PW SKILLS

Lesson Plan

Strings 1

PW SKILLS

What are strings and Why are they used?

In programming, a string is a sequence of characters, typically used to represent text. It is one of the fundamental data types in most programming languages. A string can consist of letters, numbers, symbols, or a combination of these.

Strings are used for various purposes in programming:

- **Text Representation:** Strings are commonly used to represent and manipulate text data. For example, when working with user inputs, file contents, or messages, you often use strings to store and process these textual elements.
- **Data Manipulation:** Strings provide various methods for manipulating and processing textual data. Operations such as concatenation, splitting, searching, and replacing are frequently performed on strings.
- **Output and Input:** Strings are often used to display information to users through console outputs or graphical user interfaces (GUIs). They also play a crucial role in reading and writing data to and from files.
- **Communication:** Strings are commonly used for communication between different parts of a program or between different programs. For example, when working with web development, data is often exchanged in the form of strings through APIs.
- **Representation of Data:** Strings can be used to represent data in a structured or serialized format, such as JSON or XML. This is common when dealing with data exchange between systems.

In many programming languages, strings are immutable, meaning that once a string is created, it cannot be changed. Any operation that appears to modify a string actually creates a new string with the desired modifications. This immutability ensures consistency and can help prevent unintended side effects

```
public class StringExample {
    public static void main(String[] args) {
        // Creating strings
        String greeting = "Hello, ";
        String name = "John";

        // Concatenation
        String message = greeting + name;

        // Displaying the result
        System.out.println(message); // Output: Hello, John

        // String length
        int length = message.length();
        System.out.println("Length of the message: " +
length); // Output: 12
```

```
        // Accessing characters
        char firstChar = message.charAt(0);
        System.out.println("First character: " + firstChar);
// Output: H

        // Substring
        String substring = message.substring(7);
        System.out.println("Substring from index 7: " +
substring); // Output: John

        // String comparison
        String anotherName = "John";
        boolean isEqual = name.equals(anotherName);
        System.out.println("Names are equal: " + isEqual);
// Output: true
    }
}
```

In this example, we create strings, perform concatenation, find the length of a string, access individual characters, extract substrings, and compare strings. The String class in Java provides various methods for working with strings, allowing for common operations and manipulations.

Declaration of Strings and taking Input

In Java, you can declare strings and take input using the String class and the Scanner class for user input. Here's an example:

```
import java.util.Scanner;

public class StringInputExample {
    public static void main(String[] args) {
        // Declaration and initialization of strings
        String greeting = "Hello, World!";
        String userInput;

        // Displaying the initialized string
        System.out.println(greeting);

        // Taking input from the user
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a message: ");
        userInput = scanner.nextLine();

        // Displaying the user-input string
        System.out.println("You entered: " + userInput);
```

```
        // Closing the scanner
        scanner.close();
    }
}
```

In this example:

1. greeting is a string that is already initialized with a value.
2. userInput is a string where we'll store the user's input.
3. We use the Scanner class to read input from the user. scanner.nextLine() reads the entire line entered by the user.

Note: Make sure to import java.util.Scanner at the beginning of your Java file.
Compile and run this Java program, and it will prompt you to enter a message. After entering the message, it will display the entered string.

Indexing of characters in Strings

In Java, indexing of characters in strings starts from 0. Each character in a string is associated with an index, and you can access individual characters using their index. Here's an example:

```
public class StringIndexingExample {
    public static void main(String[] args) {
        // Declare and initialize a string
        String message = "Hello, Java!";

        // Accessing characters by index
        char firstChar = message.charAt(0); // Index 0
        char fifthChar = message.charAt(4); // Index 4
        char lastChar = message.charAt(message.length() -
1); // Last index

        // Displaying the characters
        System.out.println("First character: " + firstChar);
  // Output: H
        System.out.println("Fifth character: " + fifthChar);
  // Output: o
        System.out.println("Last character: " + lastChar);
  // Output: !

        // Iterating through all characters
        System.out.println("All characters:");
        for (int i = 0; i < message.length(); i++) {
            char currentChar = message.charAt(i);
            System.out.println("Index " + i + ": " +
currentChar);
        }
    }
}
```

In this example:

- The charAt(index) method is used to retrieve the character at a specific index.
- message.charAt(0) retrieves the first character.
- message.charAt(4) retrieves the character at index 4.
- message.charAt(message.length() - 1) retrieves the last character, regardless of the length of the string.
- The loop iterates through all characters in the string, displaying each character along with its index.

