the tclsh spot

by Clif Flynt
<*clif@cflynt.com*>

Clif Flynt has been a professional programmer for almost twenty years, and a Tcl advocate for the past four. He consults on Tcl/Tk and Internet applications.

The previous Tclsh Spot articles discussed building a stock quotegathering robot, saving the data and using the BLT graph widget to display a stock's history. This article will describe more details about the BLT graph widget and discuss using Tcl's associative arrays.

The first step in any sort of data analysis is getting the data. In this case, that means reading the stock data from the file created by the stock quote—retrieval robot.

The stock-quote robot produces a datafile with lines that resemble this:

```
955127760 SUNW {95 3/8} {2 11/16} 2.9 13:00 {93 7/8} 96 {93 1/32} 6,938
955127780 INTC {134 13/16} 5 3.9 13:00 {131 3/8} {136 17/32} {131 5/16} 15,590
```

These are deliberately formatted (by the robot) to be Tcl lists, to make reading them a bit simpler. But the data isn't quite as ready-to-use as it might be.

The problems to address are:

- The prices are given as fractions instead of decimal numbers.
- The trade volumes are written with commas.
- The data for many companies is mixed into one file.

The first two problems are data-representation issues. We can deal with them with a single data-conversion procedure to convert fractions to decimals and strip out commas.

For a task this simple — only two processing options, and a simple test to figure out how to process the data — I prefer to put both the selection and the conversion intelligence in the subroutine, instead of writing two conversion procedures and putting the selection intelligence in the loop. My reasons are:

- Loops tend to get long, and anything that simplifies them is good.
- If the selection logic becomes complex as I better comprehend the problem, I can rewrite the logic in a procedure more easily than I can rewrite the spaghetti code inside a loop.
- Conversion code is likely to be useful for another project, and it's easier to grab a procedure than extract code from a loop.

This procedure converts a fraction like 2 1/2 to a decimal, using the regexp command to convert the number to an arithmetic expression (2 + 1/2.0) and then evaluating the arithmetic expression to get the floating-point number 2.5.

Converting 1/2 to 1/2.0 looks strange, but there is a reason for it. By default, the Tcl expr command leaves numbers in their native state when it does math operations. Thus, it won't convert from integer to float before doing a division and will return an integer result (1/2 == 0).

However, if one operand in an expression is a float, other operands are promoted before the operation is performed. Thus, the operation 1 / 2 returns the integer 0, while 1.0 / 2 returns the floating-point value 0.5.

We used the `regsub` command in the robot as part of the page-parsing logic. We can also use the `regsub` command to strip unwanted commas from the volume data.

Here's a procedure that will convert a number in one of the two unwanted forms (fraction, or including a comma) into a decimal:

```
proc normalize {val} {
  # Only do fraction to decimal conversion if a fraction exists.

  if {[regexp {([0-9]* +)*([0-9]*)/([0-9]*)} $val m whole num denom]} {
    set val [expr $whole + ($num / $denom.0)]
  }
  # Delete any commas that might be in the value
  regsub -all "," $val " " val

  return $val
}
```

That leaves us with the third problem, separating the data for multiple companies into the appropriate lists.

The Tcl interpreter supports three data structures:

- simple variables like numbers or strings
- lists
- associative arrays

The associative array was first introduced to Tcl in the TclX extension and was quickly merged into the Tcl core. An associative array is an array structure in which the indices are strings instead of numbers. This concept is perfectly obvious if you're familiar with Awk or Perl, or perfectly bizarre if you've always worked with languages like FORTRAN or C.

Associative arrays look and act just like normal Tcl variables, except that the array name is followed by a pair of parentheses that enclose the index value. For example, `os(unix)` represents the index `unix` in the associative array `os`.

The associative array is the most powerful data structure in Tcl. Using some do-it-yourself naming conventions, you can emulate most of the complex data structures that other languages support.

For instance, using an associative array lets us write code like:

```
set fruitPrice(apple) 0.5
set fruitPrice(banana) 0.25
```

instead of the C equivalent:

```
struct fruit {
  char *name;
  float cost;
} fruitPrice[2];

  fruitPrice[0].name = "apple";
  fruitPrice[0].cost = 0.5;
```

```
  fruitPrice[1].name = "banana";
  fruitPrice[1].cost = 0.25;
```

If we want to track more data about our fruits, we can define a naming convention to use. We might decide that the indices will be the fruit name followed by a data description. For example:

```
set fruitInfo(apple.price) 0.5
set fruitInfo(apple.inventory) 500
set fruitInfo(apple.color) red
set fruitname "banana"
set fruitInfo($fruitname.price) 0.25
set fruitInfo($fruitname.inventory) 1000
set fruitInfo($fruitname.color) yellow
```

When it comes to graphing the stock data, we want a list of timestamps and a list of prices for a given company. So, to make life (or at least this example) simple, we declare that these lists will be saved in an associative array with an index naming convention of `StockSymbol.Description.` For example,`Data(SUNW.price)` will have a list of selling prices, and `Data(SUNW.date)` will have a list of timestamps for Sun's stock.

