Table Objects (taboo)

Version 0.1

Alex Baker

https://github.com/ambaker1/taboo

September 26, 2023

Abstract

The "taboo" package implements an object-oriented tabular datatype in Tcl, using the object variable framework provided by the vutil package.

Tabular Data Structure

This package provides an object-oriented tabular datatype in Tcl, building upon the type system framework provided by the vutil package.

The string representation of this datatype is a dictionary, with keys representing the table header, and values representing the table columns. The values in the first column must be unique, and are called the table "keys". Correspondingly, the first header entry is called the "keyname". The remaining header entries are called the table "fields", and the remaining columns are the data stored in the table. The conceptual layout of the table is illustrated in the figure below.

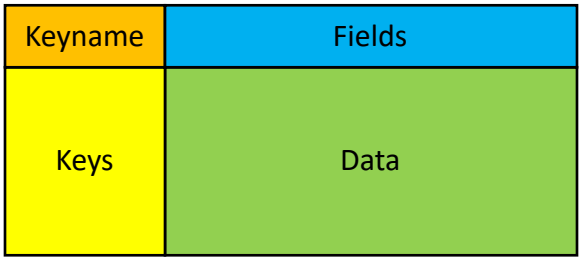

Figure 1: Conceptual Representation of Tabular Data Structure

There is no restriction on the type of data that can be stored in a table, as shown in the example below, which has keyname "key", keys "1 2 3" and fields "A B". Note that missing values are represented by blanks.

Example 1: String representation of tabular data type

Code:

```
key {1 2 3} A {5.6 {} 2.22} B {{hello world} 4.5 foo}
```

Creating Table Objects

This package provides the *table* class, using the type system provided by the "vutil" package. So, tables can also be created with the `::vutil::new` command.

```
table new $refName <$value>
::vutil::new table $refName <$value>
```

$refName	Reference name to tie object to.
$value	Value of table. Default blank.

Below is the example table used in the remainder of the documentation examples. Note that this format is also compatible with the string representation of Tcl arrays and dictionaries.

Example 2: Example table

Code:

```
table new tableObj {
    key {1 2 3 4 5}
    x {3.44 4.61 8.25 5.20 3.26}
    y {7.11 1.81 7.56 6.78 9.92}
    z {8.67 7.63 3.84 1.11 4.56}
}
```

Wiping, Clearing, and Cleaning a Table

The method *wipe* removes all data from a table object, so that its state is the same as a fresh table. The method *clear* only removes the data and keys stored in the table, keeping the fields and other metadata. The method *clean* only removes keys and fields that have no data.

```
$tableObj wipe
```

```
$tableObj clear
```

```
$tableObj clean
```

Standard Methods

Because the table objects are object variables, they have the same basic methods provided by the "vutil" package. For more info on these methods, see the documentation for the "vutil" package.

```
$tableObj --> $refName
$tableObj <- $object
$tableObj = $value
$tableObj ::= $body
$tableObj info <$field>
$tableObj print <-nonewline> <$channelID>
$tableObj destroy
```

$refName	Reference name to copy to.
$object	Table object.
$value	Table value to assign.
$body	Tcl script to evaluate and set as table value.
$field	Field to query (fields "height" and "width" added).
$channelID	Open channel to print to.

Note that the methods `.=` and `:=` are also available, but they are not recommended for tables. Also note that "taboo" tables are always initialized, so the "exists" field of the object variable will always be true.

Example 3: Copying a table

Code:

```
$tableObj --> tableCopy
$tableCopy print
```

Output:

```
key {1 2 3 4 5} x {3.44 4.61 8.25 5.20 3.26} y {7.11 1.81 7.56 6.78 9.92} z {8.67 7.63 3.84
    1.11 4.56}
```

Get/Set Keyname

The keyname of a table can be accessed or modified directly with their respective methods.

```
$tableObj keyname <$keyname>
```

$keyname	Header for table keys. Default blank to return current name.

