Week 11 - Java String & StringBuilder

Louis Botha `louis.botha@tuni.fi`

Java String

What is a Java String? It's a `char` array of course!

In Java, a string is an object that represents a sequence of characters or char values. For example, "hello" is a string containing a sequence of characters 'h', 'e', 'l', 'l', and 'o'.

The `java.lang.String` class is used to create a Java String object (not a primitive type).

There are two ways to create a String object:

1. **By string literal** : Java String literal is created by using double quotes.
`String s = "Welcome";`

> *:attention: Notice that the type is* ***S****tring (capital S)*

2. **By new keyword** : Java String is created by using a keyword “new”.
`String s = new String("Welcome");`

Immutable String in Java

> *:attention: Interview question*

An immutable object is an **object whose internal state remains constant after it has been entirely created**. This means that once the object has been assigned to a variable, we can neither update the reference nor mutate the internal state by any means.

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc.

In Java, **String objects are immutable**. Once a String object is created its data or state can't be changed but a new String object is created.

Why Strings are immutable in nature?
Here are some more reasons for making String immutable in Java:

- The String pool cannot be possible if String is not immutable in Java. A lot of heap space is saved by JRE. The same string variable can be referred to by more than one string variable in the pool. More about the String pool later.
- If we don’t make the String immutable, it will pose a serious security threat to the application. For example, database usernames, passwords are passed as strings to receive database connections. The socket programming host and port descriptions are also passed as strings. The String is immutable, so its value cannot be changed. If the String doesn’t remain immutable, any hacker can cause a security issue in the application by changing the reference value.
- The String is safe for multithreading because of its immutableness. Different threads can access a single “String instance”. It removes the synchronization for thread safety because we make strings thread-safe implicitly.

- Immutability gives the security of loading the correct class by `Classloader`. For example, suppose we have an instance where we try to load the java.sql.Connection class but the changes in the referenced value points to the myhacked.Connection class that does unwanted things to our database.

🗒️ **Small Side Note - mutable vs immutable**
In programming, an object or data type is said to be mutable if its state can be modified after it is created, while an object or data type is said to be immutable if its state cannot be modified after it is created.

Immutable objects or data types are often preferred in programming because they are simpler and more predictable. Since their state cannot be modified, they are inherently thread-safe and can be shared between multiple threads without the need for synchronization. Additionally, they are less prone to bugs caused by unintended side effects, since their state cannot be modified.

Some examples of immutable objects in Java include `String`, `Integer`, `Boolean`, and `LocalDate`. These objects cannot be modified after they are created, so any operations that appear to modify them actually create a **new** object with the modified state.

Mutable objects, on the other hand, can be modified after they are created, either by changing the values of their fields or by calling methods that modify their internal state. Examples of mutable objects in Java include `StringBuilder`, `ArrayList`, and `HashMap`.

