str1.equals(str2);
		boolean equalsIgnoreCaseresult = str1.equalsIgnoreCase(str2);
```

```
            int compareresult = str1.compareTo(str3);

            String substringres = str1.substring(5,14);

            boolean containres = str1.contains(substringres);

            System.out.println("\nEquals Result : "+equalsresult);
            System.out.println("\nEqualsIgnoreCase Result :
"+equalsIgnoreCaseresult);
            System.out.println("\nCompare Result : "+compareresult);
            System.out.println("\nSubstring Result : "+substringres);
            System.out.println("\nContains Result : "+containres);

      }
}
```

Output:-



Conclusion:

This program effectively demonstrates string comparison and substring checking in Java using various methods, highlighting the importance of case sensitivity in string operations.

2. Program to encode characters to their Unicode representations and decode them back.

Aim:
This program takes a string input from the user, converts it to Unicode code points, encodes it into UTF-8 bytes, and then decodes it back to the original string.
If the user inputs "Hello!", the program might produce output like this:
Unicode code points for the input string:
Character: H Code point: 72
Character: e Code point: 101
Character: l Code point: 108
Character: l Code point: 108
Character: o Code point: 111
Character: ! Code point: 33

String in UTF-8 byte encoding:
Byte:72
Byte:101
Byte:108
Byte:108
Byte:111
Byte:33

Decoded string from UTF-8 bytes: Hello!

Objectives:

1. To illustrate how to retrieve Unicode code points for each character in a string.
2. To demonstrate the encoding of the string into UTF-8 bytes and decoding back to the original string.

Theory:

- **Unicode Code Points:** Each character in a string can be represented by a unique Unicode code point.

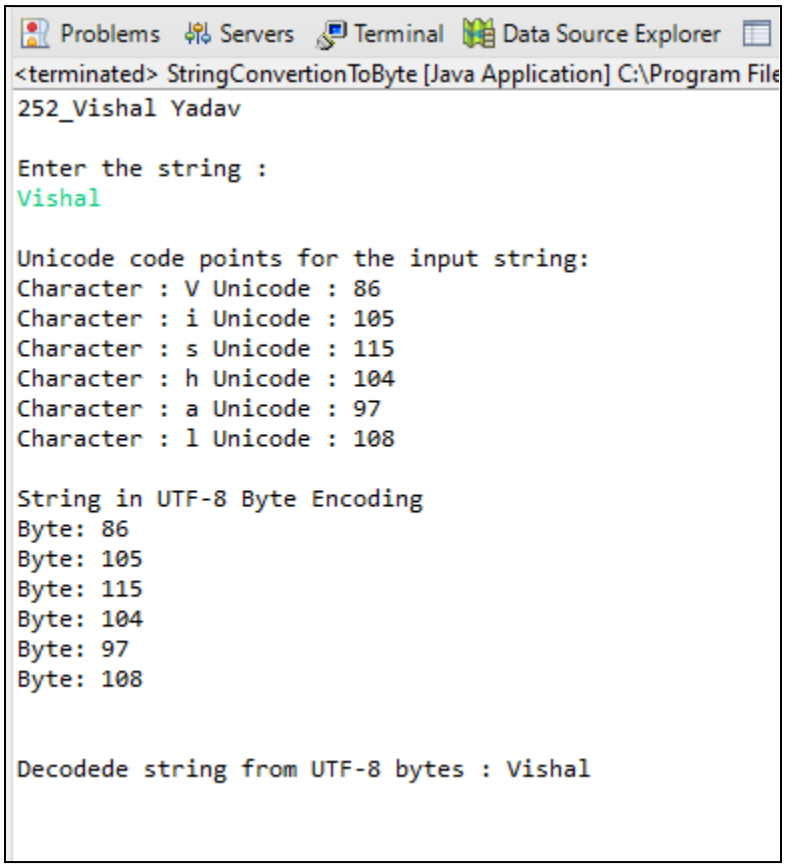
- **UTF-8 Encoding:** A variable-length character encoding that can represent every character in the Unicode character set.

Code:

```
package MyFirstPackage;
import java.nio.charset.StandardCharsets;
import java.util.*;

public class StringConvertionToByte {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		Scanner sc = new Scanner(System.in);

		System.out.println("\nEnter the string : ");
		String inputString  = sc.nextLine();

		System.out.println("\nUnicode code points for the input
string:");
		for(int i=0; i<inputString.length();i++) {
			int codePoint = inputString.codePointAt(i);
			System.out.printf("Character : "+inputString.charAt(i)+"
Unicode : "+codePoint+"\n");
		}

		byte[] utf8Bytes =
inputString.getBytes(StandardCharsets.UTF_8);
		System.out.println("\nString in UTF-8 Byte Encoding");
		for(byte b : utf8Bytes) {
			System.out.printf("Byte: "+b+"\n");
		}

		String decodedString = new String(utf8Bytes,
StandardCharsets.UTF_8);
```

```
            System.out.println("\n\nDecodede string from UTF-8 bytes :
"+decodedString);
            sc.close();
        }

}
```

Output:-



Conclusion:

The program successfully encodes a string to its Unicode representation and demonstrates how to decode it back to the original string, reinforcing the understanding of character encoding in Java.

Program 3 : Write a program that takes user input for multiple strings and appends them using StringBuilder.

Aim:
To collect multiple user-input strings and concatenate them into a single string using the **StringBuilder** class.

Objectives:

1. To show how to use **StringBuilder** for efficient string concatenation.
2. To implement a user-driven input mechanism that continues until a specific command is given.

Theory:

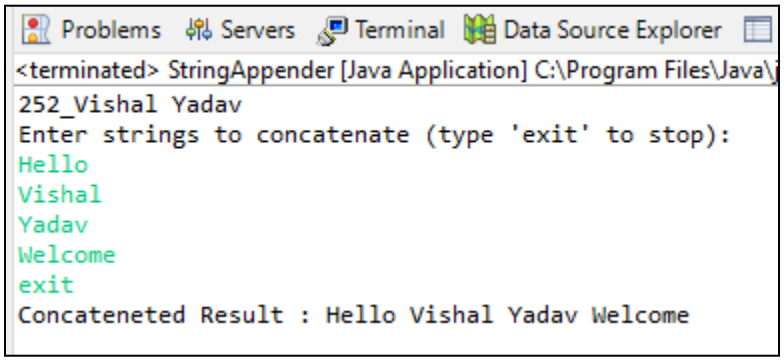
- **StringBuilder Class:** A mutable sequence of characters that provides an efficient way to manipulate strings compared to regular string concatenation using the + operator.

Code:

```
import java.util.Scanner;

package MyFirstPackage;
import java.util.*;

public class StringAppender {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		Scanner sc = new Scanner(System.in);
		StringBuilder result = new StringBuilder();

		System.out.println("Enter strings to concatenate (type 'exit' to
stop):");

		while(true) {
			String input = sc.nextLine();

			if(input.equalsIgnoreCase("exit")) {
				break;
```

```
                }
                result.append(input).append(" ");

            }

            System.out.println("Concateneted Result :
"+result.toString().trim());
            sc.close();
        }

}
```

Output:-



Conclusion:

This program demonstrates how to efficiently append multiple strings into a single string using **StringBuilder**, highlighting its usefulness in dynamic string manipulation.

Program 4 : Write a program to split a paragraph into individual sentences

Aim:
To split a given paragraph into separate sentences based on punctuation.

Objectives:

1. To utilize regular expressions for splitting a string into parts.
2. To clean up and format the output by trimming whitespace.

Theory:

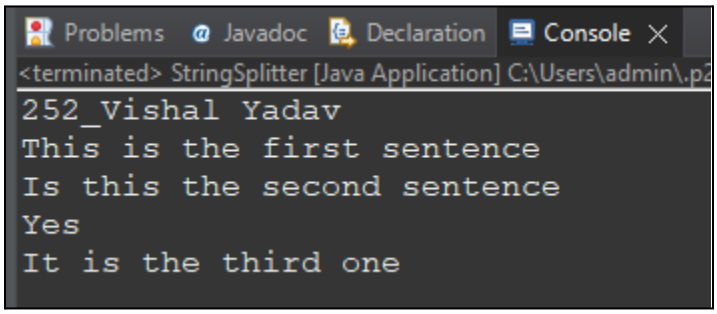
- **Regular Expressions:** A powerful tool for pattern matching and string manipulation. In this case, it helps identify sentence boundaries.

paragraph.split("[.?!]"):
This uses a regular expression to split the paragraph at every occurrence of a period (.), question mark (?), or exclamation mark (!).
sentence.trim():
Removes any extra spaces from the beginning or end of each sentence before printing.

Code:

```
package MyFirstPackage;

public class StringSplitter {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		String paragraph = "This is the first sentence. Is this the second sentence? Yes! It is the third one.";

		String[] sentences = paragraph.split("[.?!]");

		for(int i=0;i<sentences.length;i++) {
			System.out.println(sentences[i].trim());
		}
	}
}
```

Output:-



Conclusion:

The program effectively splits a paragraph into individual sentences, showcasing the use of regular expressions for string processing.

Program 5 : Write a program to convert a date object to a string in a specific format.

Aim:
To format a date object into a string representation using a specific date format.

Objectives:

1. To illustrate the use of SimpleDateFormat for date formatting.
2. To convert a Date object into a string in a user-defined format.

Theory:

- **SimpleDateFormat: A class that allows formatting and parsing of dates in a locale-sensitive manner.**

SimpleDateFormat:
This class is used to define the desired date format, like dd/MM/yyyy, yyyy-MM-dd, etc.

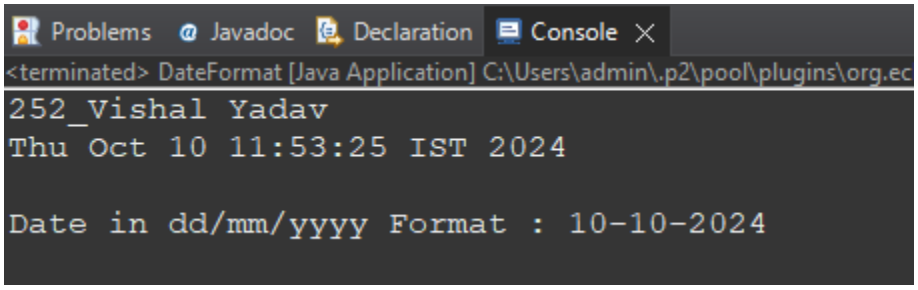
formatter.format(currentDate):
This method converts the Date object to a string according to the format specified by the SimpleDateFormat.

You can adjust the format string ("dd/MM/yyyy") to any other desired pattern, such as "yyyy-MM-dd HH:mm:ss" for a date-time format.

Code:

```
package MyFirstPackage;
import java.text.SimpleDateFormat;
import java.util.*;
public class DateFormat {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		Date currentDate = new Date();
		System.out.println(currentDate);

		SimpleDateFormat formatter = new
SimpleDateFormat("dd-MM-yyyy");

		String result = formatter.format(currentDate);
		System.out.println("\nDate in dd/mm/yyyy Format : "+result);