Remember that the index is zero-based, so the first character is at index 0, the second character is at index 1, and so on

Q. Input a string of length n and count all the vowels in the given string.

In Java, indexing of characters in strings starts from 0. Each character in a string is associated with an index, and you can access individual characters using their index. Here's an example:

```
import java.util.Scanner;

public class CountVowels {
    public static void main(String[] args) {
        // Create a Scanner object for user input
        Scanner scanner = new Scanner(System.in);

        // Input a string
        System.out.print("Enter a string: ");
        String inputString = scanner.nextLine();

        // Close the Scanner to prevent resource leak
        scanner.close();

        // Convert the string to lowercase to make the
counting case-insensitive
        inputString = inputString.toLowerCase();

        // Count the vowels
        int vowelCount = 0;
        for (int i = 0; i < inputString.length(); i++) {
            char currentChar = inputString.charAt(i);
            if (currentChar == 'a' || currentChar == 'e' ||
currentChar == 'i' ||
                currentChar == 'o' || currentChar == 'u') {
                vowelCount++;
            }
        }
```

```
        // Display the result
        System.out.println("Number of vowels in the string:
" + vowelCount);
    }
}
```

In this program:

- We use the Scanner class to take user input for a string.
- The string is converted to lowercase using toLowerCase() to make the counting case-insensitive.
- A loop iterates through each character in the string, and if the character is a vowel ('a', 'e', 'i', 'o', or 'u'), the vowelCount is incremented.
- Finally, the program prints the total count of vowels in the given string.

String immutability in java
In Java, strings are immutable, which means once a string object is created, its value cannot be changed. Here are some key points about string immutability in Java:

Creation of String Objects:

- Strings can be created using the String class constructor or string literals.
- Example with a string literal: String greeting = "Hello";
- Example with a constructor: String greeting = new String("Hello");

Concatenation:

- String concatenation using the + operator or the concat() method creates a new string object.
- Example:

String str1 = "Hello";
String str2 = "World";
String result = str1 + " " + str2; // Creates a new string object

Modification:

- Any operation that appears to modify a string actually creates a new string with the desired modifications.
- Methods like substring(), toUpperCase(), toLowerCase(), etc., return new string objects.
- Example:

String original = "Hello";
String modified = original.substring(1); // Creates a new string object

Memory Efficiency:

- Because strings are immutable, they can be more memory-efficient. String literals are stored in a string pool, and if the same literal is used elsewhere, the same object is referenced.

Thread Safety:

- String immutability contributes to thread safety. Once a string is created, its value cannot be changed, making it safe for concurrent access.

StringBuilder and StringBuffer:

- For scenarios where dynamic string modification is required, Java provides StringBuilder and StringBuffer classes. These classes are mutable and designed for efficient string manipulation.

Here's a brief example illustrating string immutability:

```
String original = "Hello";
String modified = original.substring(1); // Creates a new string object

System.out.println(original); // Output: Hello
System.out.println(modified); // Output: ello
```

In this example, even though substring() appears to modify the string, it returns a new string object, leaving the original string unchanged.
CharSequence Interface

StringBuffer

In Java, StringBuffer is a class that provides mutable sequences of characters. It is part of the java.lang package and is designed for situations where you need to perform multiple modifications on a string. Unlike the immutable String class, the StringBuffer class allows you to modify the contents of the string without creating a new object.
Here are some key features of the StringBuffer class:

Mutable Operations:

- StringBuffer provides methods for appending, inserting, deleting, and replacing characters within the string.

