

String Optimization – Best Practice Guide

In general, you should pay more attention to how you handle strings than nearly any other portion of your application – sometimes more than security. I have seen proper string optimizations increase an applications performance by more than 90% without any other changes being made. The following tips should be used as a guide. No single tip can apply to every single situation.

Empty Strings

Never create an empty string implicitly. If a string object needs to be empty upon dimming, explicitly set it to empty, but do use the quotes to do so. You will save the compiler a step if you use the `String.Empty` method.

```
Dim strString As String = String.Empty
```

Please Note: *This tip can be extended to pretty much any object, even if it needs to be dimmed as Nothing.*

String Concatenation

You should NEVER use the standard “&” or “+” operators to perform string concatenation. Every time you do, you decrease your applications performance tremendously. Instead use the `String.Concat` method or `StringBuilder` class to concatenate strings.

String.Format

When you need to have a quick and easy way to inject a string value into an existing string a limited or low number of times, the `String.Format` method will be useful. However, the more you use it in succession, the slower the concatenation will occur. Therefore, this is not recommended for loops.

```
Dim strObject As String = _  
    String.Format("This is the first object:{0}<br />", arg1)
```

String.Concat

If you have a simple concatenation to make or less than 15 or so concatenations on the same string object, use the `String.Concat` method to marry string objects. This can be done like so:

```
Dim strObject As String = String.Empty  
strObject = String.Concat("This is the first object: ", _  
    arg1, "<br />", _  
    "This is the second object: ", arg2, "<br />", _  
    "This is the third object: ", arg3, "<br />", _  
    "This is the fourth object: ", arg4, "<br />", _  
    "This is the fifth object: ", arg5, "<br />")
```

StringBuilder

If your string concatenation is complicated, or has more than 15 or so concatenations, use the `StringBuilder` class. While the `StringBuilder` can have its own problems, it is generally the fastest string concatenation practice. However, its performance can be greatly enhanced by instantiating it with the known size that it will be (when possible).

```
Dim sb As New System.Text.StringBuilder
sb.Append("This is the first object")
sb.Append(arg1)
sb.Append("<br />This is the second object:")
sb.Append(arg2)
sb.Append("<br />This is the third object")
sb.Append(arg3)
sb.Append("<br />This is the fourth object")
sb.Append(arg4)
Dim strObject As String = sb.ToString
```

The previous example could have been further optimized by specifying a size when instantiating the `StringBuilder` object. This should only be done when you know the approximate size of the final string.

```
Dim sb As New System.Text.StringBuilder(100)
```

Sometimes the `StringBuilder` does not appear to meet your needs, or it may be easier to kind of bend the rules by performing inline string concatenation with it. Such as:

```
sb.Append("This is the first object:" & arg1 & "<br />")
```

... Or ...

```
sb.Append(String.Concat("This is the first object:", arg1, "<br />"))
```

The previous snippets are not recommended. This is actually achieved by the following example:

```
sb.AppendFormat("This is the first object:{0}<br />", arg1)
```

String Evaluation

This is the second most common string optimization mistake. The first was the concatenation operators. You should not use the standard comparison operators to evaluate strings. Instead, use the `CompareTo`, `Equals`, `IsNullOrEmpty`, or `Regex.IsMatch` methods.

```
If strObject.CompareTo("somevalue") > -1 Then
    ' the string is found
End If
```



```
If RegEx.IsMatch(strObject, "[regexexpression]") Then
    ' the values match
End If
```

```
If String.Equals(strObject, "somevalue") Then
    ' the values match
End If
```

```
If Not String.IsNullOrEmpty(strObject) Then
    ' the string object has value
End If
```

Built-In String Methods

There are useful string methods that perform common tasks. These methods allow you to do things like change all of the text to lower case or upper case, replace text, etc. They should only be used when absolutely necessary, as there are likely better performing alternatives. (Examples given in the preceding statement are: `ToLower`, `ToUpper`, `Replace`, `Contains`, `EndsWith`, `Trim`, etc.)

This is not said blindly. For instance, if you call the following line of code:

```
Dim strReturn As String = strObject.Trim
```

There are not *two* string objects here. There are *three*! The call to the `Trim` method actually creates a separate string object in memory that we never know about. If you use these methods regularly, think about the number of extra string objects created in memory that you never knew existed.

String Replacement

Replacing sections of a specific string is as easy as calling the `Replace` method from the object itself. However, this is not the most effective way to perform this task. You should instead use the `Regex` class to replace string values.

```
Dim strReplace As String = "A sample string"
Return RegEx.Replace(strReplace, "sample", "REALLY COOL")
' returns "A REALLY COOL string"
```

Obviously, this is a very simple example. Depending on what you need to replace, the regular expression can be much more complicated than "sample." It is recommended that you invest in a tool such as [RegexBuddy](#) to assist you in your regular expressions. If you want a free regular expression tool, you can also try [The Regulator](#).

Strings in Memory

Very often, we are quick to specify an object that requires a string identifier by just entering the string value into its calling function.



```
Dim strString As String = Request.Params.Item("key")
```

If the code block that examples like belong to are highly trafficked, this is better achieved by have the string key object shared as a constant.

```
Private Const c_KeyName As String = "key"
```

Then, instead of a new string object being created each time that line of code being executed, the constant is loaded into memory, resulting in faster code. This also allows you to build entire classes around common string objects such as querystring keys, data column names, cache/session keys, etc.

Caching

When you have optimized all that you can, it is time to take a look at caching string objects. Cache all of the string objects that are generated dynamically, but are rarely changed. A good example would be a static query that is generated at application start, or a generated JavaScript code block. Send it to cache, and then retrieve it from memory instead of generating the string object again.