The data in the file is a set of lines, and each line is a list consisting of TimeStamp, Stock Symbol, Selling Price, Absolute Change, Percent Change, Time of Quotes, Opening Price, High Price, Low Price, and Volume.

The second item in the list is always the stock symbol. We can extract that value from the list with the lindex command. (The first item in a list is at position 0.) Once we know the symbol name, we can parse the rest of the data in the line using a foreach loop.

One of the features of foreach is that it can iterate through multiple lists simultaneously, retrieving the first item from each list, then the second item, and so on. The syntax for this is:

```
foreach variable1 list1 variable2 list2 ... variableN listN {...}
```

This code will read the lines from the file and extract the stock symbol from each line. It then iterates through the list of descriptive names and a list of values, and appends the current value to the appropriate list within the associative array:

```
proc readData {infl} {
  global Data

  while {[set len [gets $infl line]] >= 0} {
    set id [lindex $line 1]
    foreach n {date id price change pct dt o high low vol} v $line {
      lappend Data($id.$n) [normalize $v]
    }
  }
}
```

After this procedure has run, the associative array Data will have lists of all the data we've collected, indexed by stock symbol and type of data.

Now, we can get back to making graphs.

Just on general principles, I don't want to create our graph in mainline code and pack it on the main window. Putting this code into a procedure, with the parent frame as an argument, makes it easier to merge this graph into a larger application when we have multiple frames to deal with.

This procedure will create an empty graph with just a label and will return the name of the new graph widget:

```
proc makeGraph {parent name} {
  global Data
  :blt::graph $parent.g_$name -title "Stock Data for $name" -width 600 -
height 400

  return $parent.g_$name
}
```

The graph widget displays lines as graph elements. A graph element is an object that contains a list of X, Y values and some options to define how the line should be drawn.

Syntax: *widgetName* `element create`

?option value?

We can draw a line showing price versus time by adding this code to the makeGraph procedure after the graph command:

```
set name "SUNW"
$parent.g_$name element create "$name Price" -xdata $Data($name.date) \
  -ydata $Data($name.price) -symbol none
```

This procedure will create a graph like this, with a label and legend, and with the X axis tics listed as seconds since the epoch.

This is better than nothing, but not much. However, the BLT package has lots of facilities for customizing graphs. For instance, the first thing I want to change on this graph is the tic labels. The BLT widget command axis can be used to configure a graph's X and Y axes.

Syntax: *widgetName* `axis configure` *axisName -option1 value1 ...*

widgetName The name of the graph object that contains this axis
`axis configure` Identifies this command as configuring the axis
axisName Identifies which axis is being configured. The default axes are:
- X The bottom X axis
- X2 The top X axis
- Y The left-hand Y axis
- Y2 The right-hand Y axis

-option value An option name and the new value to associate with that option.

Options include:

- -fmt The name of a function to use to format the tic labels
- -max The maximum value to display
- -min The minimum value to display
- -hide 1 to hide an axis, 0 to display. By default, the X and Y axes are displayed, and the X2 and Y2 are hidden.

So, to show the X axis tics as Month/Day, we can use the `Tcl clock` command to convert the seconds since the epoch into a MM/DD format with a procedure like this:

```
proc fmt {graph sec} {
  return [clock format $sec -format {%m/%d}]
}
```

and tell the graph to use this procedure to format the tics by adding these lines to the `makeGraph` procedure:

```
  # Format the x-axis tick labels as Month/Day
  $parent.g_$name axis configure x -command fmt
```

This is better, but we've still collected a lot of data we aren't viewing. For example, it might be good to know the trade volume.

We could make another line on the graph to show the volumes, but lines imply that the there is continuity between values, and there is no connection between yesterday's and today's trade volume.

A bar chart is a more appropriate way to display this info. We can generate barcharts on our BLT graphs. In fact, there is a whole other widget (`barchart`) designed for barcharts, but for this application it makes more sense to put the bars on the price graph.

