Sample Markdown Article

Taking stock analysis to the next level

by Marc Zeedar

At a Glance

XD#: 00000

Target Reader: Intermediate

Source Code: Yes

About the Author: Marc taught himself programming in high school when he bought his first computer but had no money for software. He's had fun learning ever since.

Last issue when I wrote about my program to retrieve stock prices, it was my "just get it working" attempt. It wasn't the ultimate stock program I originally envisioned, especially in terms of how the data was presented and what you could do with it.

What I really wanted was a way to write stock analysis scripts: basically mini-programs that could do math based on current stock prices.

Naturally, I could write these in Xojo*, but by nature these are the kinds of things that change frequently, so a XojoScript approach makes a lot more sense. With XojoScript, I'd have all the power of Xojo and I could rewrite my scripts at any time without having to recompile my original program.

> (*I also could use a spreadsheet, but where's the fun in that?)

As I began thinking about just how to do what I wanted, I realized this is not only a tricky problem in several ways, it presents a great opportunity to demonstrate some unusual aspects of XojoScript.

So even if you aren't interested in stock tracking in particular, you may find some techniques here you can use in your own projects. And if you've never tried XojoScripts, you're in for a treat. (XojoScript is by far my favorite feature of Xojo.)

Creating A Stock Language

Now one of my favorite features of Soulver is that it lets me create variables of stock tickers that represent the current value of the stock. So a line like this

```
10 x AAPL
```

produces the current value of 10 shares of Apple stock.

I wanted to do the same thing with my XojoScript solution. Clearly that could easily be done by adding variable names for each stock ticker. However, that means you have to define each stock item manually:

```
dim APPL as string = "AAPL"
```

One problem with the above is that is a string, not a price. You'd then have to convert that to an amount with a function:

```
dim price as double = getPrice(AAPL)
```

This is doable, but it's a lot more coding than we want. It's also not exactly clear that items like AAPL are variables or what kind of variable they are.

For those reasons -- and a couple of others I'll get into in a moment -- I decided it would be wise to precede each stock variable with the letters "st" so APPL would be stAPPL. It's slightly uglier, but it makes it clear that these are stock ticker variables.

To solve the string/number issue, I decided to use a custom object.

It's not widely known, but XojoScripts can define classes just like you can in the Xojo IDE. Here's the XojoScript code to define our stockClass class:

```
class stockClass
  dim shares as double
  dim ticker as string
  dim price as double

  sub constructor(stockTicker as string, sharesHeld as double)
    ticker = stockTicker
    shares = sharesHeld
    price = cachedStockPrice(ticker)
  end sub

  function value() as double
    return price * shares
  end function
end class
```

What this does is give us a storage object for our stock ticker string (ticker) as well as the amount of shares held (shares). We can then pass this object around to various functions in our script(s) and do calculations with this stock's value.

Since this class has a constructor, in order to create a new object of this type, we have to call new with the stock name and amount values:

```
dim stAAPL as new stockClass("AAPL", 100)
```

This line would create a new variable called stAAPL whose ticker property would be "AAPL" and whose amount property would be 100.

You'll note that the constructor calls a special function in our main program called cachedStockPrice which returns the current price of the stock and it sets an internal class property, price, to that value.

This class solves the string/number problem since both are stored in the object and we can work with whichever we want (i.e. stAAPL.price or stAAPL.ticker).

Our class also has a value function, which conveniently returns the total value of all the shares of stock (shares multiplied by price).

Note that because our stock object contains the amount of shares we own, if you want to store *multiple* share amounts you'll have to use multiple objects.

For instance, say you and your spouse each have an IRA, or maybe you yourself have multiple retirement accounts. Each account probably has a different amount of AAPL shares. You'd have to do something like this:

```
dim stAAPL-IRA1 as new stockClass("AAPL", 74.3)
dim stAAPL-IRA2 as new stockClass("AAPL", 47.8)
```

This would allow you to create a script that calculates the total value of each IRA.

A XojoScript Primer

XojoScript is a broad topic and it's a difficult concept to explain to someone who has never used it, but I will do my best. Here are the key features of XojoScript you need to know.

- **Two XojoScripts** It's confusing at first, but there are really two XojoScript components: the XojoScript object, which is an invisible control (like a Timer) you place on a window in your main program, and the actual script (code) which will be executed when the XojoScript is called.

- **Plain text code** Scripts themselves are just plain text code like dim s as string and for i = 1 to 10. You can use all of the core Xojo language and datatypes, but keep in mind that framework and objects don't work, so you can't use things like folderItem or date.

- **Code from anywhere** Since scripts are plain text, they can come from anywhere: a file on disk, code typed into a textArea inside the main program, or even text downloaded from the Internet. This is what gives XojoScript its power, since you can modify external scripts at any time, your program can change what it does without having to recompile it.