Thread-Safe:

- Unlike StringBuilder, StringBuffer is thread-safe. This means that its methods are synchronized, making it safe to use in a multi-threaded environment.

Performance Considerations:

- While StringBuffer is useful when thread safety is required, it may have a performance overhead due to synchronization. In cases where thread safety is not a concern, StringBuilder is recommended for better performance.

Initialization:

- You can create a StringBuffer object using its default constructor or by passing an initial capacity as an argument.
- Example:

```
StringBuffer buffer1 = new StringBuffer();         // Default constructor
StringBuffer buffer2 = new StringBuffer("Hello");  // With initial content
StringBuffer buffer3 = new StringBuffer(20);       // With initial capacity
```

Methods:

- Some common methods provided by StringBuffer include append(), insert(), delete(), replace(), and more.

Here's a simple example demonstrating the use of StringBuffer:

```
public class StringBufferExample {
    public static void main(String[] args) {
        // Create a StringBuffer with initial content
        StringBuffer buffer = new StringBuffer("Hello");

        // Append additional content
        buffer.append(" World");

        // Insert content at a specific position
        buffer.insert(5, ","); // Inserting a comma after
"Hello"

        // Display the modified content
        System.out.println(buffer.toString()); // Output:
Hello, World
    }
}
```

In this example, the StringBuffer is used to modify the content of the string by appending and inserting characters. The result is a mutable string that can be modified in place.

Stringbuilder

In Java, StringBuilder is a class that provides mutable sequences of characters. It is part of the java.lang package and is similar to StringBuffer in terms of providing a mutable alternative to the immutable String class. The primary difference between StringBuilder and StringBuffer is that StringBuilder is not thread-safe, making it more efficient in a single-threaded context.
Here are some key features of the StringBuilder class:

Mutable Operations:

- StringBuilder provides methods for appending, inserting, deleting, and replacing characters within the string.

Not Thread-Safe:

- Unlike StringBuffer, StringBuilder is not synchronized. This lack of synchronization makes it more efficient in a single-threaded environment but requires caution in multi-threaded scenarios.

Performance Considerations:

- StringBuilder is recommended when thread safety is not a concern. It generally has better performance than StringBuffer due to the absence of synchronization.

Initialization:

- You can create a StringBuilder object using its default constructor or by passing an initial capacity as an argument.
- Example:

```
StringBuilder builder1 = new StringBuilder();         // Default constructor
StringBuilder builder2 = new StringBuilder("Hello");   // With initial content
StringBuilder builder3 = new StringBuilder(20);        // With initial capacity
```

Methods:

- Some common methods provided by StringBuilder include append(), insert(), delete(), replace(), and more.

Here's a simple example demonstrating the use of StringBuilder:

```
public class StringBuilderExample {
    public static void main(String[] args) {
        // Create a StringBuilder with initial content
        StringBuilder builder = new StringBuilder("Hello");

        // Append additional content
        builder.append(" World");

        // Insert content at a specific position
        builder.insert(5, ","); // Inserting a comma after
"Hello"

        // Display the modified content
        System.out.println(builder.toString()); // Output:
Hello, World
    }
}
```

In this example, the StringBuilder is used to modify the content of the string by appending and inserting characters. The result is a mutable string that can be modified in place.

String vs StringBuffer vs Stringbuilder

String, StringBuffer, and StringBuilder are three classes in Java that are used for handling strings, but they have different characteristics and use cases. Here's a comparison of these classes:

1. String:

- **Immutability:**
  - Strings in Java are immutable, meaning their values cannot be changed after they are created.
  - Any operation that appears to modify a string creates a new string object.
- **Thread Safety:**
  - Strings are inherently thread-safe because they cannot be modified. Multiple threads can safely read the same string without issues.
- **Performance Implications:**
  - Because of immutability, string concatenation using the + operator can be inefficient when performed in a loop, as it creates multiple string objects.