	}
}
```

Output:-



Conclusion:

This program successfully formats a **Date** object into a string representation, demonstrating the utility of **SimpleDateFormat** for date manipulation.

Program 6 : Write a program to insert a substring into a string at a specific position using StringBuilder.

Aim:
To demonstrate how to insert a substring into an existing string at a specified position using **StringBuilder**.

Objectives:

1. To show the use of the **insert()** method of **StringBuilder**.
2. To manipulate strings efficiently by utilizing **StringBuilder**.

Theory:

- **StringBuilder Insert Method:** Allows for easy insertion of characters or substrings at any index in the **StringBuilder** object.

StringBuilder:
The StringBuilder class is used for efficient string manipulation.

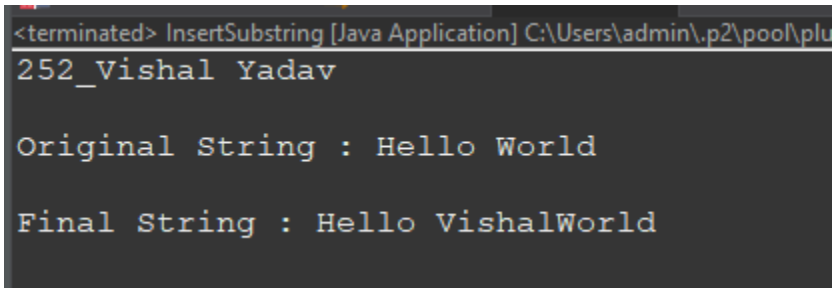
stringBuilder.insert(position, substring):
This method inserts the given substring into the StringBuilder object at the specified position (in this case, after "Hello ").

stringBuilder.toString():
Converts the StringBuilder object back to a regular String for output.

Code:

```
package MyFirstPackage;
import java.util.*;

public class InsertSubstring {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		StringBuilder sb = new StringBuilder();

		String str1 = "Hello World";
		sb.append(str1);
		System.out.println("\nOriginal String : "+str1);

		String res = (sb.insert(6, "Vishal")).toString();
		System.out.println("\nFinal String : "+res);

	}
}
```

Output:-



Conclusion:

The program effectively demonstrates how to insert a substring into a string at a specified position, showcasing the flexibility of **StringBuilder**.

Program 7: Write a program to remove null values from an array of strings

Aim:

To write a Java program that removes **null** values from an array of strings.

Objective:

- To learn how to handle arrays in Java.
- To understand the process of filtering out unwanted values (like **null**) from an array.
- To learn how to copy elements from one array to another based on conditions.

Theory:

In Java, arrays are fixed in size, and each element in an array can store values, including null. When we work with arrays that have null values, we might need to remove these null values for further processing.

The basic approach involves:

Counting the number of non-null values in the original array.
Creating a new array that will only store non-null values.
Copying the non-null values into the new array and then printing it.

This approach is commonly used when working with collections that may contain invalid or unnecessary entries, and it demonstrates the process of array filtering.

Step 1: The array Array contains some null values. First, we loop through the array to count how many values are non-null.

Step 2: Based on the count of non-null values, a new array resultArray is created. This array has the same size as the number of non-null elements in the original array.

Step 3: We copy all non-null elements from the original array Array to the resultArray.

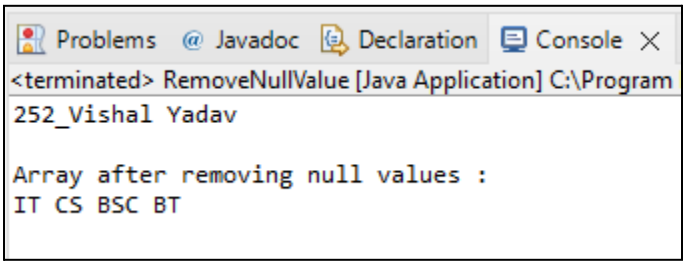
Step 4: The program prints the elements of resultArray, which contains only non-null values.

Code:
RemoveNullValue.java

```
package MyFirstPackage;

public class RemoveNullValue {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		String[] array = {"IT",null,"CS","BSC",null,"BT"};

		int count = 0;
		for(String s : array) {
			if(s != null) {
				count++;
			}
		}

		String[] resultArray = new String[count];
		int index = 0;

		for(String  s : array) {
			if(s != null) {
				resultArray[index++] = s;
			}
		}

		System.out.println("\nArray after removing null values : ");
		for(String s : resultArray) {
```

