Tcl/Tk is a nice little language, but nothing particularly special by itself. What makes Tcl/Tk exceptional is how easily it can be extended. The easy-to-extend nature of the Tcl interpreter provides a playground for experimentation and has produced lots of useful extensions.

There are two types of Tcl extensions, ones that add totally new functionality to Tcl/Tk (as the sybtcl and oratcl extensions add database commands to Tcl) and ones that enhance existing Tcl/Tk functionality.

These extensions give the Tcl community a chance to play with new ideas, and then pester the folks at Scriptics to roll the best ones into the Tcl kernel.

One of my favorite extensions is BLT, written by George Howlett. This extension enhances the Tcl base functionality with a new geometry manager, a new data type, some enhancements to the canvas, and several new widgets including graphing and tree display widgets.

Some of the more generic concepts introduced in BLT (the table driven geometry manager, for instance) have been merged into the Tcl core. Others, (like the graph widget) are sufficiently special purpose that it makes more sense to leave them in an extension, and only load that code when you need to use it.

Which brings us back to the HTTP stock robot from the last Tclsh Spot article, and some do-it-yourself historical data viewing.

The previous articles developed a little robot that will query `newsalert.com` and get prices and volume for a list of stocks. This article will discuss saving and graphing the data the robot returns.

The robot code looks like this:

```
package require http


puts [format \
   {%-5s %-9s %-7s %-5s %-6s %-9s %-9s %-9s %-8s}\
  symb last change pct date open high low volume]


foreach symbol $argv {
    set url "http://www.newsalert.com/free/stocknews?Symbol=$symbol"


    set id [::http::geturl $url]
    set page [::http::data $id]


    regexp -expanded "arts.Symbol=(.+?)\">  # Get the symbol
                  (.+?)<tr>"      $page m symb data
    regsub -all {<.+?>} $data {} lines
    set dataList [split [string trim $lines] \n]


    puts [eval format \
      {{%-5s %-9s %-7s %-5s %-6s %-9s %-9s %-9s %-8s}} \
      $symbol $dataList]
 }
```

To do some historical analysis, we need to collect some history. The simple way to do this is to save data as we read it, which just involves opening a file, and writing data.

Tcl I/O commands follow the familiar convention of creating a handle to access the data stream. This handle (called a `channel` in Tcl) may be used to access a file, device, pipe to another application, or a socket. A channel to a file, device or pipe is created with the `open` command.

This is the syntax for the `open` command:

Syntax: open *streamName ?access? ?permissions?*

streamName	By default, the name of a file to open. If the first character of the *streamName* is a pipe symbol "|", then the rest of the name is a program to run attached to a pipe.
?access?	The access method: "r" for read, "w" for write, "a" for append. Or a list of POSIX mnemonics including `RDONLY WRONLY RDWR APPEND CREAT EXCL NOCTTY NONBLOCK TRUNC` The default is "r" (`RDONLY`).
?permissions?	When a file is created, you can declare the permissions mask in numeric form. Tcl supports octal numbers, allowing you to set the modes to values like 0666.

We can use `"a+"` for the *access* parameter, to either create a new file, or add data to an existing file.

Adding a command like this before the loop will open a channel that data can be written to:

```
set outfile [open "stock.data" "a+"]
```

The `puts` command can be used to either send data to the `stdout` device (as the robot does with the formatted data) or to a specific channel (if the first argument is a channel identifier).

Putting this line inside the loop will write data to the historical data file:

```
puts $outfile "$symbol $dataList"
```

This will generate lines like this:

```
SUNW {75 1/16} {-1 7/8} -2.4 1/28 {77 1/8} {79 1/2} {73 27/32} 19,016
```

which are almost useful.

History without dates is even worse than a Friday nights without dates.

All operating systems provide a hook for getting the current time. When Tcl/Tk was ported to the Mac and Windows platforms one problem that needed to be addressed was the different ways systems represent time and date.

The Sun Tcl development team solved this problem by adapting the time and date

commands from TclX to generalize access to the underlying time/date representation. The clock command provides access to the system specific time/date, and can also convert from human-readable format to system specific format and back. For example, you can use the clock command to retrieve the system specific representation of a time (seconds since an epoch), or to convert seconds to and from human readable formats like Wed Dec

31 19:00:00 EST 1969.

Syntax: clock *subcommand args*

subcommand The clock command supports several subcommands including:

- seconds Returns current time and date as a system dependant integer.
- format converts a system dependant integer time to a human readable format. There are many formatting commands to fine tune the output.
- scan converts a human readable time/date string to a system dependant integer value.

