

TIP #185: Null Handling

Source: <http://coding.derkeiler.com/Archive/Tcl/comp.lang.tcl/2004-04/0382.html>

From: John H. Harris (JHHarris_at_valley.net)

Date: 04/12/04

Date: Mon, 12 Apr 2004 18:51:19 +0000 (UTC)

TIP #185: NULL HANDLING

=====

Version: \$Revision: 1.1 \$

Author: John H. Harris <JHHarris_at_valley.net>

State: Draft

Type: Project

Tcl-Version: 8.5

Vote: Pending

Created: Thursday, 08 April 2004

URL: <http://purl.org/tcl/tip/185.html>

WebEdit: <http://purl.org/tcl/tip/edit/185>

Post-History:

ABSTRACT

=====

Tcl lacks the ability to handle /nulls/, data with missing or unknown values. In this TIP I suggest a means for representing and propagating nulls, and command modifications for manipulating them.

RATIONALE

=====

Tcl deals with strings, the universal medium for representing data. It lacks, however, the intrinsic ability to represent /missing/ data, or /nulls/. A null datum (or just "null") is very different from an empty string. A database may represent Prince's last name as "" — his name is know and it is an empty string, but if a respondent forgets to give his weight on a questionnaire, he is not weightless; his weight is not ""; it is unknown, or /null/. Nulls are common in real-world data, particularly data obtained from interviews or questionnaires. Because of this, most modern database engines and statistics tools recognize nulls. A large fraction of the applications we are writing are client programs to such databases, though the user is not always aware of it.

The programmer can use whatever is provided with the database application program interface (API) he is using, but most Tcl APIs provide none, probably because Tcl itself lacks nulls.

A Tcl programmer writing an interface to a database must improvise something to deal with nulls. If the data representation domain does not include an empty string, as for an integer or date, then an empty string may suffice, but arbitrary-string data is common, so something fancier is often needed.

For example, here is an approach I used in a current project built on SQLite [1]. I prepend each nonnull string with an apostrophe. Thus, nulls are universally represented by an empty string. I accomplish this using the `/coalesce()/` function provided by SQLite, e.g., if I want

```
select id, name from persons
```

I must ask for

```
select id, coalesce('||name,") as name from persons
```

I wrote a layer that automates this ugly mess. To keep it simple (and because SQLite is untyped) it prepends `/all/` fields with an apostrophe, even ones that do not need it. I deal with the apostrophes later — testing for nullness is easy, but the apostrophe gets in the way for most other I/O and processing.

This is more than you wanted to know about my personal problems, but you can be sure this kind of thing is happening over and over with hundreds of programmers and applications.

Now let us imagine how it might be simplified, assuming Tcl null support and a revamped SQLite API:

```
set result [ db eval {select id, name from person}
foreach -null <unknown> { id name } $result {
  puts "$id: $name" }
```

I will explain more later, but basically, the database returns null information on `/every/` query, so there is no fancy SQL code. The `*-null*` option tells `*foreach*` how to represent a null when it assigns `/id/` and `/name/`.

SPECIFICATION

=====

It would be nice if null handling could be added to, or with, a single command suite. If we could just extend, say, the `*dict*` command, these objects would be ideal media for database APIs. Unfortunately, it seems impossible to implement any change without involving Tcl's string implementation, list syntax, and command interpreter.

REPRESENTING NULLS

Not all commands will understand nulls. We will speak of null-smart and null-dumb commands. All commands follow these rules:

1. Nulls represent a lack of information.
2. An empty string is not null.
3. Nulls can not be made "unnull", merely by being processed. This is equivalent to creating information out of thin air. Substituting a nonnull value for a null must be programmed explicitly.
4. For null-smart commands, nulls propagate. A null combined with any nonnull is null. Appending a null to a string, or substituting a null into a string nulls the entire string.
5. Logical comparisons with nulls with respect to magnitude or identity evaluate to null (i.e., /unknown/).
6. Null-dumb commands must treat nulls as empty strings.

STRINGS OBJECTS

An empty string is not a null. We have the further constraint that we must be able to handle any arbitrary binary string, including null (in the sense of 0x00) bytes. Clearly the implementation must be augmented; we must add a null flag. In implementation, null strings would probably have their string part set to an empty string, to accommodate null-dumb commands, as in rule 6 above. Thus, dumb commands can simply ignore the null flag.

LIST SYNTAX

It is easy to add a null flag to the string class. To be of any use, however, we need to pass a null string to commands, which means embedding nulls in lists. We can indicate this much in the same way the `/[expand]{ }/` syntax [TIP #157] works, by exploiting otherwise illegal list syntax. I propose using `*{null}!*` as null representation for lists. (Perhaps new syntax could be a rider on the `/[expand]{ }/` rule. I fear that increasing the number of rules could scuttle this TIP.) At present this syntax (like `{expand}{ }`) produces an "extra characters after close-brace" error.

Note that the string `"{null}!"` is /not/ interpreted as a null string, instead it is a nonnull string that is also a well-formed list with one null element.

