2015-01-13

N C D

String Handling in Java

by Jaber Adeeb

Placing some of java's built-in String related methods under the spotlight.

String handling in java

Introduction

In this research we will figure out how to deal with a string in the programming language "**java"**

A **string** is a sequence of characters and that's known for any programmer, a string is implemented as a character array in many programming languages but in java it's implemented as an object which gives advantages to dealing with strings and adds extra features to make string handling more convenient by making methods that cannot be used with character arrays this also provides the ability to construct a string by many ways depending on the user's need.

There are a couple of disadvantages of string handling in java because when creating a string object it cannot be modified so when altering a string another object is created with the new "*string value*" assigned to it, but if you want a modifiable string there's another built-in class made for this purpose called **StringBuffer** and its **String** objects could be modified.

So by now you know what a string is and you have a simple idea –that we'll go deeper inside- how java dealt with it, next we will explain different ways of creating a **String** in java, the built-in methods related to a **String** and **String concatenation** with other data type.

So at the end of this research the reader could hopefully handle a **String** in an easier way when he has to.

The Different String Constructors :

In java the **String** class holds many **constructors** and we're going to talk about the most common amongst them.

Creating an empty String :

```
String s = new String();
```

This creates an instance of the String class with no chars inside, no one is likely to use this, because you need a string that has an initial value using the following constructors :

- ***String(char chars[]) :***

This constructor creates a String with an initial value of a given char array.

Ps : when you see the word string with its first letter capitalized this means we're talking about String objects, if not it's an ordinary character sequence.

Here's an example :

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars);
```

This initializes *s* with the string **"abcdef"** (this gives the created String a value that is the joined chars of the given char array which here are **'a','b','c','d','e','f'** forming the string **"abcdef"** when joined together).

- ***String(char chars[], int start_index, int chars_num)***

This constructor is a "modification" of the previous one; here you can chose which char to start the string with using the second parameter which is an integer and its length using the third parameter that's also an integer.

Here's an example :

```
char chars[] = { 'a', 'b', 'c','d','e','f' };
String s = new String(chars, 2, 3);
```

This initializes *s* with the string **"cde"** (this gives the created String a value that is a string starting with the char holding the index 2 which is **'c'** plus the following two forming the string **"cde"** which has the length of three as given).

Ps : the first element of an array has an index of 0 and not 1.

- ***String(String string_object)***

This one is used to give a String the same value of another.

An example :

```
String s1 = new String("Java");
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
```

Another example :

```
char c[] = {'J', 'a', 'v', 'a'};
```

```
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
```

s1 contains **"Java"** and *s2* has *s1*'s same value So the output for both examples is :

```
Java
Java
```

- ***String(byte ascii_chars[])***

- ***String(byte ascii_chars, int start_index, int chars_num)***

This constructor is used to initialize a String with a value of the byte array's components after they are converted to chars; what actually happens is each of the bytes represents if right to say an ascii value of a specific char so that's how the conversion happens.

An example might explain better :

```
byte ascii[] = {65, 66, 67, 68, 69, 70 };
String s1 = new String(ascii);
System.out.println(s1);
String s2 = new String(ascii, 2, 3);
System.out.println(s2);
```

Here *ascii* is a byte array and *s1* contains its byte sequence after being converted to a char sequence so each component is converted to a matching char for example 65 is converted to **'A'** and so on, that means *s1* holds **"ABCDEF"** within it and *s2* holds *"CDE"*, that makes the output :

ABCDEF
CDE

The constructors listed above are the most common ones for a java programmer, so hopefully they were clear enough.

Dealing With a String:

Now we're going to explore how to deal with a String and not to create one.

Strings are an important part of any programming language, java took care of making it more easier to handle them, either by the **automatic creation** of a string that's provided when using the **string literals** (the **" "**) or by the **concatenation** of multiple Strings using the **+** operator and last but not least the **conversion** of other data types to a string representation. Java made it easier on its users to perform such operations allowing them to be done automatically so to add clarity.[1]

Ps : there are explicit methods available to perform all of these functions listed above.

- ***String Literals :***

Most of the examples I've listed above show how to explicitly create a String instance from an array of chars by using the new operator, but there's an easy way to do so using the string literal. Java automatically interprets almost anything in a string literal as an object of type String (constructs the String object when encountered by a string literal)

An example might speak for it better :

```
char chars[] = { 'a', 'b', 'c' };
```

[1] Introduction to programming using java, by David J. Eck.

