Strings in Java and .NET

Venkat Subramaniam
venkats@durasoftcorp.com
http://www.durasoftcorp.com/download

Abstract

Strings are one of the most basic data types in a language, yet they have been treated very differently in almost all the languages. Even Java developers with reasonable experience make some typical mistakes in this area. This article talks about working with Strings in Java and .NET languages. We present here things that we need to be aware of while working with Strings in Java. We also discuss how some of these hold true and some don't when it comes to Strings in .NET.

Common Mistake

Here is a piece of code some one brought to me and asked why it does not work. This is a developer, in my opinion, with reasonable Java experience, and his code simply compares the user entered password to what was obtained from the database:

```
boolean validate(String password)
{
     String correctPassword = …
          //code to get from the database goes here.

     if (password == correctPassword)
          return true;
     else
          return false;
}
```

Even if the correct password is entered by the user, this validate method returns a false.

We modified the comparison as follows to fix the problem:

```
     if (password.equals(correctPassword))
```

What is more interesting is the following code may return true or false depending on how it is invoked:

```
boolean validate2(String password)
{
     if (password == "hello") return true;
     return false;
}
```

It would return a true if you call it as validate2("hello") and false if you call it as validate2(new String("hello")). At the very least, this leads to confusion while coding. Before we analyze what just happened, let us understand some fundamentals on Strings like immutable objects and data types.

Immutable Objects

An immutable object[1] is an object whose state can't be changed. Immutable objects provide a significant advantage. They may be shared as a whole or their internals may be shared. Immutable objects do not require any synchronization and are inherently thread-safe. In order to guarantee immutability, the immutable class should not provide any method that would modify the state of an object. This also requires that the methods of the class are not overridable. There are at least two ways to achieve this. One is to make the class final and the other is to make all the constructors of the class private (and provide static factory methods to obtain the object). Joshua[1] says the following about using private constructors, "While this approach is not commonly used, it is often the best of the … alternatives." There are a number of examples of immutable objects in Java: Class, BigInteger, Integer, Boolean, and String to mention a few.

Immutability of String

The java.lang.String class is an immutable class. None of the 50 plus methods of the String class allow the modification of an instance once it is created. However, Java enforces immutability by making the String class final and not by making the constructors private. The immutable nature of String allows for sharing of an object in an application, which is the reason for making it immutable in the first place. However, the fact that the constructor is not private, allows one to create as many objects of String class as one desires. As a result, Java strives to share the internals of the string rather than the string object itself. This however, leads to the problem mentioned in the beginning of this article.

String as a Data Type

Primitive data types like int, double and char are well behaved. They are defined as built-in types in most languages including Java. By their nature, Strings are as primitive as the other types. The one significant difference between Strings and other primitive types is their variable length. While all other primitive types have fixed length (int is always 4 bytes), the length of String varies. This makes it difficult to treat it as a built-in type. Languages like C++ treat Strings as objects as much as any other user defined data types are treated. Java, however, tries to provide a built-in view of Strings even though they are classes. For instance, Java does not allow user defined operator overloading on any class. However, the operator + is overloaded and defined as concatenation operator on String class. Java has made significant effort to make Strings look similar to built-in primitive types and different from user defined classes.

For instance, with a built-in type like int, you can write

```
int val1 = 2;
```

However, in Java, you can't write

```
int ref = new int(2); // Error.
```

Similarly, in Java, you can write

UDTClass ref = new UDTClass(…);

However, you can't write

UDTClass ref = 2; // Error.

When it comes to String, however, both of the above are allowed. Consider the statements shown below in Figure 1.

```
String ref1 = new String("hello");
String ref2 = new String("hello");
String ref3 = "hello";
String ref4 = "hello";
```

Figure 1. Different ways to create String objects

Let's first look at two variations:
String ref1 = new String("hello");
String ref2 = "hello";

While this looks neat, there is a fundamental difference between using new and not using new when it comes to creating objects of String.

String ref1 = new String("hello"); results in creation of a new String object.
String ref2 = new String("hello"); results in creation of yet another String object. The internals of the two string objects are shared, but you still have two string objects.

String ref3 = "hello"; results in creation of a third String object.
Now, String ref4 = "hello"; actually ends up sharing the same String object that ref3 is referring to!

Figure 2 (below) illustrates how these four references shown in Figure 1 are related.

