Day 39

Strings

Strings (java.lang.String) are pieces of text stored in your program. Strings are not a primitive data type in Java, however, they are very common in Java programs.

In Java, Strings are immutable, meaning that they cannot be changed.

Comparing Strings

In order to compare Strings for equality, you should use the String object's equals or equalsIgnoreCase methods. For example, the following snippet will determine if the two instances of String are equal on all characters:

```
String firstString = “Test123”;
String second String = “Test”+ 123;

 if (firstString.equals(secondString)){
        // Both Strings have the same content.
       }
```

This example will compare them, independent of their case:

```
 String firstString = "Test123";
  String secondString = "TEST123";

if (firstString.equalsIgnoreCase(secondString)) {
    // Both Strings are equal, ignoring the case of the individual characters.
   }
```

Note that equalsIgnoreCase does not let you specify a Locale. For instance, if you compare the two words "Taki" and "TAKI" in English they are equal; however, in Turkish they are different (in Turkish, the lowercase I is ı). For cases like this, converting both strings to lowercase (or uppercase) with Locale and then comparing with equals is the solution.

```
String firstString = "Taki";
String secondString = "TAKI";

  System.out.println(firstString.equalsIgnoreCase(secondString)); //prints true

Locale locale = Locale.forLanguageTag("tr-TR");

System.out.println(firstString.toLowerCase(locale).equals(secondString.toLowerCase(locale)));
        //prints false
```

Do not use the == operator to compare Strings

Unless you can guarantee that all strings have been interned (see below), you should not use the == or != operators to compare Strings. These operators actually test references, and since multiple String objects can represent the same String, this is liable to give the wrong answer.

Instead, use the String.equals(Object) method, which will compare the String objects based on their values. For a detailed explanation, please refer to Pitfall: using == to compare strings.

Comparing Strings in a switch statement
it is possible to compare a String variable to literals in a switch statement. Make sure that the String is not null, otherwise it will always throw a **NullPointerException**. Values are compared using **String.equals**, i.e. case sensitive.

```
String stringToSwitch = "A";

      switch (stringToSwitch) {
                  case "a":
                        System.out.println("a");
                        break;
                case "A":
                        System.out.println("A"); //the code goes here
                        break;
                 case "B":
                          System.out.println("B");
                         break;
                default:
                     break;
                       }
```

Comparing Strings with constant values
When comparing a String to a constant value, you can put the constant value on the left side of equals to ensure that you won't get a **NullPointerException** if the other String is **null**.

```
"baz".equals(foo)
```

While **foo.equals("baz")** will throw a **NullPointerException** if foo is **null**, "**baz**".equals(foo) will evaluate to false.

A more readable alternative is to use Objects.equals(), which does a null check on both parameters: Objects.equals(foo, "baz")

Comparing with interned Strings
"Moreover, a string literal always refers to the same instance of class String. This is because string literals - or, more generally, strings that are the values of constant expressions - are interned so as to share unique instances, using the method **String.intern.**"

This means it is safe to compare references to two string literals using ==. Moreover, the same is true for references to String objects that have been produced using the String.intern() method.

For example:

```
String strObj = new String("Hello!");
String str = "Hello!";

// The two string references point two strings that are equal
     if (strObj.equals(str)) {
         System.out.println("The strings are equal");
    }

// The two string references do not point to the same object
    if (strObj != str) {
       System.out.println("The strings are not the same object");
    }

// If we intern a string that is equal to a given literal, the result is
// a string that has the same reference as the literal.

    String internedStr = strObj.intern();

   if (internedStr == str) {
        System.out.println("The interned string and the literal are the same object");
        }
```

Behind the scenes, the interning mechanism maintains a hash table that contains all interned strings that are still reachable. When you call **intern()** on a String, the method looks up the object in the hash table:

- If the string is found, then that value is returned as the interned string.
- Otherwise, a copy of the string is added to the hash table and that string is returned as the interned string.

