The following paper was originally published in the
Proceedings of the Sixth Annual Tcl/Tk Workshop
San Diego, California, September 14–18, 1998

# Data Objects

George A. Howlett
*Bell Labs Innovations for Lucent Technologies*

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

# Data Objects

George A. Howlett
*Bell Labs Innovations for Lucent Technologies*
*gah@bell-labs.com*

## Abstract

*Scripting languages are great for gluing together components, but they suffer as the complexity or size of data scales upward. Data objects solve this problem by marrying both high-level and low-level programming styles. Data objects are self-contained representations of data. They define both the structure of the data and the methods to access it. Data may be accessed through both Tcl and a C interface. This paper will describe two such data objects, a vector and table object.*

## Introduction

Scripting languages such as Tcl[1], have been described as fundamentally changing the way people write programs, representing a very different style of programming[2]. They let developers rapidly form new applications from components and pieces of existing applications. What distinguishes scripting languages from conventional structured programming languages (e.g. C, C++, FORTRAN) is that they are high-level, interpreted, and weakly typed.

Despite their advantages, scripting languages do not replace structured programming languages. They, in fact, offer a very different set of trade-offs.

### *Where Scripting Languages Succeed*

Scripting languages make it easy for programmers to control how components are used. Because programming is done at a high-level, components have simple interfaces.

Components are identified by strings, not pointer addresses. Even heterogeneous components can be referenced and grouped by a simple list of names. Components also have well-defined operations that can be used to access or modify their internal data without requiring an understanding of how the data is structured.

Scripting languages are usually interpreted, so the flow of program execution is simple to change and test. It's easy to wire applications together quickly or try out new algorithms.

A good example of high-level components is the Tk widgets. Widgets are referenced by string identifiers. They have well-defined **`configure`** and **`cget`** operations that change widget attributes. (You can change a widget's font without knowing anything about the structure of a Tk font). It's easy to rearrange widgets or add and test new behaviors using the **`bind`** command.

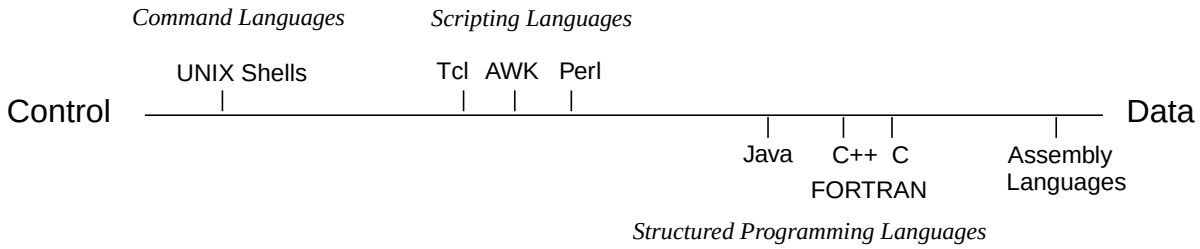
### *Where Scripting Languages Fail*

Scripting languages are inversely weak where structured programming languages are strong.

As an application's data scales from tens to thousands of elements, scripting languages perform worse than compiled low-level languages. Both in terms of performance and memory utilization, scripting languages carry greater overhead. For example, a simple array of 10,000 elements in Tcl will use as much memory to store the indices as the values. Arrays in C have no such overhead.

Flexibility has a cost. Weak typing requires that data many times be converted back-and-forth from strings to native machine types (e.g. ints, doubles). For example, the time to plot a graph of 10,000 data points is dominated by the string-to-decimal conversions, not the performance of the X server.

**Figure 1.** Programming Languages: Control Versus Data




Scripting languages lack facilities for structuring and manipulating complex data. They are, by definition high-level. You can not easily define new data types, access pointers, or manage native types.

Side-effects of weak data structuring can be seen in Tk. Widgets often get used as data containers. For example, string data is sometimes put into a text widget just to use its more powerful search and replace operations.

Data also becomes tightly coupled to widgets. Let's say you are displaying file directory information in a listbox widget. You may also maintain several Tcl arrays for various information fields (owner, date, size, etc.) other than the file name. Each of these data containers must be synchronized. If entries are inserted or deleted or the listbox is itself destroyed, the other data containers must be likewise updated. Program control becomes increasingly complex and error prone as more parallel data structures are required.

