Java Foundation with Data Structures

Lecture 7: Strings

Strings in Java

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects(we study about objects in detail in OOPS lecture). The Java platform provides the String class to create and manipulate strings. The most direct and easiest way to create a string is to write:

```
String str = "Hello world";
```

In the above example, "Hello world!" is a *string literal*—a series of characters in code that is enclosed in double quotes. Whenever it encounters a string literal in code, the compiler creates a String object with its value—in this case, Hello world!.

Note: Strings in Java are immutable, thus we cannot modify its value. If we want to create a mutable string we can use StringBuffer and StringBuilder classes.

A String can be constructed by either:

1. directly assigning a string literal to a String reference - just like a primitive, or
2. via the "new" operator and constructor, similar to any other classes(like arrays and scanner). However, this is not commonly-used and is not recommended.

For example,

```
String str1 = “Java is Amazing!”;          // Implicit construction via string literal
String str2 = new String(“Java is Cool!”); // Explicit construction via new
```

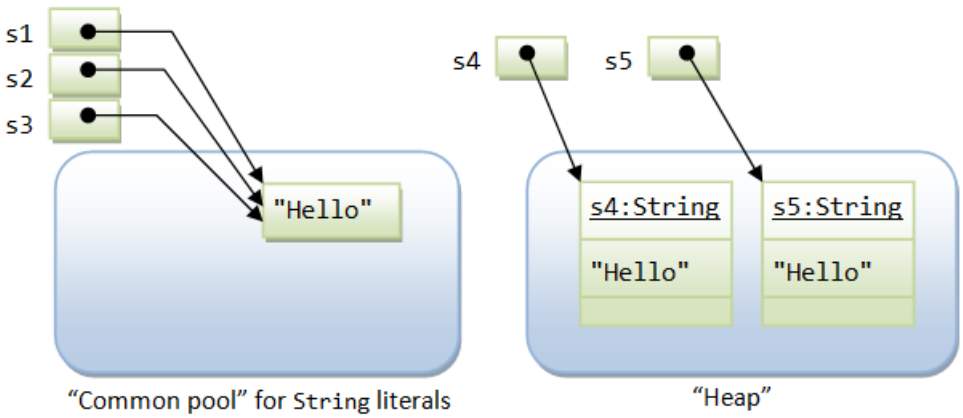
In the first statement, str1 is declared as a String reference and initialized with a string literal "Java is Amazing". In the second statement, str2 is declared as a String reference and initialized via the new operator to contain "Java is Cool".

String literals are stored in a common pool called String pool. This facilitates sharing of storage for strings with the same contents to conserve storage. String objects allocated via new operator are stored in the heap memory(all non primitives created via new operator are stored in heap memory), and there is no sharing of storage for the same contents.

1. String Literal v/s String Object

As mentioned, there are two ways to construct a string: implicit construction by assigning a string literal or explicitly creating a String object via the new operator and constructor. For example,

```
String s1 = "Hello";                  // String literal
String s2 = "Hello";                  // String literal
String s3 = s1;                       // same reference
String s4 = new String("Hello");      // String object
String s5 = new String("Hello");      // String object
```



Java has provided a special mechanism for keeping the String literals - in a so-called *string common pool*. If two string literals have the same contents, they will share the same storage inside the common pool. This approach is adopted to *conserve storage* for frequently-used strings. On the other hand, String objects created via the new operator and constructor are kept in the heap memory. Each String object in the heap has its own storage just like any other object. There is no sharing of storage in heap even if two String objects have the same contents.

You can use the method equals() of the String class to compare the contents of two Strings. You can use the relational equality operator '==' to compare the references (or pointers) of two objects. Study the following codes for s1 and s2 defined in code above:

```
s1 == s1;                 // true, same pointer
```

```
s1 == s2;              // true, s1 and s1 share storage in common pool
s1 == s3;              // true, s3 is assigned same pointer as s1
s1.equals(s3);       // true, same contents
s1 == s4;              // false, different pointers
s1.equals(s4);       // true, same contents
s4 == s5;              // false, different pointers in heap
s4.equals(s5);       // true, same contents
```

2. String is Immutable

Since string literals with the same contents share storage in the common pool, Java's String is designed to be immutable. That is, once a String is constructed, its contents cannot be modified. Otherwise, the other String references sharing the same storage location will be affected by the change, which can be unpredictable and therefore is undesirable. Methods such as toUpperCase() might appear to modify the contents of a String object. In fact, a completely new String object is created and returned to the caller. The original String object will be deallocated, once there is no more references, and subsequently garbage-collected.