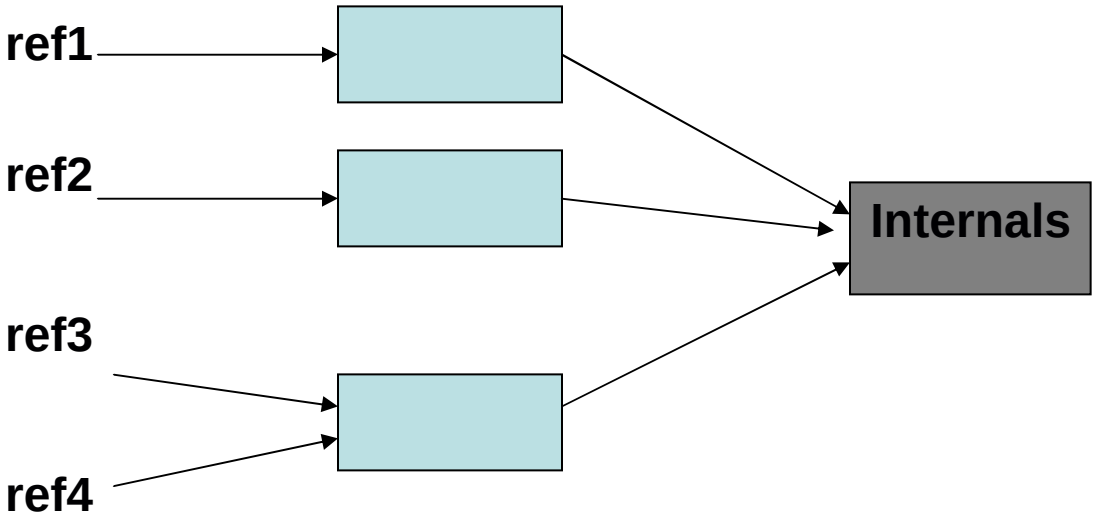

Figure 2. Effect of the four statements shown in Figure 1

The Inequality of Equality

The assignment operator (=) and the concatenate operator (+) on Strings show how Strings are meant to be used in ways similar to built-in types. Unfortunately, the similarity stops there. The equality operator "==" compares the identity of objects for user defined data types and compares values for built-in types. In the case of Strings, however, they compare the identity as well. As a result, looking at the example in Figure 2, ref1 == ref2 is false while ref3 == ref4 is true. However, ref1.equals(ref2) is true. In fact, in the above example, ref1.equals(x) is true, where x is any of the four references.

So, when comparing Strings, it is important to use the equals method and not the "==" equality operator.

How about Strings in .NET?

In .NET, String belongs to the Common Type System (CTS) and is part of the System namespace. In C#, there is an alias "string" for the System.String class. The constructors of the String class are not private in .NET. However, there is no constructor that will accept another String as argument. As a result, the following statement is not valid in C#:

```
string ref1 = new string("hello"); // Error in C#
```

String in .NET is sealed (same as final in Java). Further, the equality operator == is overloaded on String correctly. So, if I want to compare strings, I can either use the equals method (why would I) or better still use the == operator. So, in .NET, it is perfectly ok to write statements like `if (password == correctPassword)`.

Performance of Concatenation

One of the disadvantages of String immutability is poor performance of concatenation[1]. When you concatenate two strings, copies of both the strings have to be created and there is no space for optimization. It is ok to use concatenation on very small fixed size strings and when the concatenation is done very rarely. However, if the size is arbitrary or if the

concatenation happens more frequently, then it is not the best option. Figure 3 below shows a piece of Java code that uses the concatenate operator and the StringBuffer code.

```
import java.io.*;

class Test
{
     public static void main(String[] args)
     {
          int iterations = Integer.parseInt(args[0]);
          long start, end;
          if (args[1].equals("efficient"))
          {
               StringBuffer buffer = new StringBuffer();
               start = System.currentTimeMillis();
               for (int i = 0; i < iterations; i++)
               {
                    buffer.append(i);
               }
               end = System.currentTimeMillis();
          }
          else
          {
               String str = "";
               start = System.currentTimeMillis();
               for (int i = 0; i < iterations; i++)
               {
                    str += i;
               }
               end = System.currentTimeMillis();
          }

          System.out.println(end - start);
     }
}
```

Figure 3. Java code that uses + and StringBuffer

Figure 4 which is presented below shows a piece of C# code that uses the concatenate operator and the StringBuilder.

```
using System;

namespace Sample
{
     class Test
     {
          [STAThread]
          static void Main(string[] args)
          {
               int iterations = Convert.ToInt32(args[0]);
               int start, end;
               if (args[1] == "efficient")
```

```
                {
                        System.Text.StringBuilder builder =
                                new System.Text.StringBuilder();
                        start = Environment.TickCount;
                        for (int i = 0; i < iterations; i++)
                        {
                                builder.Append(i);
                        }
                        end = Environment.TickCount;
                }
                else
                {
                        String str = "";
                        start = Environment.TickCount;
                        for (int i = 0; i < iterations; i++)
                        {
                                str += i;
                        }
                        end = Environment.TickCount;
                }

                Console.WriteLine("{0:N}", end - start);
        }
    }
}
```

Figure 4. C# code that uses + and StringBuilder

Figure 5 shows the performance of the code in Figure 3 on Java JDK 1.4.1.01 and that of code in Figure 4 on .NET v1.0.3705.

Iterations	Java +	Java StringBuffer	C# +	C# StringBuilder
10	0	0	0	0
100	0	0	0	0
1000	90	10	10	0
10000	19218	50	2333	20

Figure 5. Time in milliseconds for using + vs. Buffer/Builder in the sample code in Figures 4 and 5.

Conclusion

In this article we have discussed the issues with using Strings in Java and .NET. In Java, do not use the operator== to compare Strings. Instead use the equals method. However, in .NET, it is ok to use the == operator. In both the languages, it is better not to use the concatenate operator+. Instead use the StringBuffer in Java and StringBuilder in .NET.

References

1. Joshua Block, *Effective Java Programming Language Guide*. Addison-Wesley, Boston, MA, 2001. ISBN:0-201-31005-8.
2. MSDN http://msdn.microsoft.com.