Strings In Java

Introduction to Strings

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects. 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 their value. If we want to create a mutable string, we can use StringBuffer and StringBuilder classes.

Either can construct a String:

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 scanners). However, this is not commonly used and is not recommended.

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 operators are stored in the heap memory(all non-primitive created via new

operators are stored in heap memory). There is no sharing of storage for the same contents.

Creating char arrays and treat them as a string

An array of characters works the same as a Java string.

Example:

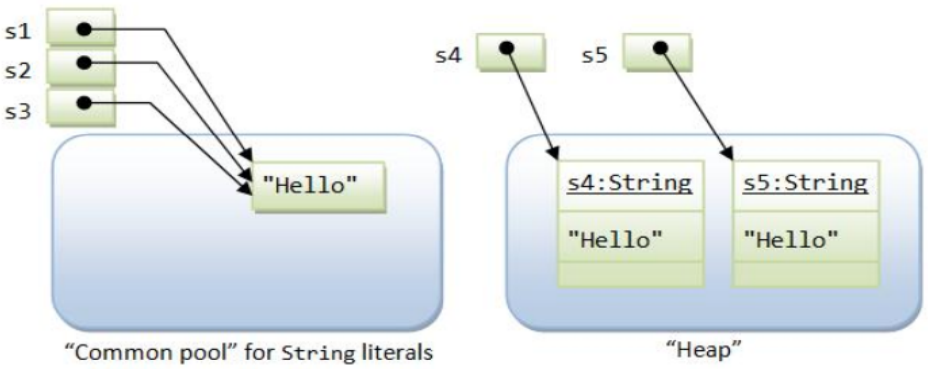
```
public class HelloWorld {
    public static void main(String[] args) {
        char[] ch = {'H', 'e', 'l', 'l', 'o'};
        System.out.println(ch);
    }
}
```

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.

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 a heap has its storage, just like any other object. There is no sharing of storage in the heap even if two String objects have the same contents.

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

Example:

```
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
```

Accessing the string elements

We can access the characters in a string using the **charAt()** method. The "charAt" method returns the character at the specified index in a string.

It throws *IndexOutOfBoundsException* if the index value passed in the charAt() method is less than zero or greater than or equal to the string's length.

Syntax:

```
string_name.charAt(index_number);
```

Example:

```
public class HelloWorld {

    public static void main(String[] args) {
        String str = "test string";

        System.out.println("char at index 3: " + str.charAt(3));
        System.out.println("char at index 5: " + str.charAt(5));

    }
}
```

Output:

```
char at index 3: t
char at index 5: s
```

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 undesirable. Methods such as toUpperCase() might appear to modify the contents of a String object. A completely new String object is created and returned to the caller. The original String object will be deallocated once there are 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**).

Example:

```
//inefficient code
String str = "Hello";
```

```
for (int i = 1; i < 1000; ++i) {
  str = str + i;
}
```

StringBuilder & StringBuffer

As explained earlier, Strings are immutable because String literals with the 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 stored in the heap and not shared. Therefore can be modified without causing adverse side-effect to other objects.

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

String Concatenation

Concatenation of two strings is the joining of them to form a new string. There are various ways to concatenate strings, and they are as follows:

- '+' Operator
- concat() method

i) + Operator: The '+' operator adds the two input strings and returns a new string that contains the concatenated string.

Syntax:

```
String new_string = string1 + string2;
```

Example:

```
public class HelloWorld {
```

```
    public static void main(String[] args) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        str1 = str1 + str2;

        System.out.println("Concatenated String: " + str1);

    }
}
```

Output:
Concatenated String: Coding Ninjas!

ii) concat() method: The String concat() method concatenates the specified string to the end of the current string.

Syntax:

```
string1.concat(string2);
```

Example:

```
public class HelloWorld {

    public static void main(String[] args) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        str1 = str1.concat(str2);

        System.out.println("Concatenated String: " + str1);

    }
}
```

Output:
Concatenated String: Coding Ninjas!

String Comparison

To check if the two strings are equal or not, a string comparison is done. Strings in Java can be compared using either of the following techniques:

- equals() method
- == method
- compareTo() method

i) equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- public boolean equals(Object another) compares this string to the specified object.
- public boolean equalsIgnoreCase(String another) compares this String to another string, ignoring case.

Example:

```
public class HelloWorld {

    public static void main(String[] args) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        String str3 = "Coding";
        String str4 = "coding";

        System.out.println(str1.equals(str2));
        System.out.println(str1.equals(str3));
        System.out.println(str1.equalsIgnoreCase(str4));

    }
}

Output:
false
true
true
```

ii) == method

The == operator compares references not values.

Example:

```
public class HelloWorld {

    public static void main(String[] args) {
        String s1 = "Coding";
        String s2 = "Coding";
        String s3 = new String("Coding");
        System.out.println(s1 == s2); //true (because both refer to same
instance)
        System.out.println(s1 == s3); //false(because s3 refers to instance
created in nonpool)

    }
}

Output:
true
false
```

iii) compareTo() Method

The String compareTo() method compares values lexicographically and returns an integer value that describes if the first string is less than, equal to or greater than the second string.

Suppose s1 and s2 are two string variables. If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

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

Example:

```
public class HelloWorld {

    public static void main(String[] args) {
        String str1 = "Coding";
        String str2 = " Ninjas!";
        String str3 = "Coding";
        String str4 = "coding";

        System.out.println(str1.compareTo(str2));
        System.out.println(str1.compareTo(str3));
        System.out.println(str1.compareToIgnoreCase(str4));

    }
}
```

Output:

```
35
0
0
```