### *What Makes a Language High Level?*

It is usually in terms of data where scripting languages fail. In applications data can scale much more quickly than code size, sometimes by several magnitudes. High level data abstractions preclude fine-grain access to data. Furthermore, the overhead of weak typing both increases memory consumption and slows data processing.

Consider programming languages in terms of how finely or coarsely they handle data. Figure 1. shows a continuum from high-level command languages to low-level structured programming languages. High level languages abstract data to remove details, making control very simple. Structured programming languages provide a very fine-grain control of data. Conversely, control is complicated by the details and complexity of the data representations.

## Data Objects

Tcl was originally designed to be extensible, allowing you to link your own C code to extend the language by adding new commands and variables. We can use this feature to represent data at both programming levels, as *data objects*.

Data objects are simply containers for data. They are objects because each instance represents both a set of data and high-level operations for querying and modifying that data†.

### *Programming at Both Levels*

At the Tcl programming level, a data object is referenced by its string identifier. Objects can be shared simply by passing their names. Object data is accessed or changed through a collection of well-known operations. Because data operations are written in C code, operations for data objects run as fast as compiled code.

Data objects also provide a low-level programming interface. An object's data can be accessed directly from its C API, without going through Tcl or converting the data from a string. For example, widgets may use this interface to access the data in its native format.

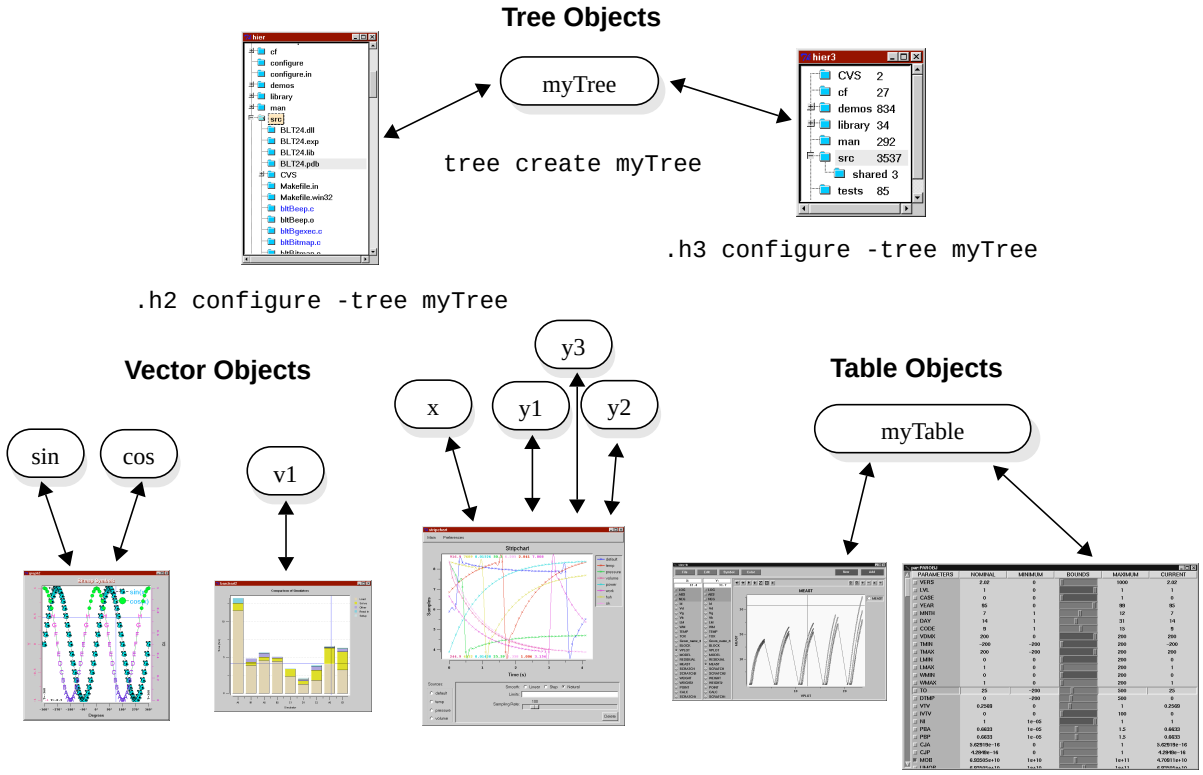
### *New Direction: Data Object Based System*

Imagine a system where data objects are ubiquitous. In this hypothetical system, data objects are the new currency for building applications. Standard data objects automatically work seamlessly with widgets. Extensions plug together. A database can automatically display its data in a spreadsheet widget because they both use the same table data object.