Get Keys and Fields

The table keys and fields are ordered lists of the row and column names of the table. They can be queried with the methods *keys* and *fields*, respectively.

```
$tableObj keys <$index>
```

```
$tableObj fields <$index>
```

$index	Index arguments, using "ndlist" index notation. Default ":" for all keys/fields.

Table Dimensions

The number of keys can be queried with *$tableObj height* and the number of fields can be queried with *$tableObj width*. These are also properties accessible with the standard method *info*. Note that rows and columns with missing data will be counted.

```
$tableObj height
```

```
$tableObj width
```

Example 4: Accessing table keys and table dimensions

Code:

```
puts [$tableObj keys]
puts [$tableObj keys 0:end-1]
puts [$tableObj height]
```

Output:

```
1 2 3 4 5
1 2 3 4
5
```

Get Table Data (Dictionary Form)

The method *data* returns the table data in unsorted dictionary form, where blanks are represented by missing dictionary entries.

```
$tableObj data <$key>
```

$key	Key to get row dictionary from (default returns all rows).

Get Table Data (Matrix Form)

The method *values* returns a matrix (list of rows) that represents the data in the table, where the rows correspond to the keys and the columns correspond to the fields. Missing entries are represented by blanks in the matrix unless specified otherwise.

```
$tableObj values <$filler>
```

$filler	Filler for missing values, default blank.

Example 5: Getting table data in dictionary and matrix form

Code:

```
puts [$tableObj data]
puts [$tableObj data 3]
puts [$tableObj values]
```

Output:

```
1 {x 3.44 y 7.11 z 8.67} 2 {x 4.61 y 1.81 z 7.63} 3 {x 8.25 y 7.56 z 3.84} 4 {x 5.20 y 6.78
    z 1.11} 5 {x 3.26 y 9.92 z 4.56}
x 8.25 y 7.56 z 3.84
{3.44 7.11 8.67} {4.61 1.81 7.63} {8.25 7.56 3.84} {5.20 6.78 1.11} {3.26 9.92 4.56}
```

Check Existence of Table Keys/Fields

The existence of a table key, field, or table value can be queried with the method *exists*.

```
$tableObj exists key $key
$tableObj exists field $field
$tableObj exists value $key $field
```

`$key`	Key to check.
`$field`	Field to check.

Get Row/Column Indices

The row or column index of a table key or field can be queried with the method *find*. If the key or field does not exist, returns an error.

```
$tableObj find key $key
$tableObj find field $field
```

`$key`	Key to find.
`$field`	Field to find.

Example 6: Find column index of a field

Code:

```
puts [$tableObj exists field z]
puts [$tableObj find field z]
```

Output:

```
1
2
```

Table Entry and Access

Data entry and access to a table object can be done with single values with the methods *set* and *get*, entire rows with *rset* and *rget*, entire columns with *cset* and *cget*, or in matrix fashion with *mset* and *mget*. If entry keys/fields do not exist, they are added to the table. Additionally, since blank values represent missing data, setting a value to blank effectively unsets the table entry, but does not remove the key or field.

Single Value Entry and Access

The methods *set* and *get* allow for easy entry and access of single values in the table. Note that multiple field-value pairings can be used in *$tableObj set*.

```
$tableObj set $key $field $value ...
```

```
$tableObj get $key $field <$filler>
```

$key	Key of row to set/get data in/from.
$field	Field of column to set/get data in/from.
$value	Value to set.
$filler	Filler to return if value is missing. Default blank.