```
String s1 = new String(chars);
String s2 = "abc";
```

Both *s1* and *s2* are initialized in two different ways but same values (*s1* from a char array & *s2* using literals), so *s1* and *s2* are equivalent.

Knowing that for each string literal a String object is created, the user should know that he could use a string literal any place a String object is used, so he can call methods directly on a quoted string like it's an object reference.

- ***String concatenation :***

Java doesn't allow any operators to be applied on a String except for the + operator, which is able to concatenate two Strings producing one String object as a result.

This allows the user to chain a series of + operators together.

An example :

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This example's output is :

```
He is 9 years old.
```

This concatenation is most often used on a long string by breaking it into smaller piece to prevent its **"wrap around"** within a source code.

- ***String concatenation with other data types :***

Java allows concatenating strings with other data types such as an integer.

Here's an example :

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

In this example *age* is an int and not a String but the output is the same :

```
He is 9 years old.
```

And that's caused by the automatic conversion applied on *age* converting it into its string representation within a String object.

The String is afterwards concatenated as before, meaning that the compiler will convert an operand to its string equivalent whenever the other operand of the + is a String.

The latter feature has its disadvantages though, still, it's not a big problem if the user pays little attention not to mix other types of operations with string concatenation expressions or else he might find some surprising results such as the following example :

```
String s1 = "four = " + 2 + 2;
System.out.println(s1);
```

The output is going to be :

```
Four = 22
```

Unlike the user's supposed output (Four = 4), one, the reader might have also expected too, but the operator precedence caused the concatenation of **"Four ="** with the string equivalent of 2 to take place first.

This result is then concatenated with the string equivalent of 2 another time.

But this is easily avoided simply if the user did the following :

```
String s1 = "four = " + (2 + 2);
```

This gives priority to what's inside the parentheses, which is adding the two integer values inside then concatenating the string **"Four = "** with the result, giving the wanted output :

```
Four = 4
```

Built-in methods associated with String objects :

There are many methods used in java to deal with a String (search it, get its properties, extract a specific part of it...) and in this chapter we're going over most of them.

Fetching String properties methods :

- ***String Length :***

when we say String length we mean the number of characters that compose a specific String.

To know the length of a String all we have to do is to call the length() method on that String.

The latter is a method that returns a value of type integer (that's the number of chars in the given String) and takes no parameters.

An example of calling this method :

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

The output will be :

```
3
```

- ***The toString() method :***

This is one of the most common string associated method, because once overridden each programmer could define his own readable version of an object of classes created by him/her.

This method is implemented by every class because it's defined by the if right to say "greatest father" of all classes in java, Object.

This method's general form is :

String toString()

To implement this method all you have to do is to return a String object with your own human-readable version that appropriately describes an object of your class.

When the toString() method is overridden for a class created by a programmer he/she allows this class to be fully integrated into java's programming environment, so they could be used in print() or println() statements and in concatenation expressions.[2]

Here's a little program to explain this more :

```
class Box
{
        double width;
        double height;
        double depth;
        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }
        // here starts our overridden version of toString() :
        public String toString()
```

[2] Learning java, by Pat Niemeyer, Jonathan Knudson.

```
        {
                return "Dimensions are " + width + " by " +
                depth + " by " + height + ".";
        }
}

class toStringDemo
{
        public static void main(String args[])
        {
                Box b = new Box(10, 12, 14);
                String s = "Box b: " + b; // here the overridden
                toString method has been called automatically
                System.out.println(b); //the same thing happened
                here
                System.out.println(s);
        }
}
```

The output is going to be :

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

So the ***Box***'s toString() is automatically invoked when a ***Box*** object is used in a concatenation expression or in a call to println().

Char Extraction Methods :

Java provides many ways to deal not just with a String a whole but with parts of it, and we're going to view most of them.

Characters comprising a string within a String object cannot be indexed like they were a character array, but java provided many methods that offer an index into the string for their operation.

Ps : the String indexes begin with a value of 0, just like an array.

- ***The charAt() method :***

It is used to obtain one single character from a given String, so you can refer directly to this char.

Its general form is :

Char charAt(int index)

Where *index* is the index of the wanted character, and it shouldn't be negative or out of the given string's bounds.

An example :

```
char ch;
ch = "abc".charAt(1);
```

Here the method returned the value **'b'** and it was then assigned to *ch*.

- ***the getChar() method :***

This method is used to extract more than one character from a String.

Its general form is :

Void getChars(int start_index, int end_index, char target[], int target_start)

start_index defines the index where the extraction begins at.

end_index defines the index where the extraction ends plus one.

So the substring contains chars that hold indexes from *start_index* until *end_index* – 1 and will be copied to a characters array which is target starting at an index that is *target_start*.

Ps : to avoid some problems make sure the target array is large enough to hold the substring within.

An example might explain better :

```
class getCharsDemo
{
        public static void main(String args[])
        {
                String s = "hello everybody";
                int start = 7;
                int end = 12;
                char target[] = new char[end - start];
                s.getChars(start, end, target, 0);
                System.out.println(target);
        }
}
```

so if we apply what we said back, the output will be :