2. StringBuffer:

- **Mutability:**
  - StringBuffer is mutable, meaning you can modify its contents without creating a new object.
- **Thread Safety:**
  - StringBuffer is thread-safe because its methods are synchronized. This makes it suitable for use in multithreaded environments.
- **Performance Implications:**
  - The synchronization in StringBuffer can introduce a performance overhead, especially in single-threaded scenarios. It is generally less efficient than StringBuilder.

3. StringBuilder:

- **Mutability:**
  - StringBuilder is also mutable, like StringBuffer, allowing for modifications without creating new objects.
- **Thread Safety:**
  - StringBuilder is not thread-safe. It lacks the synchronization present in StringBuffer.

- **Performance Implications:**
  - StringBuilder is recommended in single-threaded scenarios because it lacks the synchronization overhead of StringBuffer. It is generally more efficient than StringBuffer.

Use Cases:

- Use String when:
  - Immutability is desired.
  - Thread safety is not a concern.
  - String manipulation is minimal.
- Use StringBuffer when:
  - Immutability is not required.
  - Thread safety is a concern, especially in multithreaded environments.
  - Moderate string manipulation is needed.
- Use StringBuilder when:
  - Immutability is not required.
  - Thread safety is not a concern (single-threaded scenario).
  - Maximum performance is desired, and extensive string manipulation is needed.

Summary:

- Use String for immutable strings and situations where modification is not required.
- Use StringBuffer for thread-safe mutable strings in multithreaded environments.
- Use StringBuilder for mutable strings in single-threaded scenarios where maximum performance is desired.

Input a string of size n and Update all the even positions in the string to character 'a'. Consider 0-based indexing.

```
import java.util.Scanner;

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

        // Input a string
        System.out.print("Enter a string: ");
        String inputString = scanner.nextLine();

        // Update even positions to 'a'
        String updatedString =
updateEvenPositions(inputString);
```

```
import java.util.Scanner;

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

        // Input a string
        System.out.print("Enter a string: ");
        String inputString = scanner.nextLine();

        // Update even positions to 'a'
        String updatedString =
updateEvenPositions(inputString);

        // Display the updated string
        System.out.println("Updated String: " +
updatedString);
    }

    // Function to update even positions in the string to
'a'
    private static String updateEvenPositions(String input)
{
        char[] charArray = input.toCharArray();

        // Iterate through the characters and update even
positions
        for (int i = 0; i < charArray.length; i += 2) {
            charArray[i] = 'a';
        }

        // Convert the char array back to a string
        return new String(charArray);
    }
}
```

This program takes a string as input, updates the characters at even positions to 'a', and then prints the updated string. The function updateEvenPositions uses a char array to perform the updates efficiently.

For example, if you input the string "hello world," the output would be "aelao walad." The characters at even positions (0, 2, 4, 6, 8, ...) have been replaced with 'a'.

Strings vs Character array

Both strings and character arrays are used to represent sequences of characters in Java, but they have different characteristics and use cases. Here's a comparison:

Strings:

Immutability:

- Strings in Java are immutable, meaning their values cannot be changed after they are created.
- Any operation that appears to modify a string creates a new string object.

API and Methods:

- Strings have a rich set of built-in methods for string manipulation, searching, and comparison.
- Examples: charAt(), length(), substring(), concat(), etc.

Ease of Use:

- Strings are convenient to use when you need to perform various string operations without managing the underlying storage.

String Pool:

- String literals are stored in a special memory area called the "string pool," which can lead to more efficient memory usage.

Character Arrays:

Mutability:

- Character arrays (char[]) are mutable, meaning you can modify their individual elements directly.

API and Methods:

- Character arrays have fewer built-in methods compared to strings.
- Examples: Iteration and direct assignment are common operations.

Performance:

- Manipulating characters directly in a character array can be more performant than working with strings, especially when extensive modifications are needed.

Storage:

- Character arrays are more flexible in terms of managing storage. You can resize, modify, and manipulate the array directly.