- **XojoScripts run inside your main program** When a XojoScript is executed, it's running inside a private bubble within your main

program. This means that by default, a script cannot see properties and functions and objects that exist in your main program. And vice versa -- your main program cannot see what the script is doing.

- **Context means sharing** You can share information between the script and your main program via a *context* object. Basically, whatever functions, methods, or properties the context object has, those are available in your scripts! So while a script can't natively display a dialog, you could write a simple showDialog method in your context and call it from your script. The same thing works in reverse: you could add a property, say name as string to your context object, and then your script could refer to name and there would be no syntax error: name would be a script object your script can set or get.

- **Context extends script abilities** By using the context object, you can actually extend the abilities of scripting to do things that aren't normally supported. For instance, in one of my programs I have created my own picture type object. XojoScript by itself can't work with pictures, but my main Xojo program can, so my script class sends all the picture creating and manipulation stuff back to the main program where all the work is done. This allows my script to "draw" lines, shapes, text, and even other pictures onto a picture object which can then be exported to disk as a JPEG file.

Figure 1: The xDev Logo.

Figure 2: An xDev Magazine cover.

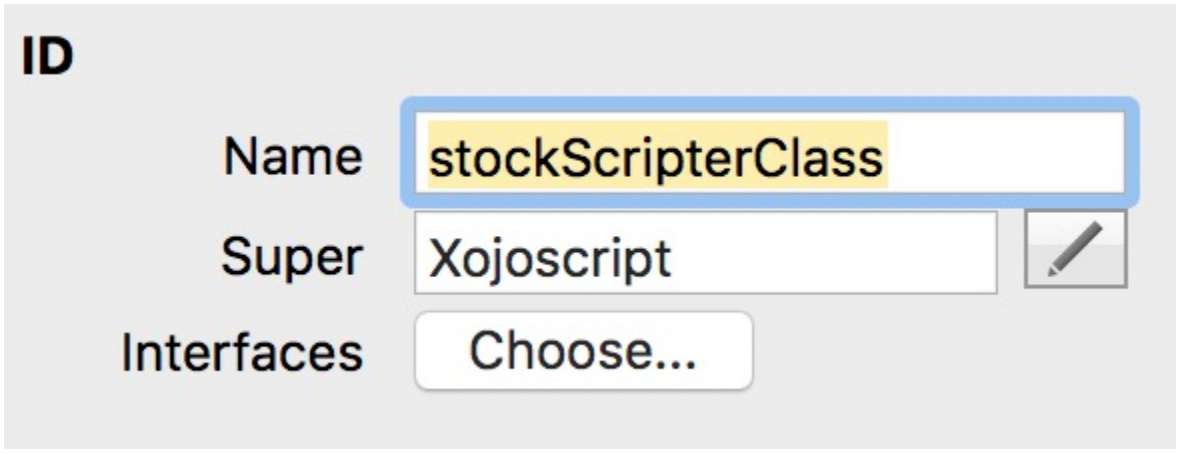

Figure 3: Creating a subclass of XojoScript.

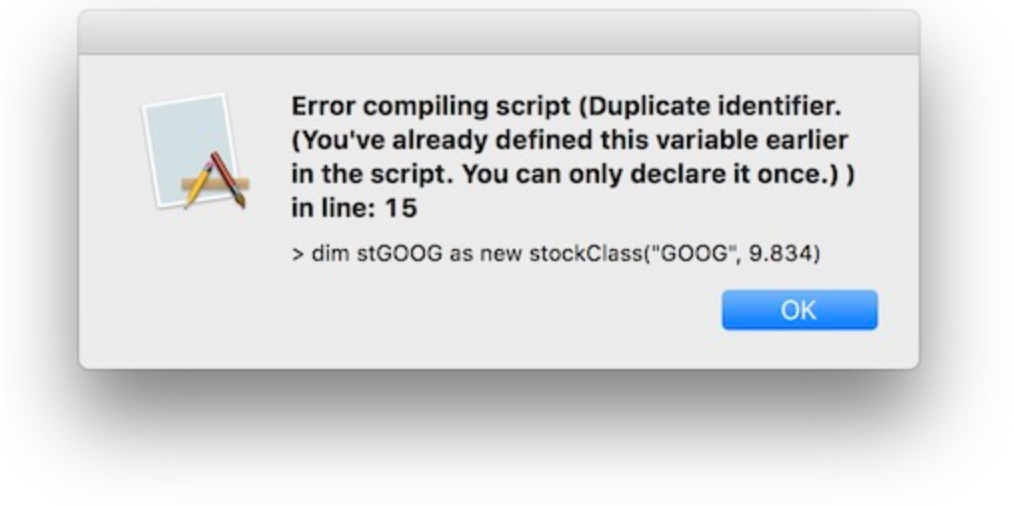

Figure 4: A sample error message when a script can't be compiled.