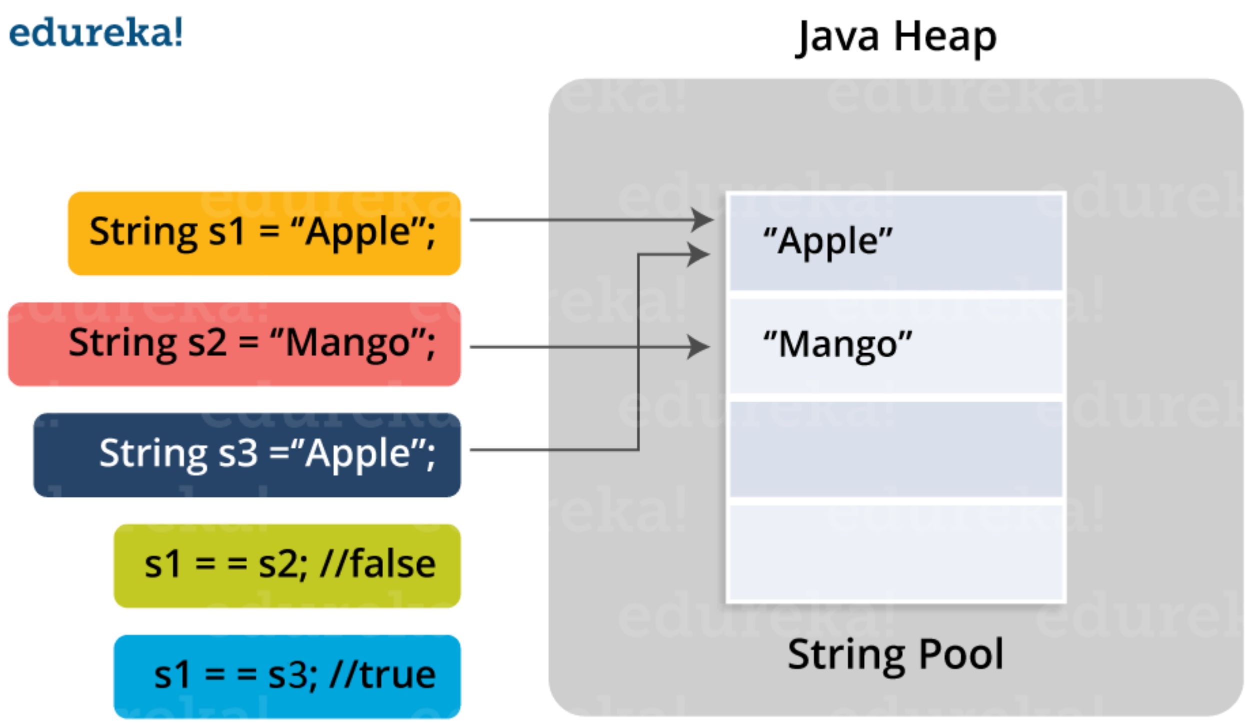
Mutable objects can be useful in certain situations where performance is a concern or where it is necessary to modify the state of an object. However, they are generally more complex and require more careful handling to avoid unintended side effects. Additionally, mutable objects can be more difficult to use in concurrent environments, since their state can be modified by multiple threads at the same time, leading to race conditions and other synchronization issues.

Java String Pool

> *:attention: Interview question*

Java String pool refers to collection of Strings which are stored in heap memory. In this, whenever a new object is created, String pool first checks whether the object is already present in the pool or not. If it is present, then same reference is returned to the variable else new object will be created in the String pool and the respective reference will be returned.

Refer to the diagrammatic representation for better understanding:



String-pool - Edureka

In the above image, two Strings are created using literal i.e “Apple” and “Mango”. Now, when third String is created with the value “Apple”, instead of creating a new object, the already present object reference is returned.

Run the following code:

```
class Main {
    public static void main(String[] args) {
        String s1 = "apple";
        String s2 = "mango";
        String s3 = "apple";
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);

        System.out.println(Integer.toHexString(System.identity
HashCode(s1)));
        System.out.println(Integer.toHexString(System.identity
HashCode(s2)));
        System.out.println(Integer.toHexString(System.identity
HashCode(s3)));
    }
}

// Output
apple
mango
apple
58b835a
62dc3c2
58b835a
```

Notice that the `s1` and `s3` are pointing to the same reference.

🗒️ **Small Side Note**

💡 We can use the `hashCode()` method because we are not overriding the implementation.

> *The `hashCode()` method returns an integer value that is based on the contents of the object. Two objects that are equal according to the `equals()` method must return the same hash code value, but two objects that are not equal according to the `equals()` method can still return the same hash code value.*

💡 We are using the `identityHashCode()`

> *The identity hash code is a system-generated integer value that is unique for each object instance. It is based on the memory address where the object is stored in memory, and is not affected by the object's state or any of its properties.*

Java String Methods

The String class has a set of built-in methods that you can use with Strings.

For example here is an example of how retrieve the character at a specific position in a string. The method is called **charAt().** This method also needs you to supply an index (number), position of the character you are interested in.

```
String myStr = "Hello";
char result = myStr.charAt(1);
System.out.println(result);

//Output
e
```

Here is a table with some of the String methods. You don’t have to memorize all of the methods but it is good to have an idea what methods are available. These methods tend to make life much easier 🙂.