---

†. Data objects are object-based. This differs from object-oriented in that data objects have no inheritance or ability to define new operations.

**Figure 2.** Data Objects in BLT




The point is not that people want to write low-level code. On the contrary, if standard data objects became widely used, many applications could be written by plugging several large components together.

The BLT toolkit has slowly moved in this direction, representing complex data not as strings but by data objects. BLT currently has two data objects: a vector object representing an array of numbers, and a tree object representing a hierarchy. A table object will also soon be available. Several widgets in BLT work with data objects as shown in Figure 2. For example, the **`hierbox`** widget can use the tree object for its data. Note that more than one widget can use the same data object. Each widget can in turn offer a different view of the same data.

The following sections will describe both a concrete example of a data object (the vector object) and a real-life application built using data objects.

## Example: Vector Objects

Originally the only way to pass X-Y coordinates to the BLT plotting widgets was as Tcl lists (`-xdata` and `-ydata` options). Internally, the graph processes the string data, converting it into an array of doubles.

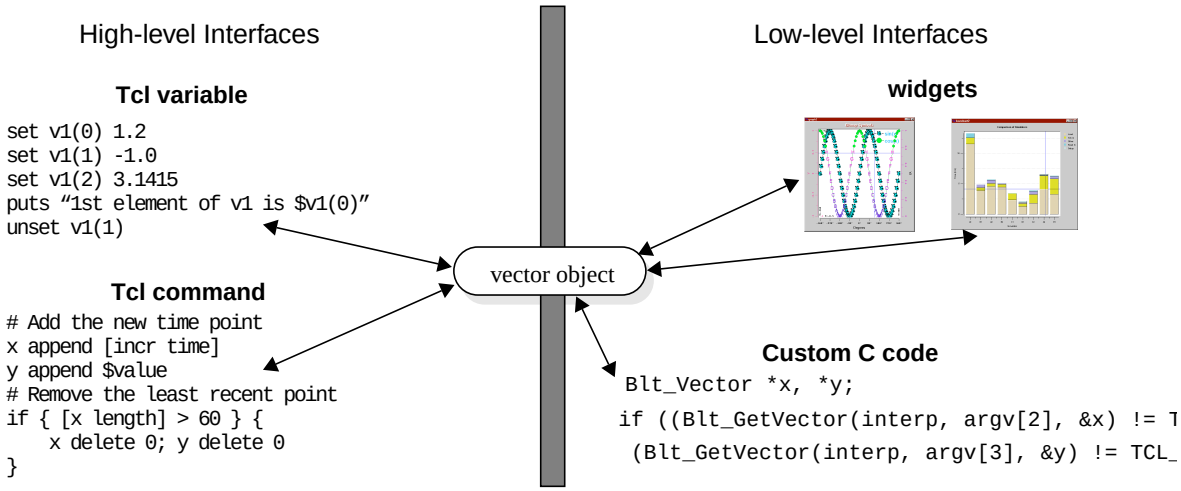
```
# Data values
set x { 0.2 0.4 0.6 0.8 1 1.2 }
set y1 { 11 21 28 34 38 39 }
set y2 { 26.2 50.5 72.9 93.3 112 128 }
barchart .b

# Add two new elements to the graph
.b element create e1 -xdata $x -ydata $y1
.b element create e2 -xdata $x -ydata $y2
```

This is inefficient in terms of memory. The same data points are stored in two different formats: Tcl lists and binary data. It's also inefficient in terms of performance. It's not usual to plot thousand of data points. New coordinates may also be added over time. Each time new data points are added to the list, the entire list must be converted from decimal strings to double precision values.

Tcl arrays can't be used because there is no implied ordering to associative arrays. Especially for plotting, you need to insure the second data point comes after the first, and so on. This isn't possible since arrays are really hash tables. Also, associative arrays consume memory because both the index and value are stored as strings for each data point.