The command to build a bar is:

Syntax: *widgetName* `bar create` *label ?-option value?*

`bar create`	Create a new set of bars on the graph
label	A label that describes this barchart (this string will be displayed in the legend)
-option value	Option and value pairs to describe this barchart. Options include:
-xdata	A list of values for the X axis
-ydata	A list of values for the Y axis
-mapy	The axis to map this data against (defaults to the lefthand Y axis)
-mapx	The axis to map this data against (defaults to the bottom X axis)
-fg	The foreground color for the bars
-bg	The background color for the bars
-barwidth	How wide to draw the bars (defaults to a single pixel)

One of the tricks in putting two sets of data on the same graph is how to scale the data. Since a high price for a stock is around 100, while a low volume is around 10,000 shares, we really can't plot both of these against the same Y axis.

This is where the Y2 axis and the -mapy option come in. We can map the prices on a 0—100 scale on the left axis, and volume on a 0—100,000 scale on the right axis.

In fact, we don't have to declare the sizes of the axes. The graph widget will automatically scale the axes from the minimum to maximum value in the data set.

So, to generate a volume barchart on the graph with our price graph, we add this line to the `makeGraph` command:

```
$parent.g_$name bar create "$name Volume" -xdata $Data($name.date) \
  -ydata $Data($name.vol) -mapy y2 -fg green -bg green \
  -barwidth 2000
```

Since the dates on the X axis are the same for the price and volume graphs, we don't need to use separate axes for that data. The `-barwidth 2000` makes the bars a little wider than a normal single-pixel line (2000 seconds wide, if you are counting units).

What this doesn't do is display the values. By default, the X2 and Y2 axes are hidden. But adding this line will display the Y2 axis:

```
$parent.g_$name axis configure y2 -hide 0
```

That leaves the high and low data. It would be nice to see the range of values in a day, and whether our stock closed at the top of the range or bottom.

Another feature that BLT supports is markers. Markers are things that you can put at location on a graph. They can be text messages (like "This is when the SEC cancelled trading"), or bitmaps (like a happyface when the stock splits), or polygons, or lines.

In this example, we'll use line markers to show the high and low prices for a stock, similar to the error lines in the graphs from your old physics lab.

The syntax for creating markers is:

Syntax: *widgetName* `marker create` *type ?-option value?*

widgetName	The name of the graph widget
`marker`	Create a marker.
type	The type of marker being created (valid types include text, line, bitmap, image, polygon, and window)
option value	An option/value pair. The options available vary depending on what type of marker is being created. Options for line markers include:

-outline	The color for the line
-coords	A list of coordinates to define the line

To create these markers, we need to tell the graph widget to draw a vertical line from the low price to the high price at each vertex. We can do this with the`create marker` command, and another loop with multiple lists and variables.

But while we're doing that, we might as well track the maximum and minimum prices in the high and low dataset. Since the graph was scaled to the maximum and minimum closing prices, there may be highs and lows outside that range.

Like most of Tcl, the BLT graph widget is introspective. You can query it to find out such things as what current configuration values are using the same configure and cget subcommands that are supported by native Tk widgets.

Our code can query the Y axis to find out what the max and min values in the price data are before we start looking at the high and low data.

The problem is that the graph widget hasn't looked at the data yet. When we invoked the graph and create element commands, the interpreter didn't actually create a graph. It placed events on the event queue to create the widgets as soon as the interpreter isn't busy doing something else (like evaluating our procedure). The event queue won't be checked until after our process finishes the `makeGraph`procedure.

The solution to this problem is to use the `update` command. The `update`command will cause the event queue to be processed before returning. The`update` command comes in two flavors, `update`, which will process all events, and `update idle`, which will only process events that are in the idle loop. Updating graphics objects is an idle-loop task, so that's the flavor we should use for this application.

This code will force the idle loop to be processed, find the starting maximum and minimum prices, draw high/low lines at each price, and then reconfigure the axis to show the new range:

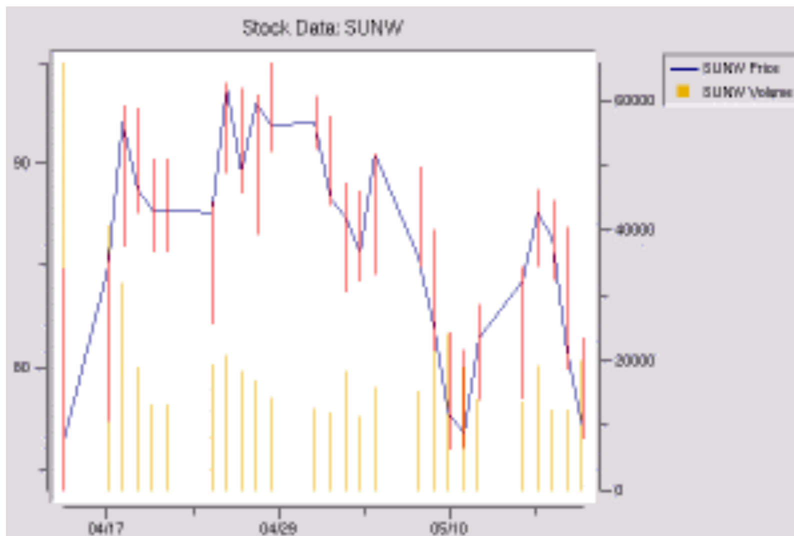
```
# Create vertical high/low lines at the vertices, and find max & min.
foreach d $Data($name.date) h $Data($name.high) l $Data($name.low) {

  $parent.g_$name marker create line -coords [list $d $h $d $l] \
    -outline blue

  if {$l < $min} {set min $l}
if {$h > $max} {set max $h}
}

# Now expand the Y axis to the real min/max range.
$parent.g_$name axis configure y -max $max
$parent.g_$name axis configure y -min $min
```