Example 7: Setting multiple values

Code:

```
$tableObj --> tableCopy
$tableCopy set 1 x 2.00 y 5.00 foo bar
puts [$tableCopy data 1]
```

Output:

```
x 2.00 y 5.00 z 8.67 foo bar
```

Row Entry and Access

The methods *rset* and *rget* allow for easy row entry and access. Entry list length must match table width or be scalar. If entry list is blank, it will delete the row, but not the key.

```
$tableObj rset $key $row
```

```
$tableObj rget $key <$filler>
```

$key	Key of row to set/get.
$row	List of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Column Entry and Access

The methods *cset* and *cget* allow for easy column entry and access. Entry list length must match table height or be scalar. If entry list is blank, it will delete the column, but not the field.

```
$tableObj cset $field $column
```

```
$tableObj cget $field <$filler>
```

$field	Field of column to set/get.
$column	List of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Matrix Entry and Access

The methods *mset* and *mget* allow for easy matrix-style entry and access. Entry matrix size must match table size or be scalar.

```
$tableObj mset $keys $fields $matrix
```

```
$tableObj mget $keys $fields <$filler>
```

$keys	List of keys to set/get (default all keys).
$fields	List of keys to set/get (default all keys).
$matrix	Matrix of values (or scalar) to set.
$filler	Filler for missing values. Default blank.

Below is an example of how you can construct a table from scratch. Note also how you can create a table using the "vutil" command *new* instead of the command *table*.

Example 8: Matrix entry and access

Code:

```
::vutil::new table T
$T add keys 1 2 3 4
$T add fields A B
$T mset [$T keys] [$T fields] 0.0; # Initialize as zero
$T mset [$T keys 0:2] A {1.0 2.0 3.0}; # Set subset of table
puts [$T values]
```

Output:

```
{1.0 0.0} {2.0 0.0} {3.0 0.0} {0.0 0.0}
```

Iterating Over Table Data

Table data can be looped through, row-wise, with the method *with*. Variables representing the key values and fields will be assigned their corresponding values, with blanks representing missing data. The variable representing the key (table keyname) is static, but changes made to field variables are reflected in the table. Unsetting a field variable or setting its value to blank unsets the corresponding data in the table.

```
$tableObj with $body
```

$body	Code to execute.

Example 9: Iterating over a table, accessing and modifying field values

Code:

```
$tableObj --> tableCopy
set a 20.0
$tableCopy add fields q
$tableCopy with {
    puts [list $key $x]; # access key and field value
    set q [expr {$x*2 + $a}]; # modify field value
}
puts [$tableCopy cget q]
```

Output:

```
1 3.44
2 4.61
3 8.25
4 5.20
5 3.26
26.88 29.22 36.5 30.4 26.52
```

Note: Just like in *dict with*, the key variable and field variables in *$tableObj with* persist after the loop.

Field Expressions

The method *expr* computes a list of values according to a field expression. In the same style as referring to variables with the dollar sign ($), the "at" symbol (@) is used by *$tableObj expr* to refer to field values, or row keys if the keyname is used. If any referenced fields have missing values for a table row, the corresponding result will be blank as well. The resulting list corresponds to the keys in the table.

```
$tableObj expr $fieldExpr
```

$fieldExpr	Field expression.

Editing Table Fields

Field expressions can be used to edit existing fields or add new fields in a table with the method *fedit*. If any of the referenced fields are blank, the corresponding entry will be blank as well.

```
$tableObj fedit $field $fieldExpr
```

$field	Field to set.
$fieldExpr	Field expression.

Example 10: Using field expressions

Code:

```
$tableObj --> tableCopy
set a 20.0
puts [$tableCopy cget x]
puts [$tableCopy expr {@x*2 + $a}]
$tableCopy fedit q {@x*2 + $a}
puts [$tableCopy cget q]
```

Output:

```
3.44 4.61 8.25 5.20 3.26
26.88 29.22 36.5 30.4 26.52
26.88 29.22 36.5 30.4 26.52
```

Querying Keys that Match Criteria

The method *filter* returns the keys in a table that match criteria in a field expression.

```
$tableObj query $fieldExpr
```

$fieldExpr	Field expression that results in boolean value (true or false, 1 or 0).