Because String is immutable, it is not efficient to use String if you need to modify your string frequently (**that would create many new Strings occupying new storage areas**).

For example,

```
// inefficient code
String str = "Hello";
for (int i = 1; i < 1000; ++i) {
    str = str + i;
}
```

3. StringBuilder & StringBuffer

As explained earlier, Strings are immutable because String literals with same content share the same storage in the string common pool. Modifying the content of one String directly may cause adverse side-effects to other Strings sharing the same storage.

JDK provides two classes to support mutable strings: StringBuffer and StringBuilder (in core package java.lang) . A StringBuffer or StringBuilder object is just like any ordinary object, which are stored in the heap and not shared,

and therefore, can be modified without causing adverse side-effect to other objects.

StringBuilder class was introduced in JDK 1.5. It is the same as StringBuffer class, except that StringBuilder is not synchronized for multi-thread operations(you can read more about multi threading). However, for single-thread program, StringBuilder, without the synchronization overhead, is more efficient.

4. Important Java Methods

1. String "Length" Method

String class provides an inbuilt method to determine the length of the Java String. For example:

```
String str1 = "test string";
//Length of a String
System.out.println("Length of String: " + str.length());
```

2. String "indexOf" Method

String class provides an inbuilt method to get the index of a character in Java String. For example:

```
String str1 = "the string";
System.out.println("Index of character 's': " + str_Sample.indexOf('s')); // returns 5
```

3. String "charAt" Method

Similar to the above question, given the index, how do I know the character at that location? Simple one again!! Use the “charAt” method and provide the index whose character you need to find.

```
String str1 = "test string";
System.out.println("char at index 3 : " + str.charAt());
// output – ‘t’
```

4. String "CompareTo" Method

This method is used to compare two strings. Use the method “compareTo” and specify the String that you would like to compare.

Use “compareToIgnoreCase” in case you don’t want the result to be case sensitive.

The result will have the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

```
String str = "test";
System.out.println("Compare To “test”: " + str.compareTo("test"));

//Compare to - Ignore case
System.out.println("Compare To “test”: - Case Ignored: " + str.compareToIgnoreCase("Test"));
```

5. String "Contain" Method

Use the method “contains” to check if a string contains another string and specify the characters you need to check.

Returns **true** if and only if this string contains the specified sequence of char values.

```
String str = "test string";
System.out.println("Contains sequence ing: " + str.contains("ing"));
```

6. String "endsWith" Method

This method is used to find whether a string ends with particular prefix or not.

Returns **true** if the character sequence represented by the argument is a suffix of the character sequence represented by this object.

```
String str = "star";
System.out.println("EndsWith character 'r': " + str.endsWith("r"));
```

7. String "replaceAll" & "replaceFirst" Method

Java String Replace, replaceAll and replaceFirst methods. You can specify the part of the String you want to replace and the replacement String in the arguments.

```
String str = "sample string";
System.out.println("Replace sample with test: " + str.replace("sample", "test"));
```

8. String Java "tolowercase" & Java "touppercase"

Use the “toLowercase()” or “ToUpperCase()” methods against the Strings that need to be converted.

```
String str = "TEST string";
System.out.println("Convert to LowerCase: " + str.toLowerCase());
//Convert to UpperCase
System.out.println("Convert to UpperCase: " + str.toUpperCase());}}
```

Other Important Java String methods:

No.	Method	Description
1	String substring(int beginIndex)	returns substring for given begin index
2	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
3	boolean isEmpty()	checks if string is empty
4	String concat(String str)	concatinates specified string
5	String replace(char old, char new)	replaces all occurrences of specified char value
6	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
7	String[] split(String regex)	returns splitted string matching regex
8	String[] split(String regex, int limit)	returns splitted string matching regex and limit
9	int indexOf(int ch)	returns specified char value index
10	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index

11	int indexOf(String substring)	returns specified substring index
12	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
13	String trim()	removes beginning and ending spaces of this string.
14	static String valueOf(int value)	converts given type into string. It is overloaded.

Some of the key points about Java Strings:

1. **Strings are not NULL terminated in Java:**
   Unlike C and C++, String in Java doesn't terminate with null character. Instead String is an Object in Java and backed by character array. You can get the character array used to represent String in Java by calling toCharArray() method of java.lang.String class of JDK.
2. **Internally, String is stored as a character array only.**
3. **String is a Immutable and Final class**; i.e once created the value cannot be altered. Thus String objects are called immutable.
4. The Java Virtual Machine(JVM) creates a memory location especially for Strings called **String Constant Pool**. That's why String can be initialized without 'new' key word.