It is possible to use interning to allow strings to be compared using ==. However, there are significant problems with doing this;

Changing the case of characters within a String

The String type provides two methods for converting strings between upper case and lower case:

- **toUpperCase** to convert all characters to upper case
- **toLowerCase** to convert all characters to lower case

These methods both return the converted strings as new String instances: the original String objects are not modified because String is immutable in Java.

```
        String string = "This is a Random String";

        String upper = string.toUpperCase();

        String lower = string.toLowerCase();
```

```
System.out.println(string); // prints "This is a Random String"

System.out.println(lower); // prints "this is a random string"

System.out.println(upper); // prints "THIS IS A RANDOM STRING"
```

Non-alphabetic characters, such as digits and punctuation marks, are unaffected by these methods. Note that these methods may also incorrectly deal with certain Unicode characters under certain conditions.

```
Scanner scanner = new Scanner(System.in);

System.out.println("Enter the String");

String s = scanner.next();

char[] a = s.toCharArray();

System.out.println("Enter the character you are looking for");

System.out.println(s);

String c = scanner.next();

char d = c.charAt(0);

for (int i = 0; i <= s.length(); i++) {

      if (a[i] == d) {

          if (d >= 'a' && d <= 'z') {

              d -= 32;

       } else if (d >= 'A' && d <= 'Z') {

         d += 32;

       }

         a[i] = d;

       break;

      }

}

s = String.valueOf(a);
System.out.println(s);
```

Finding a String Within Another String

To check whether a particular String a is being contained in a String b or not, we can use the method **String.contains()** with the following syntax:

```
b.contains(a); // Return true if a is contained in b, false otherwise
```

The **String.contains()** method can be used to verify if a CharSequence can be found in the String. The method looks for the String a in the String b in a case-sensitive way.

```
String str1 = "Hello World";
String str2 = "Hello";
String str3 = "helLO";
System.out.println(str1.contains(str2)); //prints true
System.out.println(str1.contains(str3)); //prints false
```

To find the exact position where a String starts within another String, use String.indexOf():

```
String s = "this is a long sentence";

int i = s.indexOf('i'); // the first 'i' in String is at index 2

int j = s.indexOf("long"); // the index of the first occurrence of "long" in s is 10

int k = s.indexOf('z'); // k is -1 because 'z' was not found in String s
int h = s.indexOf("LoNg"); // h is -1 because "LoNg" was not found in String s
```

String pool and heap storage

Like many Java objects, all String instances are created on the heap, even literals. When the JVM finds a String literal that has no equivalent reference in the heap, the JVM creates a corresponding String instance on the heap and it also stores a reference to the newly created String instance in the String pool. Any other references to the same String literal are replaced with the previously created String instance in the heap.

Let's look at the following example:

```
class Strings
{
	public static void main (String[] args)
	{
		String a = "alpha";
		String b = "alpha";
		String c = new String("alpha");

		//All three strings are equivalent
		System.out.println(a.equals(b) && b.equals(c));

		//Although only a and b reference the same heap object
		System.out.println(a == b);
```

```
                System.out.println(a != c);
                System.out.println(b != c);
        }
}
```

When we use double quotes to create a String, it first looks for String with same value in the String pool, if found it just returns the reference else it creates a new String in the pool and then returns the reference. However using new operator, we force String class to create a new String object in heap space.

We can use intern() method to put it into the pool or refer to other String object from string pool having same value. The String pool itself is also created on the heap.

Splitting Strings

You can split a String on a particular delimiting character or a Regular Expression, you can use the String.split() method that has the following signature:

public String[] split(String regex)

Note that delimiting character or regular expression gets removed from the resulting String Array.
Example using delimiting character:

```
String lineFromCsvFile = "Mickey;Bolton;12345;121216";
String[] dataCells = lineFromCsvFile.split(";");
// Result is dataCells = { "Mickey", "Bolton", "12345", "121216"};
```

Example using regular expression:

```
String lineFromInput = "What do you need from me?";
String[] words = lineFromInput.split("\\s+"); // one or more space chars
// Result is words = {"What", "do", "you", "need", "from", "me?"};
```

Joining Strings with a delimiter

An array of strings can be joined using the static method String.join():

```
String[] elements = { "foo", "bar", "foobar" };
String singleString = String.join(" + ", elements);

System.out.println(singleString); // Prints "foo + bar + foobar"
```

Similarly, there's an overloaded String.join() method for Iterables.