Method	Description	Return Type
charAt()	Returns the character at the specified index (position)	char
compareTo()	Compares two strings lexicographically	int
compareToIgnoreCase()	Compares two strings lexicographically, ignoring case differences	int
concat()	Appends a string to the end of another string	String

contains()	Checks whether a string contains a sequence of characters	boolean
contentEquals()	Checks whether a string contains the exact same sequence of characters of the specified CharSequence or StringBuffer	boolean
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
equalsIgnoreCase()	Compares two strings, ignoring case considerations	boolean
hashCode()	Returns the hash code of a string	int
indexOf()	Returns the position of the first found occurrence of specified characters in a string	int
isEmpty()	Checks whether a string is empty or not	boolean
lastIndexOf()	Returns the position of the last found occurrence of specified characters in a string	int
length()	Returns the length of a specified string	int
replace()	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]

startsWith()	Checks whether a string starts with specified characters	boolean
substring()	Returns a new string which is the substring of a specified string	String
toLowerCase()	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String
trim()	Removes whitespace from both ends of a string	String

Java String Concatenation

We can combine strings with **+**

```
public class Main {
  public static void main(String args[]) {
    String firstName = "John";
    String lastName = "Doe";
    System.out.println(firstName + " " + lastName);
  }
}


//Output
John Doe
```

> *:attention: Note that we are explicitly adding the space between the 2 strings.*

We can also combine stings with the concat() method

```
public class Main {
  public static void main(String[] args) {
    String firstName = "John ";
    String lastName = "Doe";
    System.out.println(firstName.concat(lastName));
  }
}


//Output
John Doe
```

> *:attention: Notice that we added the space at the end of the firstName string.*

Use `System.out.println()` to test the following concatenation examples.

```
1 + 2
"string1" + "string2"
"string1" + 2
"string1" + 3.3
3.3 + "string" + 3.3
2 + "string2"
"a" + "b"
"a" + 'b'
'a' + "b"
'a' + 'b'  //97 + 98
```

> *:attention: ‘a’ + ‘b’ = 195, WTF?*

In Java, the `char` variable stores the ASCII value of a character (number between 0 and 127) rather than the character itself.

The ASCII value of lowercase alphabets are from 97 to 122. And, the ASCII value of uppercase alphabets are from 65 to 90. That is, alphabet a is stored as **97** and alphabet z is stored as **122**.
Similarly, alphabet A is stored as **65** and alphabet Z is stored as **90**.

Now, in that statement we are adding up the ASCII values of the letters, why we get the result of 195.

String Escape Character

The backslash \ escape character turns special characters into string characters:

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

Here we use the \ to display the double quotes around Vikings.

```
public class Main {
  public static void main(String[] args) {
    String txt = "We are the so-called \"Vikings\" from the no
rth.";
    System.out.println(txt);
  }
}


//Output
We are the so-called "Vikings" from the north.
```

Some common escape sequences that are valid in Java are:

Code	Result
\n	New Line
\r	Carriage Return
\t	Tab
\b	Backspace
\f	Form Feed

The two that are most often used are **\n** and **\t**. Here we use the **\n** escape sequence to display the Doe on a new line:

```
public class Main {
  public static void main(String args[]) {
    String firstName = "John\n";
    String lastName = "Doe";
    System.out.println(firstName + " " + lastName);
  }
}

//Output
John
 Doe
```

Here we use the **\t** escape sequence to display John and Doe separated by one tab away

```
public class Main {
  public static void main(String args[]) {
    String firstName = "John\t";
    String lastName = "Doe";
    System.out.println(firstName + " " + lastName);
  }
}

//Output
John    Doe
```

StringBuilder

Java `StringBuilder` is a class in Java that represents a mutable sequence of characters. It is similar to the String class, but provides more flexibility and performance when it comes to manipulating strings.

The `StringBuilder`class is part of the `java.lang` package and provides methods to append, insert, and delete characters in a string, as well as methods to reverse, replace, and substring a string. It also has a capacity, which is the number of characters that can be stored without requiring the `StringBuilder` object to allocate more memory. If more characters are added to the `StringBuilder` object than its current capacity, it automatically expands its capacity.