**Figure 3.** Vector Object Interfaces




A worse problem is that data is tightly coupled to the widget. Data selection and analysis are important parts of plotting. Every data operation requires coordinate data to be translated from Tcl strings.

One alternative to is add selection and analysis functions to the widget itself (e.g. text widget). It seems the "essential" set of functions is always increasing. The danger is that as the widget grows in complexity and code size, it becomes monolithic. The focus of the widget shifts from displaying data to managing it.

*Vector Data Object*

Instead the **graph** and **barchart** widgets use vector data objects. A vector object simply represents an array of doubles, indexed by integers. The `-xdata` and `-ydata` options of the graph recognize vector names. The graph directly accesses the vector object's data in its native format.

```
vector create x y1 y2
x set { 0.2 0.4 0.6 0.8 1 1.2 }
y1 set { 11 21 28 34 38 39 }
y2 set { 26.2 50.5 72.9 93.3 112 128 }
barchart .b
.b element create "e1" -xdata x -ydata y1
.b element create "e2" -xdata x -ydata y2
```

Vector objects have a variety of interfaces. They are shown in Figure 3. From Tcl, vector object data is accessed through a variable or command. Vector objects also can be extended by user-defined C code.

Vector objects are created with the **`vector`** command.

```
vector create v1
```

A new vector `v1` is created. At the same time, both a Tcl command and Tcl array variable `v1` are also created.

Vectors can be used like Tcl arrays. Vectors are indexed by integers, starting from zero. When elements of the array are read, set, or unset, the corresponding elements in the vector are accessed.

```
set v1(0) 1.2
set v1(1) -1.0
set v1(2) 3.1415
puts "1st element of v1 is $v1(0)"
unset v1(1)
```

The advantage of mapping an array to the data object is that it makes vector data appear and act like ordinary data in Tcl.

The vector's Tcl command can be used to access or modify elements, invoking one of several operations. For example, the **`delete`** operation removes elements by their index.

```
# Delete the first element
v1 delete 0
```

The **`expr`** operation performs arithmetic on vector.

```
v1 expr { v1 * (v2 + 10) }
v3 expr { sin(v2)^2 + cos(v1)^2 }
```

It's important to note that data objects provide *high-level* operations. This is different than translating low-level C or C++ code into Tcl. For example, to find numbers that lie in a certain range, I could write a loop that builds a new list. In fact, the code isn't very different than if this was programmed in C.

**Listing 1.** Example of Vector Object C API

```
Blt_Vector *xVec, *yVec;
int i, length;
double *x, *y;

/* Get the two vectors to multiple.*/
if ((Blt_GetVector(interp, argv[2], &xVec) != TCL_OK) ||
   (Blt_GetVector(interp, argv[3], &yVec) != TCL_OK)) {
    return TCL_ERROR;
}
length = Blt_VecLength(xVec);
/* Check that the vectors are the same length */
if (length != Blt_VecLength(yVec)) {
    Tcl_AppendResult(interp, "vectors ", argv[2], " and ", argv[3],
   " have different lengths",(char *)NULL);
    return TCL_ERROR;
}
/* Allocate result array and compute the product */
array = (double *)malloc(legnth * sizeof(double));
x = Blt_VecData(xVec), y = Blt_VecData(yVec)
for (i = 0; i < length; i++) {
    array[i] = x[i] * y[i];
}

/* Update the vector so it knows that its data has changed.
Old data will* automatically be freed. */
if (Blt_ResetVector(yVec, array, length, length, TCL_DYNAMIC) != TCL_OK) {
    return TCL_ERROR;
}
return TCL_OK;
```

```
set len [v1 length]
for { set i 0 } { $i < $len } { incr i } {
    if {($v1($i) < $lo) && ($v1($i) > $hi)} {
        lappend values $v1($i)
    }
}
```

The vector object instead provides a simple, high-level **search** operation.

```
set values [v1 search -value $low $high]
```

Vectors can be converted to and from strings (lists). The **set** operation sets the values of the vector from a list. The **range** operation returns a list of vector elements between two indices.

```
# Set the vector from a string.
set list { 1.1 0.0 2.3 4.4 -1.0 }
v1 set $list
# Set the string from the vector.
set list [v1 range 0 end]
```

## *C API*

No matter how many built-in operations a data object may have, it's likely that some functionality will always be missing. Data may need to be read in from a specific file format. Special calculations may be required. Therefore objects themselves need to be extensible.