Example 11: Getting keys that match criteria

Code:

```
puts [$tableObj query {@x > 3.0 && @y > 7.0}]
```

Output:

```
1 3 5
```

Filtering Table Based on Criteria

The method *filter* filters a table to the keys matching criteria in a field expression.

```
$tableObj filter $fieldExpr
```

$fieldExpr	Field expression that results in boolean value (true or false, 1 or 0).

Example 12: Filtering table to only include keys that match criteria

Code:

```
$tableObj --> tableCopy
$tableCopy filter {@x > 3.0 && @y > 7.0}
puts [$tableCopy keys]
```

Output:

```
1 3 5
```

Searching a Table

Besides searching for specific field expression criteria with *$tableObj query*, keys matching criteria can be found with the method *search*. The method *search* searches a table using the Tcl *lsearch* command on the keys or field values. The default search method uses glob pattern matching, and returns matching keys. This search behavior can be changed with the various options, which are taken directly from the Tcl *lsearch* command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation, with the exception of the -inline option. The -inline option filters a table based on the search criteria.

```
$tableObj search <$option ...> <$field> $value
```

$option ...	Searching options. Valid options:
-exact	Compare strings exactly
-glob	Use glob-style pattern matching (default)
-regexp	Use regular expression matching
-sorted	Assume elements are in sorted order
-all	Get all matches, rather than the first match
-not	Negate the match(es)
-ascii	Use string comparison (default)
-dictionary	Use dictionary-style comparison
-integer	Use integer comparison
-real	Use floating-point comparison
-nocase	Search in a case-insensitive manner
-increasing	Assume increasing order (default)
-decreasing	Assume decreasing order
-bisect	Perform inexact match
-inline	Filter table instead of returning keys.
--	Signals end of options
$field	Field to search. If blank, searches keys.
$value	Value or pattern to search for

Note: If a field contains missing values, they will only be included in the search if the search options allow (e.g. blanks are included for string matching, but not for numerical matching).

Sorting a Table

The method *sort* sorts a table by keys or field values. The default sorting method is in increasing order, using string comparison. This sorting behavior can be changed with the various options, which are taken directly from the Tcl *lsort* command. Therefore, while brief descriptions of the options are provided here, they are explained more in depth in the Tcl documentation. Note: If a field contains missing values, the missing values will be last, regardless of sorting options.

```
$tableObj sort <$option ...> <$field ...>
```

$option ...	Sorting options. Valid options:
-ascii	Use string comparison (default)
-dictionary	Use dictionary-style comparison
-integer	Use integer comparison
-real	Use floating comparison
-increasing	Sort the list in increasing order (default)
-decreasing	Sort the list in decreasing order
-nocase	Compare in a case-insensitive manner
--	Signals end of options
$field ...	Fields to sort by (in order of sorting). If blank, sorts by keys.

Example 13: Searching and sorting

Code:

```
$tableObj --> tableCopy
puts [$tableCopy search -real x 8.25]; # returns first matching key
$tableCopy sort -real x
puts [$tableCopy keys]
puts [$tableCopy cget x]; # table access reflects sorted keys
puts [$tableCopy search -sorted -bisect -real x 5.0]
```

Output:

```
3
5 1 2 4 3
3.26 3.44 4.61 5.20 8.25
2
```

Merging Tables

Data from other tables can be merged into the table object with *$tableObj merge*. In order to merge, all the tables must have the same keyname and fieldname. If the merge is valid, the table data is combined, with later entries taking precedence. Additionally, the keys and fields are combined, such that if a key appears in any of the tables, it is in the combined table.

```
$tableObj merge $object ...
```

$object ...	Other table objects to merge into table. Does not destroy the input tables.

