StringBuffer versus String

What is the performance impact of the StringBuffer and String classes?

By Reggie Hutcherson, JavaWorld.com, 03/24/00

Java provides the `StringBuffer` and `String` classes, and the `String` class is used to manipulate character strings that cannot be changed. Simply stated, objects of type `String` are read only and immutable. The `StringBuffer` class is used to represent characters that can be modified.

The significant performance difference between these two classes is that `StringBuffer` is faster than `String` when performing simple concatenations. In `String` manipulation code, character strings are routinely concatenated. Using the `String` class, concatenations are typically performed as follows:

```
String str = new String ("Stanford  ");
str += "Lost!!";
```

If you were to use `StringBuffer` to perform the same concatenation, you would need code that looks like this:

```
StringBuffer str = new StringBuffer ("Stanford ");
str.append("Lost!!");
```

Developers usually assume that the first example above is more efficient because they think that the second example, which uses the `append` method for concatenation, is more costly than the first example, which uses the `+` operator to concatenate two `String` objects.

The `+` operator appears innocent, but the code generated produces some surprises. Using a `StringBuffer` for concatenation can in fact produce code that is significantly faster than using a `String`. To discover why this is the case, we must examine the generated bytecode from our two examples. The bytecode for the example using `String` looks like this:

```
0 new #7 <Class java.lang.String>
3 dup
4 ldc #2 <String "Stanford ">
6 invokespecial #12 <Method java.lang.String(java.lang.String)>
9 astore_1
10 new #8 <Class java.lang.StringBuffer>
13 dup
14 aload_1
15 invokestatic #23 <Method java.lang.String valueOf(java.lang.Object)>
18 invokespecial #13 <Method java.lang.StringBuffer(java.lang.String)>
21 ldc #1 <String "Lost!!">
23 invokevirtual #15 <Method java.lang.StringBuffer append(java.lang.String)>
26 invokevirtual #22 <Method java.lang.String toString()>
29 astore_1
```

The bytecode at locations 0 through 9 is executed for the first line of code, namely:

```
String str = new String("Stanford ");
```

Then, the bytecode at location 10 through 29 is executed for the concatenation:

```
str += "Lost!!";
```

Things get interesting here. The bytecode generated for the concatenation creates a `StringBuffer` object, then invokes its `append` method: the temporary `StringBuffer` object is created at location 10, and its `append` method is called at location 23. Because the `String` class is immutable, a `StringBuffer` must be used for concatenation.

After the concatenation is performed on the `StringBuffer` object, it must be converted back into a `String`. This is done with the call to the `toString` method at location 26. This method creates a new `String` object from the temporary `StringBuffer` object. The creation of this temporary `StringBuffer` object and its subsequent conversion back into a `String` object are very expensive.

In summary, the two lines of code above result in the creation of three objects:

1. A `String` object at location 0
2. A `StringBuffer` object at location 10
3. A `String` object at location 26

Now, let's look at the bytecode generated for the example using `StringBuffer`:

```
0 new #8 <Class java.lang.StringBuffer>
3 dup
4 ldc #2 <String "Stanford ">
6 invokespecial #13 <Method java.lang.StringBuffer(java.lang.String)>
```

```
9 astore_1
10 aload_1
11 ldc #1 <String "Lost!!">
13 invokevirtual #15 <Method java.lang.StringBuffer append(java.lang.String)>
16 pop
```

The bytecode at locations 0 to 9 is executed for the first line of code:

```
StringBuffer str = new StringBuffer("Stanford ");
```

The bytecode at location 10 to 16 is then executed for the concatenation:

```
str.append("Lost!!");
```

Notice that, as is the case in the first example, this code invokes the `append` method of a `StringBuffer` object. Unlike the first example, however, there is no need to create a temporary `StringBuffer` and then convert it into a `String` object. This code creates only one object, the `StringBuffer`, at location 0.

In conclusion, `StringBuffer` concatenation is significantly faster than `String` concatenation. Obviously, `StringBuffers` should be used in this type of operation when possible. If the functionality of the `String` class is desired, consider using a `StringBuffer` for concatenation and then performing one conversion to `String`.

About the author

Reggie Hutcherson is a Sun technology evangelist. He evangelizes Sun's Java 2 Platform technologies around the world concentrating on J2SE and the HotSpot performance engine.