A vector object's data can also be accessed from C using its library API. In the same way that Tcl is extensible, new code can be written to manipulate vectors in ways not available from its Tcl interface For example, the **spline** command in BLT uses vectors to create interpolating splines. Listing 1. shows an example of the C API to multiple to vectors together.

## *Vector Notifications*

Vectors provide a hook or callback to notify clients (such as the graph or barchart) when the vector data changes. Notifications usually as occur as idle tasks, but this can be user-controlled.

Notification occurs automatically, no matter how the vector was changed: via the vector's Tcl command, array variable, or C API. You can therefore separate the data processing portion of your application from the GUI.

For example, a graph that displays only the last 60 time points can be built using a barchart and a pair of vector data objects to hold the X-Y coordinates. As new data arrives, the new time point is appended to the `x` and `y` vectors. If there are more than sixty time points, the oldest is removed.

```
# Add the new time point
x append [incr time]
y append $valuea
# Remove the least recent point
if { [x length] > 60 } {
    x delete 0; y delete 0
}
```

There is no code to synchronize the barchart. It's not needed. The chart is redrawn automatically. The barchart sees the new values because it shares the vector's data instead of making its own internal copy. While several graphs can use the same vector, there will be only one copy of the data.

### *Vector Performance*

Since vectors are very simple data types, they can be compared with Tcl lists.

Numeric values were read into a variable **`data`** and arithmetic was performed on those values. The vector-based example *vector.tcl* is listed below.

```
vector create a b c d x
a set $data
b set $data
c set $data
d set $data
x expr { a * b + c * d }
x expr { a + b * c + d }
```

For lists, both byte-compiled (8.0) and pure-interpreted (7.6) versions of Tcl were tested. Vector operations are written in C code, so they are unaffected by the byte-compiler. The following example *list1.tcl* replicates the vector example, this time using lists. The procedures **`add`** and **`mult`** perform arithmetic on lists.

```
proc mult { list1 list2 } {
    set len1 [llength $list1]
    set len2 [llength $list2]
    if { $len1 != $len2 } {
         error "lists are different lengths"
    }
    foreach e1 $list1 e2 $list2 {
        lappend result [expr $e1 * $e2]
    }
    return $result
}

proc add { list1 list2 } {
    set len1 [llength $list1]
    set len2 [llength $list2]
    if { $len1 != $len2 } {
        error "lists are different lengths"
    }
    foreach e1 $list1 e2 $list2 {
        lappend result [expr $e1 + $e2]
    }
    return $result
}
set a $data
set b $data
set c $data
set d $data
set x [add [mult $a $b] [mult $c $d]]
set x [add [add $a [mult $b $c]] $d]
```

Another version, *list2.tcl*, is included below. The only difference is that the **`add`** and **`mult`** routines are replaced by a generic procedure **`calc`**. The operator is passed as an argument.