We can decide that we'll save and view the history data on the same platform, and use the clock seconds command to add a timestamp to the output:

```
puts $outfile "[clock seconds] $symbol $dataList"
```

Which will generate lines like this:

```
 949199093 SUNW {75 1/16} {-1 7/8} -2.4 1/28 {77 1/8} {79 1/2} {73 27/32} 19,016
```

After this robot has been running for a few months, we'll have some historical data, and can think about looking at trends. Since this is an article about Tcl, not how to analyze the stock market (empirical evidence indicates you should not take stock advice from me), the analysis will consist of reading the history file and graphing selected stocks.

We can architect this program using one of two patterns.

Single Loop

```
instantiate graph
open data file
while data in file {
  read data
  plot data
  }
```

Split Loops

```
open data file
while data in file {
  read data into internal structure
  }
instantiate graph
foreach dataset {
  plot data
  }
```

The single loop pattern looks seductively simple, but it's a trap.

Programs (and especially GUI based programs) should be architected with modules that require a single type of I/O. The module that reads data from a file should only

interact with the file, the module that displays data should only interact with display, and the modules that analyze data should only analyze.

Using a single loop pattern makes a small program, but the monolithic structure makes it difficult to extract a piece of functionality for another program (like reading the data from the file), and leads to spagetti like code that is difficult to maintain (you would have to work around the graphing code when you want to the change the data format).

Thus, this graphing program will read the required data from the history file with one procedure, and display that data in another.

A good design for this program would probably be one function to read the data, one function to format it, and one to plot it, but to make life simpler (and this example smaller), the `readData` procedure will select data for a single company, and will reduce the data to the two fields we'll be graphing (the time and current price). The procedure will return the time and price data as two lists, which can be passed to the BLT graph widget.

Since the Tcl `channel` construct for I/O is so generalized, I prefer to pass a channel handle to a procedure that will be reading data, rather than have the procedure open the channel. This puts a bit more of the project specific information (what kind of channel I'm reading from) outside the procedure, and keeps the procedure a bit more generalized.

Finally, the stock prices are reported as fractions, not decimal, but the graph widget only groks decimal numbers. A simple regular expression will split the price into a whole number, numerator and denominator, and the `expr` command can convert that into a decimal value.

The procedure to read in the data looks like this:

```
proc readData {infl symbol} {

  while {[set len [gets $infl line]] >= 0} {

    set id [lindex $line 1]
    if {![string match $id $symbol]} {continue}

    set price [lindex $line 2]
    set time [lindex $line 0]

    # Convert "X Y/Z" or "Y/Z" prices to decimal format

    if {[regexp {([0-9]* +)*([0-9]*)/([0-9]*)} $price m whole num denom]} {
        set price [expr $whole + ($num / $denom.0)]
    }

    lappend priceList $price
    lappend timeList $time
  }
  return [list $priceList $timeList ]
}
```

This procedure can be invoked with something like:

```
set inputFileHandle [open $fileName "r"]
set data [readData $inputFileHandle $symbol]


set price [lindex $data 0]
set time [lindex $data 1]
```

Now that the data has been read and formatted, we can graph it. Which brings us back to the BLT extension.

Before we can use the BLT extension, we need to either get a pre-compiled version of the extension (from the Tcl-Blast CDROM, for instance), or download and build it.

You can download the latest BLT source from `www.tcltk.com/blt/`.

To build BLT, unpack the archive, cd to the directory it created (probably `blt2.4n`), and type:

```
./configure;make;
```

If you have root permission you can type `make install` to complete the installation. Otherwise, you can use the extension where it's built by setting the `auto_path` variable in your script to point to the BLT library directory with a line like this:

```
lappend auto_path $env(HOME)/blt2.4n/library
```

There are two ways to use extensions with Tcl/Tk.

The old technique (which works on all platforms) is to create a new tclsh or wish executable linked with the new extension code. When you invoke the new tclsh or wish executable (probably named bltwish for the BLT extension), it will have all the normal Tcl/Tk commands, as well as the new extension commands.

If your platform supports dynamicly linked shared libraries (BSD/OS after Version 4.0, Solaris, FreeBSD, MS-Windows or Linux for example), the configure script will generate a makefile which will create a .so (or .dll on a MS-Windows platform) file in the `src/shared` directory.

Extensions built from dynamicly linked shared libraries can be loaded into a running wish interpreter with the `load` command, or loaded as needed with the `package require` command.

The `load` command is the easiest to use in a development mode. The syntax is:

Syntax: load *libFile.so*

If you are going to make an extension part of your regular coding, you'll want to install the extension in a normal place (like `/usr/local/lib`), and let Tcl find it with the `package require` command.

Syntax: package require *packageName ?revision?*

packageName The name that the package is identified by. This will probably be the letters between `lib` and `.so` for a shared library, but might be any identification string.

revision A number that defines the acceptable revisions. A revision number may be a single integer, or a pair of integers. A single integer is interpreted as a major revision number, and a pair of numbers is interpreted as a major and minor release specification.

none	Use the newest revision available.
integer	Use the newest revision in this major revision number.
integer.integer	Use only the revision number specified by the major.minor revision number pair.

