Unit 6: Handling Strings in Java

Strings are one of the most commonly used data types in Java. They are used to represent a sequence of characters. Java provides a rich set of methods to handle strings, making it easy to manipulate and process text data.

1. Creation of a String

In Java, strings can be created in two ways:

- Using **String Literals**
- Using the **`new`** keyword

Lab 1: Creation of a String

```
public class StringCreation {
    public static void main(String[] args) {
        // Using String Literal
        String str1 = "Hello, World!";

        // Using new Keyword
        String str2 = new String("Hello, Java!");

        System.out.println("String 1: " + str1);
        System.out.println("String 2: " + str2);
    }
}
```

Explanation:

- **String Literal**: When we create a string using double quotes (" "), Java automatically creates a String object and stores it in the String Pool (a special memory area for strings). If the string already exists in the pool, Java reuses it instead of creating a new object.
- **`new` Keyword**: When we create a string using the new keyword, Java creates a new String object in the heap memory, even if the same string already exists in the String Pool.

Output:

```
String 1: Hello, World!
String 2: Hello, Java!
```

Key Differences Between String Literal and `new` Keyword

Aspect	String Literal	`new` Keyword
Memory Allocation	Stored in the **String Pool**	Stored in the **Heap Memory**
Object Reuse	Reuses existing objects in the String Pool	Always creates a new object
Performance	More memory-efficient	Less memory-efficient
Use Case	Preferred for fixed strings	Used when dynamic string

		creation is needed

Notes: The **String Pool** is a special memory region in the Java Virtual Machine (JVM) that is used to store string literals and immutable string objects. It is part of the JVM's heap memory but is managed separately to optimize memory usage and improve performance when working with strings.

Why is the String Pool Important?

Strings are one of the most commonly used data types in programming, and they are often created repeatedly in applications. **To avoid creating multiple copies of the same string in memory, Java uses the String Pool as a mechanism to reuse existing string objects whenever possible.** This approach saves memory and enhances efficiency.

Heap memory is a region of the computer's memory used for dynamic memory allocation during program execution. It is one of the key components of memory management in programming languages like Java, C++, Python, and others. Unlike stack memory (which is used for static memory allocation), heap memory is flexible and allows programs to allocate and deallocate memory dynamically as needed. **How Heap Memory Works** Object Creation :

- When an object is created (e.g., String s = new String("Hello"); in Java), it is stored in the heap memory .
- A reference to the object (e.g., s) is stored in the stack or registers, pointing to the actual object in the heap.

Stack memory is used for:

- Storing primitive variables (e.g., int, float, boolean).
- Storing references to objects (the actual objects are stored in the heap).
- Managing method calls and local variables.

In String s = new String("Hello");:

- "Hello" is stored in the heap .
- s (the reference to "Hello") is stored in the stack .

Lab 2: Creation of a String

```
public class StringCreationExample {
    public static void main(String[] args) {
        // Using String Literal
        String str1 = "Hello, World!";
        String str2 = "Hello, World!"; // Reuses the same object from the String Pool

        // Using new Keyword
        String str3 = new String("Hello, World!"); // Creates a new object in heap
memory

        System.out.println("String 1: " + str1);
        System.out.println("String 2: " + str2);
        System.out.println("String 3: " + str3);
```

```
        // Check if str1 and str2 point to the same object
        System.out.println("Are str1 and str2 the same object? " + (str1 == str2));

        // Check if str1 and str3 point to the same object
        System.out.println("Are str1 and str3 the same object? " + (str1 == str3));
    }
}
```

Output:

```
String 1: Hello, World!
String 2: Hello, World!
String 3: Hello, World!
Are str1 and str2 the same object? true
Are str1 and str3 the same object? false
```

---

2. Concatenation of Strings

String concatenation is the process of combining two or more strings. In Java, we can concatenate strings using the `+` operator or the `concat()` method.

Lab 3: Concatenation of Strings

```
public class StringConcatenation {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";

        // Using + operator
        String result1 = str1 + " " + str2;

        // Using concat() method
        String result2 = str1.concat(" ").concat(str2);

        System.out.println("Result 1: " + result1);
        System.out.println("Result 2: " + result2);
    }
}
```

Explanation:

- The `+` operator is overloaded in Java to concatenate strings.
- The `concat()` method appends the specified string to the end of the current string.

Output:

```
Result 1: Hello World
Result 2: Hello World
```

---

3. Conversion of a String

Strings can be converted to other data types (e.g., `int`, `double`) and vice versa using wrapper classes.

Lab 4: Conversion of a String

```
public class StringConversion {
    public static void main(String[] args) {
        // String to int
        String str1 = "123";
        int num1 = Integer.parseInt(str1);

        // int to String
        int num2 = 456;
        String str2 = Integer.toString(num2);

        System.out.println("String to int: " + num1);
        System.out.println("int to String: " + str2);
    }
}
```

Explanation:

- `Integer.parseInt()` converts a string to an integer.
- `Integer.toString()` converts an integer to a string.