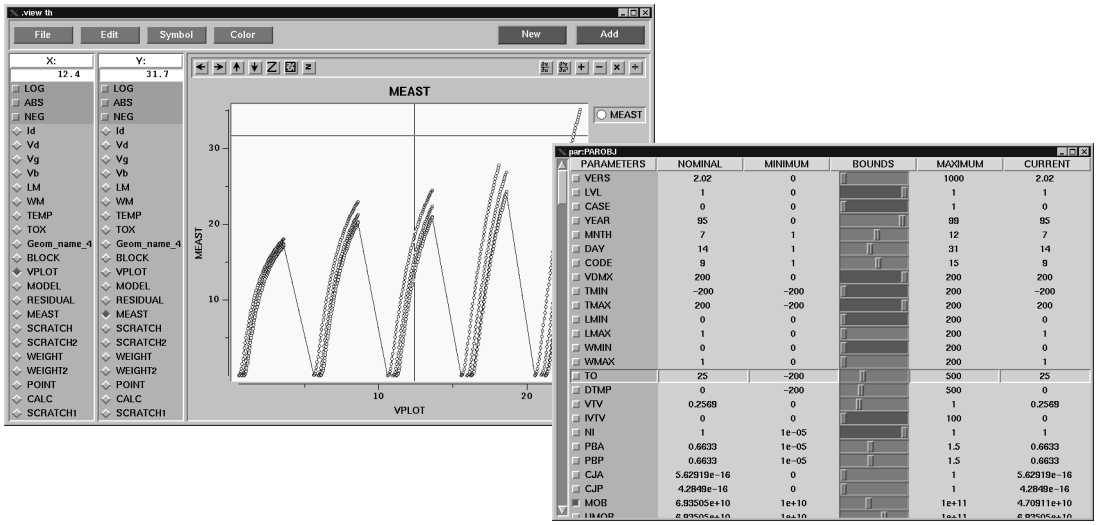
```
proc calc { list1 op list2 } {
    set len1 [llength $list1]
    set len2 [llength $list2]
    if { $len1 != $len2 } {
         error "lists are different lengths"
    }
    foreach e1 $list1 e2 $list2 {
        lappend result [expr $e1 $op $e2]
    }
    return $result
}
set a $data
set b $data
set c $data
set d $data
set x [calc [calc $a * $b] + [calc $c * $d]]
set x [calc [calc $a + [calc $b * $c]] + $d]
```

Each test was run using an increasing number of values. The time (in seconds) for each test is listed below.

| | Number of Values | | |
|---|---|---|---|
| | **1,000** | **10,000** | **100,000** |
| **7.6 *list1.tcl*** | 0.60 secs | 5.3 secs | 59.5 secs |
| **7.6 *list2.tcl*** | 0.63 | 5.6 | 62.3 |
| **8.0 *list1.tcl*** | 0.27 | 1.7 | 21.4 |
| **8.0 *list2.tcl*** | 0.90 | 8.0 | 85.3 |
| ***vector.tcl*** | 0.16 | 0.6 | 10.5 |

The improvement of (byte-compiled) Tcl 8.0 over Tcl 7.6 is striking. Not surprisingly though, vectors out perform both the byte-compiled and non-compiled versions

**Figure 4.** Snapshot of Camelot Parameter Extractor




of Tcl using lists‡. Vector operations written in C code will generally have less overhead than byte-compiled code. For *vector.tcl*, the read in and string-to-decimal conversions were the greatest portion of the time spent, not the arithmetic.

## Application: Parameter Extractor

A parameter extractor compares the behavior of a model of an IC circuit to measured results from real devices. A model is a set of equations (written in a structured programming language like C or FORTRAN) that given input voltages, returns an output current. Models also have parameters, such as temperature and device size, that affect its calculations.

The job of a parameter extractor is to evaluate the model of a circuit repetitively, bumping parameter values up and down in a way that minimizes the error between the model's outputs and measured data. The end product is a set of model parameters that best reflect the real devices.

This a computationally hard task. There can be thousands of measured data points. Models can have scores of parameters, allowing many degrees of freedom. Complex or over-parameterized models may exhibit multiple local minima. One method to stabilize the extraction process is to limit the number of the parameters that can be altered. It may also be desirable to constrain the parameters to lie in certain ranges (e.g. a resistance must be positive). How one selects initial parameter values, their constraints, and representative data regions matter greatly.

### *High-level Programming?*

Such compute-intensive tasks are usually the domain of low-level structured programming languages. The extraction step requires speed and numeric precision. But data selection and filtering is an equally important aspect of the application. The extractor must manage the various inputs (voltage, temperature, etc.) and the outputs (currents returned from a particular circuit). Additionally the model parameters must also be coordinated. Scripting languages are better at controlling how and where data is used. For parameter extraction, this is especially useful to experiment with different heuristics or wire in new models.

### *Camelot Parameter Extractor*

The Camelot parameter extractor developed at Bell Laboratories, lives in both programming worlds. Figure 4. shows a snapshot of the application. The currency of the application is a table data object.

The table data object represents a dynamically resizeable table of values. Rows and columns of the table can be selected, sorted, duplicated, etc. Each row and column has its own label and can be used just as a vector. For example, you can plot different columns of values.