```
                System.out.print(s + " ");
            }

        }
}
```

Output:-



Conclusion:

This program efficiently removes null values from an array of strings

Program 8: Write a program to find all occurrences of a pattern in a string using Pattern and Matcher.

Aim:
To find and display all occurrences of a specific pattern within a given string using Java's Pattern and Matcher classes.

Objective:
- To learn how to use regular expressions in Java for pattern matching.
- To understand the functionality of the Pattern and Matcher classes.
- To implement a program that identifies all occurrences of a specified substring in a larger string and displays their positions.

Theory:
Java provides the java.util.regex package, which includes classes for pattern matching with regular expressions. The two primary classes are:
- **Pattern**: Represents a compiled regular expression. It allows you to define a pattern that can be reused for matching against different strings.
- **Matcher**: An engine that interprets the Pattern and performs matching operations on a target string. It provides methods like find(), start(), and end() to locate matches.

Key Concepts:
Regular Expressions: A sequence of characters that forms a search pattern. It can be used to search, match, and manipulate strings based on specific patterns.

- **Finding Matches:** The find() method in the Matcher class is used to search for occurrences of the pattern. Each match can be accessed using start() and end() methods to get the indices.

- **Pattern Compilation:**Pattern.compile(patternString) compiles the regular expression "ab" into a Pattern object.

Matcher:
pattern.matcher(inputString) creates a Matcher object that will search for the pattern in the input string "abcabcababcdab".

matcher.find():
This method finds the next occurrence of the pattern in the string and returns true if a match is found.

matcher.start() and matcher.end():

- start() returns the starting index of the match.
- end() returns the ending index of the match (exclusive).

Code:

```
package MyFirstPackage;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class PatternMatcher {

	public static void main(String[] args) {
		System.out.println("252 Vishal Yadav");

		String str = "abcabcababcdab";
		String ptstr = "ab";

		Pattern pattern = Pattern.compile(ptstr);

		Matcher matcher = pattern.matcher(str);

		System.out.println("\nString Pattern found at : ");
		while(matcher.find()) {
			System.out.println("Start : "+matcher.start()+" , "+"End :
"+matcher.end());
		}

	}
```

```
}
```

Output:-

```
<terminated> PatternMatcher [Java Application] C:\Program F
252 Vishal Yadav

String Pattern found at :
Start : 0 , End : 2
Start : 3 , End : 5
Start : 6 , End : 8
Start : 8 , End : 10
Start : 12 , End : 14
```

Conclusion:

The program successfully demonstrates how to utilize Java's regex capabilities to locate all occurrences of a specified pattern in a string. By compiling the pattern and using the matcher, we can effectively identify multiple matches and retrieve their positions. This technique is useful in various applications, such as text processing, data validation, and searching through strings for specific content. Understanding regex in Java empowers developers to perform complex string operations with ease and efficiency.

Program 9: Write a program to count the number of vowels in a string using the Character class

Aim:
To implement a linear search algorithm to find an element in an array.

Objectives:

1. To illustrate how to perform a search operation within an array.
2. To demonstrate the concept of linear search and its time complexity.

Theory:

A straightforward method that checks each element in the array until the desired element is found or the end of the array is reached.

Convert to Lowercase:
The input string is converted to lowercase to treat uppercase and lowercase vowels the same ("A" and "a" are both vowels).

Character.isLetter(char):
This method checks if the character is a letter (to avoid counting non-alphabetic characters).

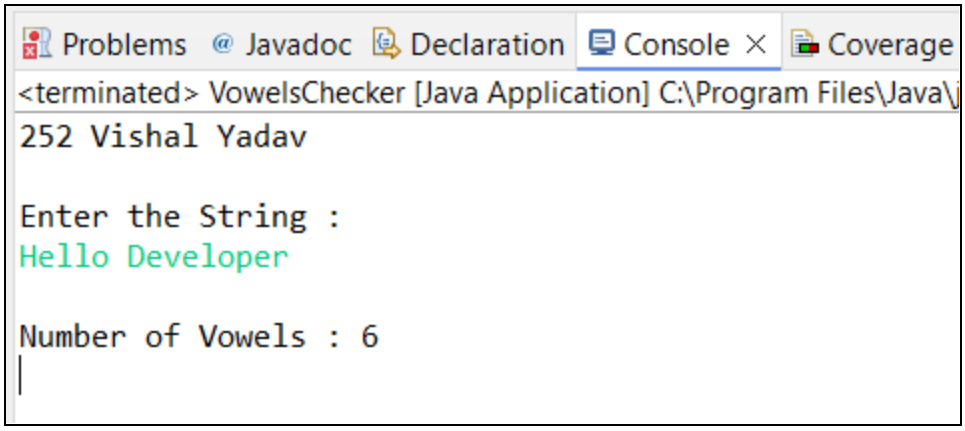
Custom isVowel(char) Method:
This helper method returns true if the character is one of the vowels ('a', 'e', 'i', 'o', 'u').

Code:

```
package MyFirstPackage;
import java.util.*;
public class VowelsChecker {

	public static boolean isVowels(char ch) {
		ch = Character.toLowerCase(ch);

		return ch == 'a' ||ch == 'e' ||ch == 'i' ||ch == 'o' ||ch == 'u';
```

```
    }

    public static int countvowels(String str) {
        int count = 0;

        for(int i=0;i<str.length();i++) {
            char ch = str.charAt(i);

            if(Character.isLetter(ch) && isVowels(ch)) {
                count ++;
            }
        }
        return count;
    }

    public static void main(String[] args) {
        System.out.println("252 Vishal Yadav");

        Scanner  sc =  new Scanner(System.in);
        System.out.println("\nEnter the String : ");

        String input = sc.nextLine();

        int vowelscount = countvowels(input);

        System.out.println("\nNumber of Vowels : "+vowelscount);
        sc.close();
    }
}
```

Output:-



Conclusion:

This program effectively implements a linear search algorithm to find an element in an array, demonstrating the basic principles of searching algorithms in programming.

Program 10 : Write a program to check whether a given string is a palindrome or not by using:
A.StringBuffer Class
B.String Class

Aim:
To count the number of vowels and consonants in a given string.

Objectives:

1. To illustrate character classification and counting techniques in Java.
2. To demonstrate string manipulation and character checking.

Theory:

Using StringBuffer:
The StringBuffer class is used to reverse the input string.
stringBuffer.reverse() reverses the string, and the original string is compared with the reversed string.
Using String Class:
A loop iterates through the input string in reverse order to construct the reversed string.
The original string is compared with the reversed string using equals().

Code:
A.Using StringBuffer Class

Code:-

```
package MyFirstPackage;

public class StringComparsion {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");

		StringBuffer sb = new StringBuffer("Vishal");
```

```
		System.out.println("\nOriginal String : "+sb);
		String reverseStr = sb.reverse().toString();

		System.out.println("Reverse String : "+reverseStr);

		boolean res = sb.equals(reverseStr);

		if(res == true) {
			System.out.println("\nOriginal String is same as Reversed
String");
		}else {
			System.out.println("\nOriginal String is not same as
Reversed String");
		}
	}
}
```

Output:-

```
252_Vishal Yadav

Original String : Vishal
Reverse String : lahsiV

Original String is not same as Reversed String
```

B.Using String Class

Code:-

```
package MyFirstPackage;

public class StringOperations {

	public static void main(String[] args) {
		System.out.println("252_Vishal Yadav");
```

```
                String str1 = "Vishal";
                String nstr="";
                char ch;

                System.out.println("\nOriginal String : "+str1);
                for(int i=0;i<str1.length();i++) {
                        ch = str1.charAt(i);
                        nstr = ch+nstr;
                }

                System.out.print("Reversed String : "+nstr);

                boolean res = str1.equals(nstr);

                if(res == true) {
                        System.out.println("\nOriginal String is same as
Reversed String");
                }else {
                        System.out.println("\nOriginal String is not same as
Reversed String");
                }
        }

}
```

Output:-

```
252_Vishal Yadav

Original String : Vishal
Reversed String : lahsiV
Original String is not same as Reversed String
```

```
252_Vishal Yadav

Original String : Hello
Reversed String : olleH
Original String is not same as Reversed String
```

Conclusion:

The program successfully counts the number of vowels and consonants in a given string, showcasing string manipulation and character processing techniques in Java.