Output:

```
String to int: 123
int to String: 456
```

Summary Table of String Conversion

Conversion Type	Method Used
String → int	`Integer.parseInt(str)`
int → String	`Integer.toString(num)`, `String.valueOf(num)`
String → double	`Double.parseDouble(str)`
double → String	`Double.toString(num)`
String → boolean	`Boolean.parseBoolean(str)`
boolean → String	`Boolean.toString(boolValue)`
String → char array	`toCharArray()`
char array → String	`new String(charArray)`
String → byte array	`getBytes()`
byte array → String	`new String(byteArray)`
String → long	`Long.parseLong(str)`
long → String	`Long.toString(num)`

String → float	`Float.parseFloat(str)`
float → String	`Float.toString(num)`
String → BigInteger	`new BigInteger(str)`
BigInteger → String	`bigInteger.toString()`
String → BigDecimal	`new BigDecimal(str)`
BigDecimal → String	`bigDecimal.toString()`

---

4. Changing Case

Java provides methods to change the case of a string:

- `toUpperCase()`: Converts the string to uppercase.
- `toLowerCase()`: Converts the string to lowercase.

Lab 5: Changing Case

```
public class ChangeCase {
    public static void main(String[] args) {
        String str = "Hello, World!";

        String upperCaseStr = str.toUpperCase();
        String lowerCaseStr = str.toLowerCase();

        System.out.println("Uppercase: " + upperCaseStr);
        System.out.println("Lowercase: " + lowerCaseStr);
    }
}
```

Output:

```
Uppercase: HELLO, WORLD!
Lowercase: hello, world!
```

---

5. Character Extraction

We can extract characters from a string using the `charAt()` method or by converting the string to a character array.

Lab 6: Character Extraction

```
public class CharacterExtraction {
    public static void main(String[] args) {
        String str = "Java";

        // Using charAt()
        char ch1 = str.charAt(2); // Index starts from 0

        // Using toCharArray()
        char[] charArray = str.toCharArray();
```

```
        System.out.println("Character at index 2: " + ch1);
        System.out.println("Character Array: " + Arrays.toString(charArray));
//Character Array: [J, a, v, a]
        //String str = new String(charArray);
        //System.out.println(str);  // Output: Hello
    }
}
```

Output:

```
Character at index 2: v
Character Array: [J, a, v, a]
```

---

6. String Comparison

Strings can be compared using:

- `equals()` : Compares the content of two strings.
- `equalsIgnoreCase()` : Compares strings ignoring case.
- `compareTo()` : Compares strings lexicographically(based on the Unicode value of each character in the string). The Unicode value for 'H' is 72 and for 'h' is 104. Since 'H' (72) is lexicographically smaller than 'h' (104), the compareTo() method will return a negative integer (specifically, -32, which is the difference in Unicode values between 'H' and 'h').

Lab 7: String Comparison

```
public class StringComparison {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "hello";

        System.out.println("equals(): " + str1.equals(str2));
        System.out.println("equalsIgnoreCase(): " + str1.equalsIgnoreCase(str2));
        System.out.println("compareTo(): " + str1.compareTo(str2));
    }
}
```

Output:

```
equals(): false
equalsIgnoreCase(): true
compareTo(): -32
```

---

7. Searching Strings

We can search for characters or substrings within a string using methods like `indexOf()` and `contains()`.

Lab 8: Searching Strings

```java
public class Main {
    public static void main(String[] args) {
        String str = "Hello, World!";//main string

        // Search for a character
        int index = str.indexOf('W');
        System.out.println("Index of 'W': " + index);

        // Search for a substring
        boolean contains = str.contains("World");//sub-string
        System.out.println("Contains 'World': " + contains);
    }
}
```

Output:

```
Index of 'W': 7
Contains 'World': true
```

Explanation:

- `indexOf()` returns the index of the first occurrence of a character or substring.
- `contains()` returns `true` if the substring is found.

Summary of String Searching Methods:

- **`indexOf()`**: Finds the first occurrence of a character or substring within a string.
- **`lastIndexOf()`**: Finds the last occurrence of a character or substring within a string.
- **`contains()`**: Checks if a specific substring exists within the string.
- **`startsWith()`**: Checks if the string starts with a given substring.
- **`endsWith()`**: Checks if the string ends with a specified substring.
- **`matches()`**: Checks if the string matches a given regular expression.

---

8. Modifying Strings

Strings in Java are immutable, which means once a string is created, it cannot be changed. However, we can create new strings with modifications using methods like `replace()`, `substring()`, and `trim()`.