Unlike the String class, `StringBuilder` objects are mutable. This means that it allows you to modify the value of a string without creating a new object each time. This can be more efficient than using the String class, which creates a new object every time you modify the string. `StringBuilder` is especially useful when you need to concatenate multiple strings or modify a string repeatedly.

To use `StringBuilder`, you can create a new `StringBuilder` object and initialize it with an initial value, such as an empty string or an existing string. You can then use its methods to append, insert, or modify the characters in the `StringBuilder` object. Once you have finished modifying the string, you can convert the `StringBuilder` object to a String using the `toString()` method.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(", world!");   // Append ", world!" to the StringBuilder object
sb.insert(5, " there");  // Insert " there" at index 5
String result = sb.toString(); // Convert the StringBuilder object to a String

System.out.println(result);    // Output: "Hello there, world!"
```

Here's a more advanced example of how to use Java `StringBuilder` to generate a CSV (Comma Separated Values) file from a list of data:

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class CSVGenerator {
    public static void main(String[] args) {
        // Sample data
        List<String[]> data = new ArrayList<>();
        data.add(new String[]{"Name", "Age", "Gender"});
        data.add(new String[]{"John", "25", "Male"});
        data.add(new String[]{"Jane", "30", "Female"});
        data.add(new String[]{"Bob", "40", "Male"});

        // StringBuilder to generate CSV content
        StringBuilder sb = new StringBuilder();

        // Loop through each row of data
        for (String[] row : data) {
            // Loop through each column of the row
            for (int i = 0; i < row.length; i++) {
                // Append the value to the StringBuilder
                sb.append(row[i]);

                // If this is not the last column, append a co
mma
                if (i < row.length - 1) {
                    sb.append(",");
                }
            }
            // Append a new line character to separate rows
            sb.append("\n");
        }

        // Write the CSV content to a file
        try (FileWriter writer = new FileWriter("data.csv")) {
            writer.write(sb.toString());
            System.out.println("CSV file generated successfull
y.");
        } catch (IOException e) {
            System.err.println("Error generating CSV file: " +
e.getMessage());
        }
    }
}
```

StringBuilder Methods

The StringBuilder in Java provides numerous methods to perform different operations on the string builder. The table depicted below enlists some primary methods from the StringBuilder class.

Method	Description
StringBuilder append (String s)	This method appends the mentioned string with the existing string. You can also with arguments like boolean, char, int, double, float, etc.
StringBuilder insert (int offset, String s)	It will insert the mentioned string to the other string from the specified offset position. Like

	append, you can overload this method with arguments like (int, boolean), (int, int), (int, char), (int, double), (int, float), etc.
StringBuilder replace(int start, int end, String s)	It will replace the original string with the specified string from the start index till the end index.
StringBuilder delete(int start, int end)	This method will delete the string from the mentioned start index till the end index.
StringBuilder reverse()	It will reverse the string.
int capacity()	This will show the current StringBuilder capacity.
void ensureCapacity(int min)	This method ensures that the StringBuilder capacity is at least equal to the mentioned minimum.
char charAt(int index)	It will return the character at the specified index.
int length()	This method is used to return the length (total characters) of the string.
String substring(int start)	Starting from the specified index till the end, this method will return the substring.

String substring(int start, int end)	It will return the substring from the start index till the end index.
int indexOf(String str)	This method will return the index where the first instance of the specified string occurs.
int lastIndexOf(String str)	It will return the index where the specified string occurs the last.
Void trimToSize()	It will attempt to reduce the size of the StringBuilder.

Activity

We are going to create a cheatsheet containing **runnable** examples of the `String` and `StringBuilder` methods. The examples are collected on Mural as **screenshots**. The methods on the tables above are the most used but you can go have a look at the String and StringBuilder documentation to find a method not listed.

An example of the code snippet to make and paste on Mural

```
public class Example {
    public static void main(String[] args) {
        String myStr = "Hello";
        char result = myStr.charAt(1);
        System.out.println(result);
    }
}
//Output
e
```