The code we've discussed in this article will generate a display that looks like this from a command like `wish stockShow.tcl SUNW`:



This code, and code from other Tclsh Spot articles, is available on my new Web site <http://www.noucorp.com>.

This is plenty of info, but running a new command-line task for each stock is too much finger work. The next article will discuss ways to display multiple stocks in this application.

```
#!/usr/local/bin/wish

package require BLT

set Graph(dataName) stock.data
set name $argv

################################################################
# proc normalize {val} - -
#  Convert fractions to decimals and remove any commas
# Arguments
#  val    A numeric value
#
# Results
```

```
#  Returns a legal floating point or integer value.
#
proc normalize {val} {

  # Only do fraction to decimal conversion if a fraction exists.
  if {[regexp {([0-9]* +)*([0-9]*)/([0-9]*)} $val m whole num denom]} {
    set val [expr $whole + ($num / $denom.0)]
 }

  # Delete any commas that might be in the value
  regsub -all "," $val " " val

  return $val
}

################################################################

# proc readData {infl} - -
#  Reads data from a stock data file
# Arguments
#  infl A channel to the data file
#
# Results
#  Creates lists of values in the Data global associative array,
#  sorted by StockSymbol.DataType
#
proc readData {infl} {
  global Data

  while {[set len [gets $infl line]] >= 0} {
    set id [lindex $line 1]
    foreach n {date id price change pct dt o high low vol} v $line {
      lappend Data($id.$n) [normalize $v]
    }
  }
}
################################################################
# proc fmt {graph sec} - -
#  Formats a time-since-epoch time into MM/DD
# Arguments
#  graph The name of the graph which includes this tic mark.
#  sec Seconds since the Epoch.
# Results
#  Returns the appropriate MM/DD value.
#
proc fmt {graph sec} {
  return [clock format $sec -format {%m/%d}]
}
################################################################
# proc makeGraph {parent name} - -
#  Makes a stock graph.
# Arguments
#  parent  A parent frame to hold this graph
#  name;    The stock symbol to use as a key to access the data
#          to display in this graph.
# Results
#  Creates a new graph widget, and returns the name of that
```

```
#  widget.
#
proc makeGraph {parent name} {
  global Data

  if {![winfo exists $parent.g_$name]} {
      # Create the graph
      ::blt::graph $parent.g_$name -title "Stock Data: $name"
        -width 600 -height 400

      # Format the x-axis tick labels as Month/Day
      $parent.g_$name axis configure x -command fmt

      # Create a line showing the stock price when
      # the robot ran.
      $parent.g_$name element create "$name Price" -xdata
        $Data($name.date) \
        -ydata $Data($name.price) -symbol none

      # Generate a bar chart for volume, and display the second
      #   Y axis that the bar chart references
      $parent.g_$name bar create "$name Volume" -xdata
        $Data($name.date) \
        -ydata $Data($name.vol) -mapy y2 -fg green -bg green \
        -barwidth 2000

      $parent.g_$name axis configure y2 -hide 0

      # Do an update to force the graph to run through the data
      # and calculate the min and max values.

      update idle;
      set max [$parent.g_$name axis cget y -max]
      set min [$parent.g_$name axis cget y -min]

      # Create vertical high/low lines at the vertices,
   # and find max & min.
      foreach d $Data($name.date) h $Data($name.high)
        l $Data($name.low) {

    $parent.g_$name marker create line -coords
      [list $d $h $d $l] \
          -outline blue

        if {$l < $min} {set min $l}
        if {$h > $max} {set max $h}
      }

      # Now expand the Y axis to the real min/max range.

      $parent.g_$name axis configure y -max $max
      $parent.g_$name axis configure y -min $min
    }
  return $parent.g_$name
}
```

```
set infl [open $Graph(dataName) r]
readData $infl
close $infl

set w [frame .graphs]

pack $w -side bottom

pack [makeGraph $w $name]
```