Example 14: Merging data from other tables

Code:

```
$tableObj --> tableCopy
table new newTable
$newTable set 1 x 5.00 q 6.34
$tableCopy merge $newTable
$tableCopy print
```

Output:

```
key {1 2 3 4 5} x {5.00 4.61 8.25 5.20 3.26} y {7.11 1.81 7.56 6.78 9.92} z {8.67 7.63 3.84
    1.11 4.56} q {6.34 {} {} {} {}}
```

Table Manipulation

The following methods are useful for adding, removing, and rearranging rows and columns in a table.

Overwriting Keys/Fields

The method *define* overwrites the keys and fields of the table, filtering the data or adding keys and fields as necessary. For example, if the keys are defined to be a subset of the current keys, it will filter the data to only include the key subset.

```
$tableObj define keys $keys
$tableObj define fields $fields
```

`$keys`	Unique list of keys.
`$fields`	Unique list of fields.

Adding or Removing Keys/Fields

The method *add* adds keys or fields to a table, appending to the end of the key/field lists. If a key or field already exists it is ignored. The method *remove* removes keys or fields and their corresponding rows and columns from a table. If a key or field does not exist, it is ignored.

```
$tableObj add keys $key ...
$tableObj add fields $field ...
```

```
$tableObj remove keys $key ...
$tableObj remove fields $field ...
```

`$key ...`	Keys to add/remove.
`$field ...`	Fields to add/remove.

Inserting Keys/Fields

The method *insert* inserts keys or fields at a specific row or column index. Input keys or fields must be unique and must not already exist.

```
$tableObj insert keys $index $key ...
$tableObj insert fields $index $field ...
```

$index	Row/column index to insert at.
$key ...	Keys to insert.
$field ...	Fields to insert.

Renaming Keys/Fields

The method *rename* renames keys or fields. Old keys and fields must exist. Duplicates are not allowed in old and new key/field lists.

```
$tableObj rename keys $old $new
$tableObj rename fields $old $new
```

$old	Keys/fields to rename. Must exist.
$new	New keys/fields. Must be same length as $old.

Example 15: Renaming fields

Code:

```
$tableObj --> tableCopy
$tableCopy rename fields [string toupper [$tableCopy fields]]
puts [$tableObj fields]
puts [$tableCopy fields]
```

Output:

```
x y z
X Y Z
```

Moving Keys/Fields

Existing keys and fields can be moved with the method *move*.

```
$tableObj move key $key $index
$tableObj move field $field $index
```

$key	Key to move.
$field	Field to move.
$index	Row/column index to move to.

Swapping Keys/Fields

Existing keys and fields can be swapped with the method *swap*. To swap the a field column with the key column, use the method *mkkey*.

```
$tableObj swap keys $key1 $key2
$tableObj swap fields $field1 $field2
```

$key1 $key2	Keys to swap.
$field1 $field2	Fields to swap.

Example 16: Swapping table rows

Code:

```
$tableObj --> tableCopy
$tableCopy swap keys 1 4
$tableCopy print
```

Output:

```
key {4 2 3 1 5} x {5.20 4.61 8.25 3.44 3.26} y {6.78 1.81 7.56 7.11 9.92} z {1.11 7.63 3.84
    8.67 4.56}
```

Making a Field the Key of a Table

The method *mkkey* makes a field the key of a table, and makes the key a field. If a field is empty for some keys, those keys will be lost. Additionally, if field values repeat, only the last entry for that field value will be included. This method is intended to be used with a field that is full and unique, and if the keyname matches a field name, this command will return an error.

```
$tableObj mkkey $field
```

`$field`	Field to swap with key.

Transposing a Table

The method *transpose* transposes the table, making the keys the fields and the fields the keys.

```
$tableObj transpose
```

Example 17: Transposing a table

Code:

```
$tableObj --> tableCopy
$tableCopy transpose
$tableCopy print
```

Output:

```
key {x y z} 1 {3.44 7.11 8.67} 2 {4.61 1.81 7.63} 3 {8.25 7.56 3.84} 4 {5.20 6.78 1.11} 5
    {3.26 9.92 4.56}
```

Command Index