Lab 9: Modifying Strings

```java
public class Main {
    public static void main(String[] args) {
        String str = "  Hello, World!  ";

        // Replace characters
        String replacedStr = str.replace('o', 'x');// immutable: The original string 'str' remains unchanged but returns a new string with the modified value.
        System.out.println("Replaced: " + replacedStr);
```

```
        // Extract substring
        String substring = str.substring(7, 12);// immutable: The original string 'str' remains unchanged but returns a new string with the modified value.
        System.out.println("Substring: " + substring);

        // Trim whitespace
        String trimmedStr = str.trim();// immutable: The original string 'str' remains unchanged but returns a new string with the modified value.
        System.out.println("Trimmed: " + trimmedStr);
    }
}
```

Output:

```
Replaced:  Hellx, Wxrld!
Substring: World
Trimmed: Hello, World!
```

Explanation:

- `replace()` replaces all occurrences of a character.
- `substring()` extracts a portion of the string.
- `trim()` removes leading and trailing whitespace.

Summary of String Modification Methods:

- **`concat()`** : Concatenate two strings.
- **`replace()`** : Replace characters or substrings.
- **`toUpperCase()` and `toLowerCase()`** : Convert to uppercase or lowercase.
- **`trim()`** : Remove leading and trailing whitespace.
- **`substring()`** : Extract a portion of the string.
- **`replaceAll()` and `replaceFirst()`** : Replace based on regular expressions.
- **`split()`** : Split the string into an array of substrings.
- **`StringBuilder` and `StringBuffer`** : Use mutable strings for efficient modifications.

---

9. String Buffer

`StringBuffer` is a mutable sequence of characters. It is synchronized(thread-safe) and is used when we need to modify strings frequently.

Lab 10: String Buffer

```
public class Main {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("Hello");

        // Append to StringBuffer
        sb.append(", World!");//This modifies the original StringBuffer (mutable operation)
        System.out.println("After append: " + sb);

        // Insert into StringBuffer
```

```
        sb.insert(5, " Java");//This modifies the original StringBuffer (mutable
operation)
        System.out.println("After insert: " + sb);

        // Reverse StringBuffer
        sb.reverse();//This modifies the original StringBuffer (mutable operation)
        System.out.println("After reverse: " + sb);
    }
}
```

Output:

```
After append: Hello, World!
After insert: Hello Java, World!
After reverse: !dlroW ,avaJ olleH
```

Explanation:

- `append()` adds text to the end.
- `insert()` inserts text at a specified position.
- `reverse()` reverses the string.

Summary of StringBuffer Methods:

- **`append(String str)`**: Adds the specified string to the end of the `StringBuffer`.
  - **Arguments**: `str` - The string to be appended.
- **`insert(int index, String str)`**: Inserts the specified string at the given index in the `StringBuffer`.
  - **Arguments**: `index` - The index where the string should be inserted, `str` - The string to be inserted.
- **`delete(int start, int end)`**: Removes characters from the `StringBuffer` between the specified start and end indices.
  - **Arguments**: `start` - The starting index, `end` - The ending index (exclusive).
- **`replace(int start, int end, String str)`**: Replaces characters between the specified start and end indices with a new string.
  - **Arguments**: `start` - The starting index, `end` - The ending index (exclusive), `str` - The string to replace the characters.
- **`reverse()`**: Reverses the characters in the `StringBuffer`.
  - **Arguments**: None.
- **`capacity()`**: Returns the current capacity of the `StringBuffer`.
  - **Arguments**: None.
- **`length()`**: Returns the number of characters currently stored in the `StringBuffer`.
  - **Arguments**: None.

- **`setLength(int newLength)`**: Sets the length of the `StringBuffer`, truncating or padding as needed.
  - **Arguments**: `newLength` - The new length to set for the `StringBuffer`.
- **`ensureCapacity(int minimumCapacity)`**: Ensures the `StringBuffer` has at least the specified capacity.
  - **Arguments**: `minimumCapacity` - The minimum capacity to ensure.
- **`charAt(int index)`**: Returns the character at the specified index.
  - **Arguments**: `index` - The index of the character to retrieve.
- **`substring(int start, int end)`**: Extracts a substring from the `StringBuffer` and returns it as a new string.
  - **Arguments**: `start` - The starting index, `end` - The ending index (exclusive).

What students have learned

1. **Creation, Concatenation, and Conversion of Strings**: Students learned how to create strings, combine them using concatenation, and convert data types (e.g., numbers to strings or vice versa).
2. **Changing Case**: They explored methods for altering the case of strings, such as converting text to uppercase or lowercase.
3. **Character Extraction**: Students gained skills in extracting specific characters or substrings from a string, including accessing individual characters by index or slicing portions of the string.
4. **String Comparison**: They studied techniques for comparing strings to determine equality or lexicographical order.
5. **Searching Strings**: Students learned how to search for specific characters, substrings, or patterns within a string using various methods.
6. **Modifying Strings**: They were introduced to methods for modifying strings, such as replacing parts of a string or removing unwanted characters.
7. **String Buffer**: Finally, students understood the concept of mutable strings through the use of String Buffer (or StringBuilder in some languages), which allows efficient modification of strings without creating new objects.

By the end of the chapter, students acquired practical skills for manipulating and managing strings, which are essential for tasks like text processing, data validation, and more in programming.