Use Cases:

Use Strings when:

- Immutability is desired.
- You need to perform various string manipulations.
- Memory efficiency and ease of use are more critical than direct manipulation.

Use Character Arrays when:

- You need mutability and direct control over individual characters.
- Performance is critical, and you have specific memory management requirements.
- You want to work with lower-level operations for efficient character manipulation.

Example:

Here's a simple example to illustrate the difference:

```
// Using String
String str = "Hello";
str = str.concat(" World");

// Using Character Array
char[] charArray = {'H', 'e', 'l', 'l', 'o'};
charArray[4] = 'a'; // Direct modification

// Convert char array to String
String modifiedStr = new String(charArray);
```

In summary, choose between strings and character arrays based on your specific needs. If you need immutability, a rich set of methods, and ease of use, go for strings. If you require mutability, direct control over characters, and better performance in certain scenarios, consider using character arrays.

Built-in stringBuilder functions

capacity() & charAt()

In Java's StringBuilder class:

capacity():

- The capacity() method is used to get the current capacity of the StringBuilder. The capacity is the maximum number of characters the StringBuilder can hold without reallocating its internal buffer.

- The capacity is always greater than or equal to the length of the StringBuilder.

StringBuilder sb = new StringBuilder("Hello");
int capacity = sb.capacity();
System.out.println("Capacity: " + capacity);

charAt(int index):

- The charAt(int index) method is used to get the character at the specified index within the StringBuilder.

- The index is zero-based, meaning the first character is at index 0.

```
StringBuilder sb = new StringBuilder("Hello");
char character = sb.charAt(2);  // Gets the character at index 2 (third character)
System.out.println("Character at index 2: " + character);
```

These methods provide useful functionalities for inspecting the state of a StringBuilder. The capacity() method is particularly useful when you want to know the current capacity of the StringBuilder object. The charAt(int index) method helps in retrieving specific characters at a given position within the StringBuilder.

chars() & delete(int start,int end)

In Java's StringBuilder class:

chars():

- The chars() method returns an IntStream of the characters from the StringBuilder. This method is part of the CharSequence interface and allows you to perform various operations on the characters using the Stream API.

```
StringBuilder sb = new StringBuilder("Hello");
sb.chars().forEach(System.out::println);
```

delete(int start, int end):

- The delete(int start, int end) method is used to remove characters from the StringBuilder between the specified start (inclusive) and end (exclusive) indices

```
StringBuilder sb = new StringBuilder("Hello World");
sb.delete(6, 11); // Removes characters from index 6 to 10 (11 - 1)
```

- After the above operation, sb will contain the string "Hello".

These methods provide additional functionality for working with characters in a StringBuilder. The chars() method allows you to process the characters using streams, and the delete(int start, int end) method is useful for removing a range of characters from the StringBuilder.
deleteCharAt() & indexOf()

In Java's StringBuilder class:

deleteCharAt(int index):

- The deleteCharAt(int index) method is used to remove the character at the specified index from the StringBuilder.

```
StringBuilder sb = new StringBuilder("Hello");
sb.deleteCharAt(2); // Removes the character at index 2
```

After the above operation, sb will contain the string "Helo".

indexOf(String str):

- The indexOf(String str) method is used to find the index of the first occurrence of the specified substring (str) in the StringBuilder. If the substring is not found, it returns -1.

```
StringBuilder sb = new StringBuilder("Hello World");
int index = sb.indexOf("World"); // Returns the index of the substring "World"
```

After the above operation, index will be set to 6.
These methods provide useful functionalities for manipulating characters in a StringBuilder. The deleteCharAt(int index) method allows you to remove a specific character at a given index, and the indexOf(String str) method helps in finding the index of a substring within the StringBuilder.

length () & setCharAt()

In Java's StringBuilder class:

length():

- The length() method is used to get the number of characters currently present in the StringBuilder. It returns the length of the character sequence.

```
StringBuilder sb = new StringBuilder("Hello");
int length = sb.length();
```

