Hidden Traps: String Mutations

Posted on **April 2, 2016**

In the world of Sci-Fi movies and video games, mutants are not always a bad thing. In fact, sometimes, the mutants are the hero or the key to the hero's success. On the other hand, no matter how good their intentions, trouble always seems to follow them.

In the world of programming, data can be mutated as well and it attracts trouble there too. Data is mutated when its internal state is modified. The alternative is data that is immutable. This means that the internal value cannot be modified, it can only be replaced with a new value.

Let's see an example of immutable data in action first:

```
a=42
b=a
a+= 4
puts a   #Displays 46
puts b   #Displays 42
```

This is the safe, sane case. Changes to the variable a do not affect the variable b. Now, in Ruby, strings *are* mutable. Let's see how this plays out.

```
a="hello"
b=a
a << " world"
puts a    #Displays: hello world
puts b    #Also displays: hello world
```

In this case, the string data in a and b was mutated. The change to a was expected. The change to the seemingly unrelated variable b could be a nasty surprise and a source of bugs. I can say that this very issue is without doubt the number one source of trouble in my own code.

The sensible question to ask is: Why are string mutated at all? Why not always protect the programmer from this issue? The answer is performance. Creating new instance of strings all the time is wasteful of resources and places a heavy load on the memory management system. Let's see how this issue is handled by two leading languages: C# and Ruby.

C#

In C#, the mutation of strings issue is handled by giving the programmer choice. The default, goto class for strings, namely the String class, is fully safe and immutable. Thus for simple programming tasks and less experienced programmers, the obvious choice also is the easiest to get right. Maybe not super-fast, but the expected results.

When (and where) performance issues do arise, the StringBuffer class exists that can be more efficient because is allows data mutation. In fact, it embraces it with special mutating methods, not part of the String class. Since a conscious choice must be made to allow and use mutation, hopefully, the programmer will focus there attention on those areas, thus avoiding nasty surprises.

Ruby

In contrast, Ruby (currently) allows string mutation by default. This means that any string at any time could be mutated. This causes a lot of problems. There are some possible answers:

- When assigning a string, clone it. A statement like: *a = b.clone* breaks the connection between a and b by making a full copy of the string. It works, but this can be slow.
- When creating a string, freeze it. This looks like: *b = "Hello".freeze* and now any mutation will raise an error that reveals the bug at the point it first occurs. This is not slow, but requires thorough testing to avoid exceptions in the field. You also need to remember to use the freeze in the first place.
- In upcoming versions of Ruby (version 3 I believe), strings will be frozen by default. You will now need special syntax to create mutable strings. This will look (probably) something like *b = String.new("Hello")*. This suffers from the fact that it is likely to break a lot of older code.

These two languages show two divergent approaches to the issue of string mutation. Now I really like Ruby, and C# is a pretty decent language too. On this issue though, C# looks like the clear winner here. Safe by default, gentle on programmers, efficient where needed, and above all, not breaking older code. To me it's a slam-dunk!

What are your thoughts? Does another language do an even better job? I'd love to hear from you, so chime in with a comment, idea, or objection!

Yours Truly

Peter Camilleri (aka Squidly Jones)

Article Update: April 15, 2016

I am pleased to announce the release of the fOOrth 0.6.0 language system. The major change in this version is to split the String class into an immutable String class and a mutable StringBuffer class. Even though fOOrth is based on Ruby, the underlying architecture is more than flexible enough to accommodate this change.

This entry was posted in **Hidden Traps** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=1681]** .