INTERPRETER BEHAVIOR

If the command interpreter encounters the word "{null}!" in a list, this element is replaced by a null string in the array of strings passed to the command.

When the Tcl command interpreter encounters a word that contains a variable substitution, if the variable contains a null, the interpreter will behave as if the entire word were replaced with "{null}!", and pass a null string to the command in its place.

Command substitutions behave the same way when a command returns a null.

MANIPULATING NULLS

We need not be decided how commands should respond to nulls all at once. Indeed, it would be better to let this evolve as we gain experience with this new dimension of data. At first we will need at least some basic support to create, copy and test nulls.

THE STRING COMMAND

The simplest approach is simply to test for nullness. The obvious candidate for this test would be an addition to the `/string is/` command suite:

```
if { [ string is null $s ] } {  
    error {Missing data for s. } }
```

The string command could also be used to generate nulls:

```
set null [ string null ]
```

THE SET COMMAND

We can use `/set/` to create a variable that contains a null string:

```
% set s {null}!
```

Now `/s/` exists and has its null flag set. This is such a natural syntax that it probably make `/string null/` unnecessary.

We can retrieve a null value with `/set/` — suppose `/puts/` is dumb and sees the null as an empty string:

```
% puts "s equals [ set s ]"
```

`*Set*` returns a null. In this case, the null is in a command substitution, and it nulls the entire string being passed to `*puts*`, which, being null-dumb, outputs an empty string and newline. This may be unsatisfactory. We would rather substitute a nonnull, perhaps an

empty string, or "<NULL>".

When and what to substitute is an ad hoc programming choice, so should be an option. Here is the `/set/` command with an option that tells how to represent a null:

```
% set s {null}!
% set -null <NULL> s
<NULL>
% set -null huh? s
huh?
```

The default value for this option is a null, so `*[set s]*` can be used as a direct substitution for `*$s*`, as you would expect.

While we must allow for the worst case of representing a null amidst the set of arbitrary binary strings, in practical data this seldom occurs. When it does, we must resort to an explicit test and conditional execution, but more often there is some gap in the domain of valid data that can be used to represent a null. We have already seen the example of using an empty string for a null integer or date. For other data types there are better choices. The programmer knows these gaps and can choose a string that fits in the gap and is also easily understood by humans or other software.

The `*-null*` option should have the same meaning when applied to any `null-smart` command: if the return value is a null, change it to the option value.

The `set` command can use a similar option to assign a null. This time the option `*-nullify*` tells what value, by exact match, should be replaced by a null.

```
% set s NULL
NULL
% set -nullify NULL t $s
% puts '[ set -null Void t ]'
'Void'
```

Notice that string `/s/` above is *not* a null, it is the string "NULL". The second command translates it to a true null, and the third translates it to "Void".

The `*-nullify*` option provides a way of assigning a null to a variable that is independent of list syntax:

```
% set -nullify {} s {}
```

The `*-nullify*` option should have the same meaning when applied to any `null-smart` command: if an argument value is an exact match to the options value, change it to a null.

LIST COMMANDS

Most smart commands can just test the null flag of their arguments and take appropriate action. A few commands, such as `*lindex*`, `*lset*`, `*join*`, and `*split*` must understand list null syntax. Consider the string:

```
% set s {a {b0 {null}! b2} c}
```

This string represents a list whose second element is a list containing a null. We expect this behavior:

```
% lindex -null ~ $s 1 1
~
% lset -nullify {} s 1 2 {}
a {b0 {null}! {null}!} c
% lindex -null NUL $s 1 2
NUL
```

`*Join*` creates a string from a list and must understand `*-null*`. Notice that the second command below returns null, by rule 4.

```
% join -null void {a {null}!}
a void
% join {a {null}!}
```

`*Split*` creates a list from a string and must understand `*-nullify*`:

```
% split {a {null}!}
a {{null}!}
% split -nullify NULL {a NULL}
a {null}!
```

EXPR AND CONTROL-FLOW COMMANDS

Nulls can be tested using `*string is null*`, but testing occurs so often in practice that we need to have `*expr*` and the control commands behave properly, allowing three-valued logic — true, false, and null. (In this context, a logic value of null is often called "unknown".) `*Expr*` should recognise the `*-null*` option. These examples illustrate typical three-valued logic tautologies:

```
% set u {null}!
% expr -null unk { $u }
unk
% expr -null unk { $u && 1 }
1
% expr -null unk { $u && 0 }
unk
% expr -null unk { $u || 0 }
0
```

```
% expr -null unk { $u == $u }
unk
% expr -null unk { $u != 1 }
unk
% expr -null unk { $u eq { } }
unk
% expr -null unk { $u > 0 }
unk
```

Notice that, logic expressions containing nulls may may have nonnull results. This may seem like a violation of rule 5, but actually is just a special kind of lazy logic. **Expr** can simply ignore the null term because it is tautologously irrelevant.