```
every
```

- ***The getBytes() method :***

It is an alternative to the getChars() method, the only difference here is that this method stores the characters in a bytes array.

The method uses the default character-to-byte conversion given by the used platform.

Its simplest form is :

Byte[] getbytes()

There are other forms of this method too.

There are many uses related to this method the most common one is exporting a String value into an environment that does not support 16-bit Unicode characters.

- ***The toCharArray() method :***

This one is used to convert all the characters in a String into a characters array easily.

Its general form is :

Char[] toCharArray()

This method is not much of a difference compared to the getChars() because both of them can be used for the same purpose with more ability provided by the getChars() method.

String searching methods :

There are two methods used to search a String for a specific character or substring, which are :

- ***indexOf()***
- ***lastIndexOf()***

the first one searches for the first occurrence of a character or a substring, while the second one searches for the last.

These methods are overloaded in many different ways (has many forms), but in all cases they return the index at which the character or substring was found, or if not they return -1.

To search for the first occurrence of a character or substring :

Int indexOf(int ch)

Int indexOf(String str)

And for the last occurrence of a character or a substring :

Int lastIndexOf(int ch)

Int lastIndexOf(String str)

Where *str* and *ch* specifies the substring or char to be searched for.

There's a form to these methods where you can specify an index to be a starting point to begin the search by passing another parameter to the methods :

Int indexOf(int ch, int start_index)

Int indexOf(String str, int start_index)

Int lastIndexOf(int ch, int start_index)

Int lastIndexOf(String str, start_index)

indexOf() runs either from zero or from the specified index to the end of the string.

lastIndexOf() runs either from the end of the string or from the specified index to zero.

The next example is a little bit long but it should explain better :

```
String s = "Now is the time for all good men " + "to come to the
aid of their country.";
System.out.println(s); System.out.println("indexOf(t) = " +
s.indexOf('t'));
System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
System.out.println("indexOf(the) = " + s.indexOf("the"));
System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the",
60));
```

The output will be :

```
Now is the time for all good men to come to the aid of their
country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

String modifying methods :

as we previously said Strings in java are **immutable** (can't be modified),but java provided a couple of ways to overcome this problem, the first is the **StringBuffer** class[3] and the other way is

[3] Teach yourself java in 21 days, by Laura Lemay, Charles L. Perkins.

using methods provided to modify a String without creating a new String object.

- ***Substring()***

This one has two forms and its uses are to extract a substring.

Its first form is :

String substring(int start_index)

Where *start_index* is the index defining the beginning of the extraction.

This form returns a substring starting from the char with the given index to the end of the given String.

The other form gives the ability to define the ending index :

String substring(int start_index, int end_index)

Here the substring begins at the char with the index *start_index* and ends at the char with the index *end_index* – 1 .

An example maybe…

```
class StringReplace
{
        public static void main(String args[])
        {
                String org = "This is a test. This is, too.";
                String search = "is";
                String sub = "was";
                String result = "";
```

```
                int i;
                do
                {
                        System.out.println(org);
                        i = org.indexOf(search);
                        if(i != -1)
                        {
                                result = org.substring(0, i);
                                result = result + sub;
                                result = result + org.substring(i +
                                search.length());
                                org = result;
                        }
                } while(i != -1);
        }
}
```

The output for this example will be :

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

- ***Concat()***

This method allows you to concatenate two strings together.

Its general form is :

String concat(String str)

This method creates a new object that contains the invoking string with the contents of *str* added to the end, so it works just like the + operator.

An example :

```
String s1 = "one";
String s2 = s1.concat("two");
```

So this adds *s2* to *s1*, so *s2* has a value of **"onetwo"**

And it's just like the following :

```
String s1 = "one";
String s2 = s1 + "two";
```

- ***Replace()***

This one replaces all of the occurrences of a character in the given string with another character.

It has the following general form :

String replace(char original, char replacement)

Here, *original* specifies the character to be replaced by the character specified by *replacement*, then the resulting string is returned.

Example :

```
String s = "Hello".replace('l', 'w');
```

In this example s is given a value of **"Hewwo"**.

- ***Trim()***

this is the last method to be examined in this category.

It is used to return a copy of the given string from which any leading and trailing whitespaces has been removed.

Its general form is :

String trim()

An example :

```
String s = "   Hello World    ".trim();
```

this gets the string **"Hello World"** contained in *s*.

This method is quite useful when you process user commands, because a user can unintentionally leave some whitespaces when typing some commands.

conclusion

Java added many methods to deal with a String (knowing its length or modifying it…etc) because it is a very important if right to say "type"; it is used in almost every language and is strongly needed in most programs.

The ways java made this happen vary from the many ways to construct a String to built-in methods used to modify a String since it's immutable and last but not least giving the ability to concatenate String with each other and with other data types.

I have not covered every way or method java added, but as we have seen I've hopefully covered many features that are commonly known amongst every java programmer, these additions made dealing with Strings a piece of cake to almost any user at any level.

references :

Java 2: The complete reference, fifth edition, By Herbert Schildt.

Introduction to programming using java, by David J. Eck.

Getting started with java (version 8)