‡. The results of *list2.tcl* demonstrate the penalties of dynamic code. Because the operator (* or +) was passed as an argument to the **`calc`** procedure, the Tcl 8.0 interpreter was unable to fully compile the procedure. As a result, it is slower than the Tcl 7.6 interpreter.

The code below creates a new table object `d0`. A new Tcl command `d0` is also created that can be used to access the table of data from Tcl. Table objects have several generic operations. The **`read`** operation reads tabular ASCII data from a file into the object. The **`column extend`** operation adds a new column to the table. The **`column label`** operation sets the label for columns.

```
table d0
d0 read sh.data
d0 column extend 1
d0 column label 2 VGS
d0 column label end IDSHAT
```

During the extraction process the model will write its outputs into the table. Models can be evaluated thousands of times during an extraction step. The outputs are later compared against the measured outputs to determine the fit. For example, a MOS model may store the modeled currents ($I_{DS}$) in the last column of the table. Models are quickly evaluated because the table object's data structure is immediately available through a C API.

Filtering and selecting of data are done from the table's high-level programming interface††. The **`select`** operation, chooses all rows that match a vector expression. In this case, we are looking for rows where column `VGS` is equal to `$value`.

```
d0 row select { VGS == $value }
d0 dup d1
```

The **`dup`** operation creates a new table object `d1` that contains only the selected rows of `d0`.

Parameters are stored in a second table object. It contains various parameter information; the initial and current parameter values, the minimum and maximum bounds, etc.

A new table object `p0` is created.

```
table p0
p0 read sh.pars
```

The parameters file `sh.pars` is then read into the table. The file contains the following information.

```
NAME        INCLUDE MIN     MAX     NOMINAL
BETA        0.0     25e-4    1e-4   50e-4
LAMBDA      0.0     0.0     0.0     0.1
VTH         0.0     0.8     0.4     1.2
DELTA       0.0     0.0     0.0     1.0
ATS         0.0     7.0     2.0     40.0
AST         0.0     15.0    2.0     50.0
RST         0.0     0.05    0.000001 1.0
NST         0.0     1.1     1.01    4.0
THS         0.0     0.0     0.0     0.5
THC         0.0     0.0     0.0     1.0
```

The first column is the parameter name. The second column is the inclusion status of the parameter. If the value is 0.0, the extractor will not adjust its value. The third and forth columns are the bounds of the parameter. The last column is the initial parameter value.

To run the extractor, you select data and the parameters that you wish to optimize. The **`extract`** command runs the extractor using the two table objects. The resulting new set of parameters will be appended as new column in `p0`.

```
d0 row select { IBL == 3.0 }
d0 dup d1
extract d1 p0
```

Since data objects are easily connected to widgets, the new parameters are automatically displayed in a table widget. The parameter fits are plotted in the graph widget.

The table objects act as a go-between to the high and low-level programming worlds. Like many applications, Camelot benefits from working at both levels.

## Conclusion

Data objects act as basic application building blocks. The high-level Tcl interface allows large numbers of objects to managed easily. Built-in operations let the user accomplish most data transformations from the Tcl level. The C level interface allows specialized code to be performed by revealing the object's internal data.

Two examples of data objects are vectors and tables. They both represent common data structures for many applications. The idea or utility of these objects is nothing new. Interactive statistical programming languages such as S[3] or MATLAB have demonstrated the power of vector and matrix programming for many years. What is new is how these objects can be applied to very high-level languages such as Tcl to solve a wide class of performance problems and to simplify interactions with data.

††. It is interesting to note that while custom C code tends to be application specific, the high level Tcl filtering and selection operations are generic.

Several questions remain open. What are the standard data objects? What like of low-level API is required? Is a better object system, such as [incr Tcl] required?

The hope is that if a set of objects becomes ubiquitous, it will form the basis for large plug-able components. These components would in turn become the building blocks for greater applications.

## References


1. J. K. Ousterhout, *Tcl: An Embeddable Command Language*, Proceedings of the 1990 Winter USENIX Conference, 1990, pp. 133-146.
2. J. K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century", *Computer*, March 1998, pp. 23-30.
3. R. A. Becker, J. M. Chambers, A. R. Wilks, *The New S Language*, Wadsworth & Brooks/Cole, Pacific, CA. 1988.