The control commands **if**, **while**, and **for** all throw errors if the expression evaluates to null.

We can also use nulls to represent undefined mathematical results, or /NaN/s (not a number), in the terminology of IEEE 754 floating point arithmetic [2]. This allows Tcl to give the programmer access to hardware features that are currently hidden. To do this we need a **-nocomplain** option:

```
% expr -nocomplain -null nan { log(-1) }
nan
```

We can use the **-null** option to assign a usable value if /a/ goes to zero:

```
% set INF 1e16
% set x [ expr -nocomplain -null $INF { 5 / $a } ]
```

Nulls can propagate sensibly through a computation and give a useful result when, without them, the same expression would have throw an exception. Again, suppose /a/ goes to zero:

```
% expr -nocomplain -null unk { $x / $a > 0.54 || $c < 0.042 }
1
```

OTHER BASIC COMMANDS

We need not make all commands null-smart immediately. Old commands can treat nulls as empty strings and function as before. They may be less useful than they could be, but nothing need break, because old code contains nothing that would introduce nulls in the first place. It is up to the programmer who decides to use Tcl's null facility to be aware of which commands respond intelligently to nulls. Perhaps a greater danger is that, once nulls are introduced, evolving commands may break applications that use nulls. Because of this, it would be wise to choose the initial set of null-smart commands with care.

I propose modifying the following commands, if necessary, to respond to nulls appropriately: `*lset*`, `*foreach*`, `*format*`, the list commands, `*return*`, `*set*`, `*subst*`, `*string*`, `*switch*`. Except as noted below, this means recognizing the `*-null*` option.

Now let us discuss the modifications needed for these commands.

String command suite:

Except for `*string is*` commands, all `*string*` subcommands should return null if any argument is null, simply because the commands are meaningless when applied to a null string.

Note that a null string is not unequal to any given string.

Batman may or may not be Bruce Wayne -- his identity is unknown. Similarly, two null strings are neither equal nor unequal to each other. Batman may or may not be the same guy as the Lone Ranger.

String is: Except for `*string is null*`, all `*string is*` subcommands return 0 with a null argument.

Switch: The switch command should recognize the `*-null*` option. Having a switch leg that matches null is easy to imagine:

```
switch $s {
  a { do this }
  b { do that }
  {null}! { punt } }
```

but by rule 5 it would never execute -- nothing matches a null. Instead, use `*-null*`:

```
switch -null NULL $s {
  a { do this }
  b { do that }
  NULL { punt } }
```

If you must match any other nonnull string separately from nulls, use `*-glob*` and catch the null with `/default/`:

```
switch -glob -null NULL -- $s {
  a { do this }
  b { do that }
  * { any other nonnull }
  default { punt } }
```

Foreach: The `*Foreach*` command recognizes both `*-null*` and `*-nullify*`.

Format: The `*format*` command should understand the `*-null*` option.

The list commands:

In general, all the list commands treat nulls as distinct

elements. Nulls never disappear from a list. `*length*` includes them in the count. Commands that convert lists to or from strings (`*lappend*`, `*lindex*`, `*linsert*`, `*list*`, `*lset*`, `*split*`, `*join*`) should understand the `*-null*` option. Those that stay in the list domain (`*concat*`, `*lrange*`) should not.

`lsort`: If a list contains a null it cannot be sorted — `*lsort*` returns a null string. With the `*-null*` option, it can be sorted after substituting the `*-null*` option value.

ISSUES AND EXTENSIONS

=====

In this section we discuss design issues that are either debatable or not entirely resolved.

* Null attributes. It is common practice in statistics to have several distinct kinds of null. In the SAS language, a null datum can be represented by a period, '.', or a period followed by a letter, '.a'. Different codes represent different reasons for the null, e.g., `.a` = "no response", `.b` = "doubly-entry error", `.c` = "out of range".

Another use of a null attribute would be to have a value carry its string representation.

I think both these ideas are of dubious value and violate rule 1, but if, in the future, we want to add this functionality, it could be added with the syntax `{null}{ }`, where the second set of braces contain the attribute information.

* Choice of symbol.

`{null}!` or `NULL`, `void`, `nan`, `unk`, `unknown`, `{ }!`,

* SQL inconsistencies. The various implementations of SQL are inconsistent in their treatment of nulls. This is well documented on both the SQLite [1] and MySQL [4] web sites. I prefer a conservative, consistent mathematical interpretation. The `*-null*` and `*-nullify*` options go a long way toward simulating the inconsistencies of other systems, if necessary.

DISCUSSION

=====

REFERENCES

=====

[1]: SQLite

[2]: IEEE Floating point

[4]: MySQL

COPYRIGHT

=====

This document has been placed in the public domain.

TIP AutoGenerator – written by Donal K. Fellows

[[Send Tcl/Tk announcements to tcl-announce@mitchell.org
Announcements archived at http://groups.yahoo.com/group/tcl_announce/
Send administrivia to tcl-announce-request@mitchell.org
Tcl/Tk at <http://tcl.tk/>]]