Once we've loaded the BLT package, creating and populating a graph is easy. The command to create a new graph widget is `graph`.

Syntax: graph *name ?option value?*

name A name for this graph widget. Using the standard Tcl window naming conventions.

?option value? Option and Value pairs to fine tune the appearance of the graph. The available options include:

-background	The color for the graph background.
-height	The height of the graph widget
-title	A title for this graph.
-width	The width of the graph widget

The BLT package loads the new commands into the `blt` namespace, so we create a graph with a command like:

```
::blt::graph .g -title "Stock Prices for $symbol" -width 500
```

This creates an empty graph widget. As with other Tk widgets, when a graph widget is created, a command with the same name is created. A Tcl script can control the widget using that command. You can think of Tk widgets as objects with transparent internal data and a single method.

The next step is to display our data. The BLT graph widget deals with data as graph elements. An element is a set of X and Y data and the options that describe how the data should be displayed.

The BLT graph widget command has several subcommands including the `element` command, that lets us define and modify a graph element.

Syntax: widgetName *element create ?option value?*

`widgetName` The name of a previously created `graph` widget.

element create Create a new graph element.

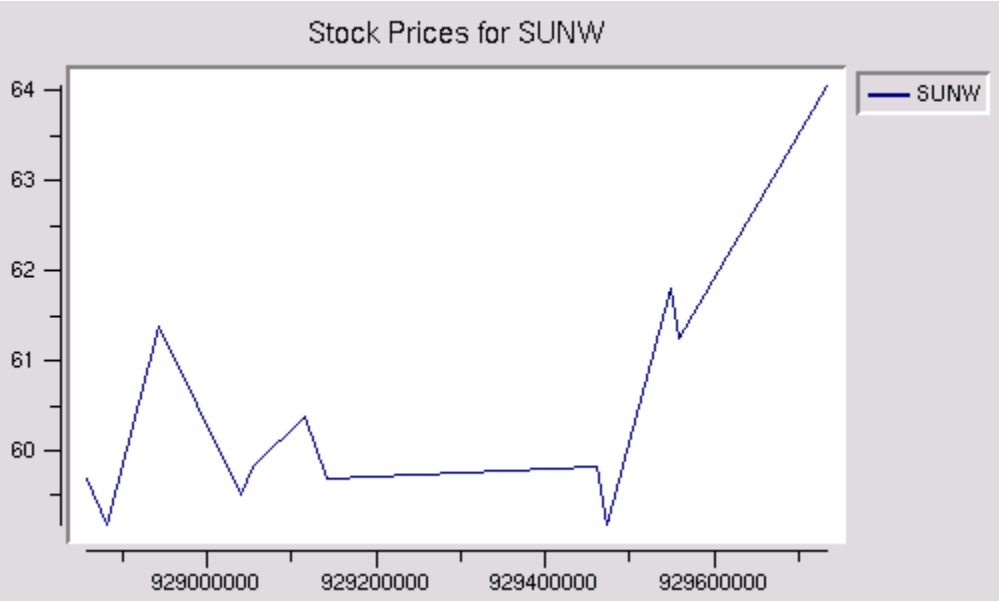
option value Option value pairs that define this element. Options include:

- -color The color of the line
- -symbol The symbol to display at data points. Values include square, circle, diamond, plus, cross, triangle, none and user specified bitmaps.
- -xdata A list of numeric data to display on the X axis.
- -ydata A list of numeric data to display on the Y axis.

Those two commands will create a graph and display a set of data. The complete program looks like this:

```
#!/usr/local/bin/wish8.2

package require BLT

proc readData {infl symbol} {

  while {[set len [gets $infl line]] >= 0} {

    set id [lindex $line 1]
    if {![string match $id $symbol]} {continue}

    set price [lindex $line 2]
    set time [lindex $line 0]

    if {[regexp {([0-9]* +)*([0-9]*)/([0-9]*)} $price m whole num denom]} {
        set price [expr $whole + ($num / $denom.0)]
    }

    lappend priceList $price
    lappend timeList $time
  }
  return [list $priceList $timeList ]
}

set inputFileName [lindex $argv 0]
set symbol [lindex $argv 1]

set infl [open $inputFileName r]
set data [readData $infl $symbol]

set price [lindex $data 0]
set time [lindex $data 1]

::blt::graph .g -title "Stock Prices for $symbol" -width 500
grid .g

.g element create $symbol -xdata $time -ydata $price  \
        -symbol none
```

which will generate this image:



when invoked as:

```
graph.tcl stockdata.dat SUNW
```

This is better than nothing, but the dates are in seconds since the epoch, which I don't translate to year and day in my head, and I'd like to see more than one stock at a time. The next article will discuss ways to make this graph prettier and more useful.