After the above operation, length will be set to 5.

setCharAt(int index, char ch):

- The setCharAt(int index, char ch) method is used to set the character at the specified index to the given character (ch).

```
StringBuilder sb = new StringBuilder("Hello");
sb.setCharAt(1, 'a'); // Sets the character at index 1 to 'a'
```

After the above operation, sb will contain the string "Hallo".
These methods provide functionalities for inspecting and modifying the content of a StringBuilder. The length() method helps in determining the current length of the StringBuilder, and the setCharAt(int index, char ch) method allows you to replace a character at a specific index.

append()

The append() method in Java's StringBuilder class is used to append various types of data to the end of the current StringBuilder object. It has multiple overloaded versions to accept different types of data.

Here are a few examples:

Appending a String:

StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");

After the above operation, sb will contain the string "Hello World".
Appending a Character:

StringBuilder sb = new StringBuilder("Hello");
sb.append('!'); // Appends the character '!'

After the above operation, sb will contain the string "Hello!".
Appending an Integer:

StringBuilder sb = new StringBuilder("Value: ");
sb.append(42); // Appends the integer 42 as a string

reverse()

The reverse() method in Java's StringBuilder class is used to reverse the characters of the sequence currently held by the StringBuilder object. This method modifies the existing StringBuilder object and does not create a new one.

Here's an example:

StringBuilder sb = new StringBuilder("Hello");
sb.reverse();

After the above operation, the content of sb will be reversed, and it will contain the string "olleH".
The reverse() method is useful when you need to reverse the order of characters in a StringBuilder, providing an efficient way to manipulate the contents of the string.

Q. Input a string of even length and reverse the first half of the string.

```
import java.util.Scanner;

public class ReverseFirstHalf {
    public static void main(String[] args) {
        // Input a string of even length
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a string of even length: ");
        String inputString = scanner.nextLine();
        scanner.close();
```

```
        // Check if the length is even
        if (inputString.length() % 2 == 0) {
            // Reverse the first half of the string
            String reversedFirstHalf =
reverseFirstHalf(inputString);

            // Display the result
            System.out.println("Original String: " +
inputString);
            System.out.println("Reversed First Half: " +
reversedFirstHalf);
        } else {
            System.out.println("Please enter a string of
even length.");
        }
    }

    private static String reverseFirstHalf(String str) {
        int length = str.length();
        int halfLength = length / 2;

        // Reverse the first half using StringBuilder
        StringBuilder reversedFirstHalf = new
StringBuilder(str.substring(0, halfLength)).reverse();

        // Concatenate the reversed first half with the
second half
        return reversedFirstHalf +
str.substring(halfLength);
    }
}
```

This program takes a string of even length as input, checks if the length is even, and then reverses the first half of the string using the reverseFirstHalf method. The reversed string is then displayed.

Q. Input a string of length greater than 5 and reverse the substring from position 2 to 5 using inbuilt functions.

```
import java.util.Scanner;

public class ReverseSubstring {
    public static void main(String[] args) {
        // Input a string of length greater than 5
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a string of length greater
than 5: ");
        String inputString = scanner.nextLine();
        scanner.close();
```

```
        // Check if the length is greater than 5
        if (inputString.length() > 5) {
            // Reverse the substring from position 2 to 5
            String reversedSubstring =
reverseSubstring(inputString, 2, 5);

            // Display the result
            System.out.println("Original String: " +
inputString);
            System.out.println("Reversed Substring: " +
reversedSubstring);
        } else {
            System.out.println("Please enter a string of
length greater than 5.");
        }
    }

    private static String reverseSubstring(String str, int
start, int end) {
        // Extract the substring from position 2 to 5
        String substringToReverse = str.substring(start, end
+ 1);

        // Reverse the substring using StringBuilder
        StringBuilder reversedSubstring = new
StringBuilder(substringToReverse).reverse();

        // Replace the reversed substring in the original
string
        return str.substring(0, start) + reversedSubstring +
str.substring(end + 1);
    }
}
```

This program takes a string of length greater than 5 as input, checks if the length is greater than 5, and then reverses the substring from position 2 to 5 using the reverseSubstring method. The reversed substring is then displayed.

substring()

In Java, the substring() method is used to extract a portion of a string. It has two overloaded versions:

substring(int beginIndex):

- Returns a new string that is a substring of the original string, starting from the specified beginIndex to the end of the string.