To have a fine-grained control over joining, you may use StringJoiner class:

```
StringJoiner sj = new StringJoiner(", ", "[", "]");
        // The last two arguments are optional,
        // they define prefix and suffix for the result string
```

```
sj.add("foo");
sj.add("bar");
sj.add("foobar");

System.out.println(sj); // Prints "[foo, bar, foobar]"
```

To join a stream of strings, you may use the joining collector:

```
Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", "));
System.out.println(joined); // Prints "foo, bar, foobar"

//There's an option to define prefix and suffix here as well:

Stream<String> stringStream = Stream.of("foo", "bar", "foobar");
String joined = stringStream.collect(Collectors.joining(", ", "{", "}"));
System.out.println(joined); // Prints "{foo, bar, foobar}”
```

String concatenation and StringBuilders

String concatenation can be performed using the + operator. For example:

```
String s1 = "a";
String s2 = "b";
String s3 = "c";
String s = s1 + s2 + s3; // abc
```

Normally a compiler implementation will perform the above concatenation using methods involving a `StringBuilder` under the hood. When compiled, the code would look similar to the below:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append("c").toString();
```

`StringBuilder` has several overloaded methods for appending different types, for example, to append an `int` instead of a `String`. For example, an implementation can convert:

```
String s1 = "a";
String s2 = "b";
String s = s1 + s2 + 2; // ab2
```

to the following:

```
StringBuilder sb = new StringBuilder("a");
String s = sb.append("b").append(2).toString();
```

The above examples illustrate a simple concatenation operation that is effectively done in a single place in the code. The concatenation involves a single instance of the `StringBuilder`. In some cases, a concatenation is carried out in a cumulative way such as in a loop:

In such cases, the compiler optimization is usually not applied, and each iteration will create a new `StringBuilder` object. This can be optimized by explicitly transforming the code to use a single `StringBuilder`:

```
StringBuilder result = new StringBuilder();
for(int i = 0; i < array.length; i++) {
    result.append(extractElement(array[i]));
```

```
}
return result.toString();
```

A `StringBuilder` will be initialized with an empty space of only 16 characters. If you know in advance that you will be building larger strings, it can be beneficial to initialize it with sufficient size in advance, so that the internal buffer does not need to be resized:

```
StringBuilder buf = new StringBuilder(30); // Default is 16 characters
buf.append("0123456789");
buf.append("0123456789"); // Would cause a reallocation of the internal buffer otherwise
String result = buf.toString(); // Produces a 20-chars copy of the string
```

If you are producing many strings, it is advisable to reuse `StringBuilders`:

```
StringBuilder buf = new StringBuilder(100);
for (int i = 0; i < 100; i++) {
    buf.setLength(0); // Empty buffer
    buf.append("This is line ").append(i).append('\n');
    outputfile.write(buf.toString());
}
```

If (and only if) multiple threads are writing to the *same* buffer, use StringBuffer, which is a **`synchronized`** version of `StringBuilder`. But because usually only a single thread writes to a buffer, it is usually faster to use `StringBuilder` without synchronization.

Using concat() method:

```
String string1 = "Hello ";
String string2 = "world";
String string3 = string1.concat(string2);  // "Hello world"
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in:

```
"My name is ".concat("Buyya");
```

Reversing Strings

There are a couple ways you can reverse a string to make it backwards.

1. StringBuilder/StringBuffer:

```java
String code = "code";
System.out.println(code);

StringBuilder sb = new StringBuilder(code);
code = sb.reverse().toString();

System.out.println(code);
```

2. Char array:

```java
String code = "code";
System.out.println(code);

char[] array = code.toCharArray();
for (int index = 0, mirroredIndex = array.length - 1; index < mirroredIndex; index++,
mirroredIndex--) {
    char temp = array[index];
    array[index] = array[mirroredIndex];
    array[mirroredIndex] = temp;
}

// print reversed
System.out.println(new String(array));
```