String originalString = "Hello, World!";
String substring = originalString.substring(7); // "World!"

substring(int beginIndex, int endIndex):

- Returns a new string that is a substring of the original string, starting from beginIndex and ending at endIndex - 1.

String originalString = "Hello, World!";
String substring = originalString.substring(7, 12); // "World"

In both cases, the original string remains unchanged, and a new string is returned containing the specified portion.
Keep in mind that the indices are zero-based, and the endIndex is exclusive, meaning the character at the endIndex is not included in the result. If endIndex is greater than or equal to the length of the string, it is treated as the length of the string, and no exception is thrown.
Note: The substring() method returns a new string and doesn't modify the original string. If you want to modify the original string, you need to assign the result back to the original string variable.

Q. Input a string of even length and return the second half of that string using inbuilt substr function

```
import java.util.Scanner;

public class SecondHalfOfString {
    public static void main(String[] args) {
        // Input a string of even length
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a string of even length: ");
        String inputString = scanner.nextLine();
        scanner.close();

        // Check if the length is even
        if (inputString.length() % 2 == 0) {
            // Return the second half of the string
            String secondHalf = getSecondHalf(inputString);

            // Display the result
            System.out.println("Original String: " +
inputString);
            System.out.println("Second Half: " +
secondHalf);
        } else {
            System.out.println("Please enter a string of
even length.");
        }
    }

    private static String getSecondHalf(String str) {
        // Calculate the midpoint of the string
        int midpoint = str.length() / 2;
```

```
        // Return the second half using substring
        return str.substring(midpoint);
    }
}
```

In this program, the getSecondHalf method takes a string as input, calculates the midpoint of the string, and then uses the substring function to return the second half of the string. The program ensures that the input string has an even length before proceeding.

toString()

In Java, the toString() method is a method of the Object class that is overridden by many other classes. Its primary purpose is to return a string representation of the object. The default implementation provided by the Object class returns a string consisting of the class name followed by the "@" character and the object's hashcode.
Classes often override the toString() method to provide a more meaningful representation of their objects. For example, if you have a custom class, you might override toString() to return a string that includes important information about the object's state.

Here's an example of a custom class overriding the toString() method:

```
public class Person {
    private String name;
    private int age;

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

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

    public static void main(String[] args) {
        // Create an instance of the Person class
        Person person = new Person("John", 30);

        // Call toString() implicitly or explicitly
        System.out.println(person); // Calls toString()
implicitly
        System.out.println(person.toString()); // Calls
toString() explicitly
    }
}
```

In this example, the Person class overrides toString() to provide a string representation of a Person object with the person's name and age. When you print a Person object, the overridden toString() method is called implicitly.
Keep in mind that when you concatenate an object with a string using the + operator, the toString() method is automatically invoked.

Q. Return the total number of digits in a number without using any loop.

Hint : Try using inbuilt to_string() function.

```
public class CountDigitsInNumber {
    public static void main(String[] args) {
        // Example number
        int number = 12345;

        // Convert the number to a string
        String numberAsString = Integer.toString(number);

        // Find the length of the string (number of digits)
        int numberOfDigits = numberAsString.length();

        // Display the result
        System.out.println("Number: " + number);
        System.out.println("Number of Digits: " +
numberOfDigits);
    }
}
```

In this example, the Integer.toString(number) converts the integer number to its string representation, and then numberAsString.length() returns the length of the string, which is the total number of digits in the original number.
Remember that this approach involves using the toString() function, which is not a loop, as requested in the hint.

PW SKILLS

THANK YOU !

PW SKILLS