Essentials

- Tk Backgrounder (../resources/backgrounder.html)
- Installing Tk (../tutorial/install.html)
- Tutorial (../tutorial/index.html)
- Widget Roundup (../widgets/index.html)
- Languages Using Tk (../resources/languages.html)
- Official Tk Command Reference
  (Tcl-oriented; at www.tcl.tk) (http://www.tcl.tk/man/tcl8.6/TkCmd/contents.htm)

Tutorial

Show: All Languages ▾

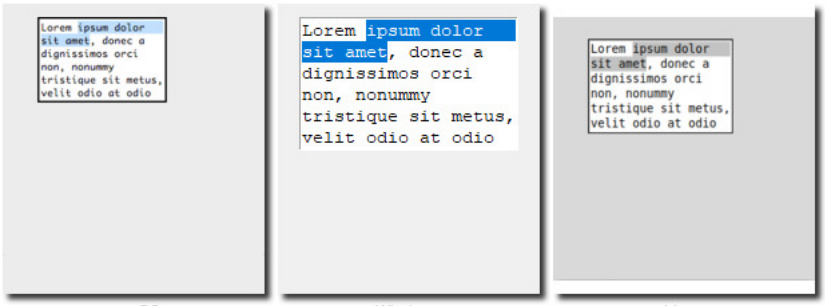
This tutorial will quickly get you up and running with the latest Tk from Tcl, Ruby, Perl or Python on Mac, Windows or Linux. It provides all the essentials about core Tk concepts, the various widgets, layout, events and more that you need for your application.

Previous: Canvas (canvas.html) | Contents (index.html) | Single Page (onepage.html) | Next: Tree (tree.html)

Text

- Widget Roundup (../widgets/text.html)
- Reference Manual (http://www.tcl.tk/man/tcl8.6/TkCmd/text.htm)

A **text** widget manages a multi-line text area. Like the canvas widget, Tk's text widget is an immensely flexible and powerful tool which can be used for a wide variety of tasks. Some example uses of text widgets have included anything from providing a simple multi-line text area as part of a form, to a stylized code editor, to an outliner, to a web browser. Text widgets are part of the classic Tk widgets, not the themed Tk widgets.



Text Widgets

Text widgets are created using the **`tk::text`** command:

```
tk::text .text -width 40 -height 10
```

Text widgets are created using the **`TkText`** class:

```
text = TkText.new(parent) {width 40; height 10}
```

Text widgets are created using the **`new_tk__text`** method, a.k.a. **`Tkx::tk__text`**:

```
$text = $parent->new_tk__text(-width => 40, -height => 10);
```

Text widgets are created using the **`Text`** function:

```
text = Text(parent, width=40, height=10)
```

While we briefly introduced text widgets in an earlier chapter, here we'll go into much more detail, to give you a sense of the level of sophistication it allows. Still, for any significant work with the text widget, the reference manual (http://www.tcl.tk/man/tcl8.6/TkCmd/text.htm) is very well organized and useful.

The Basics

If you just want to use the text widget to get a simple multi-line text from the user as part of a form, there's only a few things you'll need to worry about: creating and sizing the widget (check), providing an initial value for the text in the form, and retrieving the text in the widget after the user has submitted the form.

Providing Initial Content

When you first create it, the text widget has nothing in it, so if you want to provide an initial piece of text, you're going to have to add it yourself. Unlike for example the entry widget, text widgets don't support a `"textvariable"` configuration option; as we'll soon see, text widgets can contain a lot more than just plain text, so a simple variable isn't sufficient to hold it all.

Instead, to set the initial text for the widget, you'll use the widget's `"insert"` method:

```
.text insert 1.0 "here is my text to insert"
```

```
text.insert(1.0, "here is my text to insert")
```

```
$text->insert("1.0", "here is my text to insert");
```

```
text.insert('1.0', 'here is my text to insert')
```

The "1.0" here represents where to insert the text, and can be read as "line 1, character 0". This refers to the first character of the first line; for historical conventions related to how programmers normally refer to lines and characters, line numbers are 1-based, and character numbers are 0-based.

The text to insert is just a string. Because the widget can hold multi-line text, the string you supply can be multi-line as well. To do, simply embed `"\n"` (newline) characters in your string at the appropriate locations.

Scrolling

Scrollbars, both horizontal and vertical, can be attached to the text widget. This works exactly the same as using scrollbars in any other widget, such as a listbox or canvas.

You can also ask the widget to ensure that a certain part of the text is visible. For example, if you've added more text to the widget than will fit onscreen (so it will scroll) but want to ensure that the top of the text rather than the bottom is visible, call the `"see"` method, passing it the position of the text (e.g. "1.0").

Controlling Wrapping

What if some lines of text in the widget are very long, longer than the width of the widget? By default, the text will just wrap around to the next line, but if and how it does this can be controlled by the `"wrap"` configuration option. The default value is `"char"`, meaning wrap around right at the character at the end of the line; other options are `"word"` to cause wrapping, but only at word breaks (e.g. spaces), and `"none"` meaning don't wrap around at all. In the latter case, some of the text won't be visible unless you attach a horizontal scrollbar to the widget.

Disabling the Widget

Some forms will temporarily disable editing in particular widgets unless certain conditions are met (e.g. some other options are set to a certain value). To prevent the user from making any changes to a text widget, set the `"state"` configuration option to `"disabled"`; re-enable editing by setting this option back to `"normal"`.

Retrieving the Text

Finally, after the user has made any changes and submitted the form, your program will want to retrieve the contents of the widget, which is done with the `"get"` method:

```
set thetext [.text get 1.0 end]
```

```
thetext = t.get("1.0", 'end')
```

```
$thetext = $text->get("1.0", "end");
```

```
thetext = text.get('1.0', 'end')
```

Modifying the Text in Code

While the user can modify the text in the text widget interactively, your program can also make changes. Adding text is done with the `"insert"` method, which we used above to provide an initial value for the text widget.

Text Positions and Indices

When we specified a position of "1.0" (first line, first character), this was an example of an *index*. It tells the insert method where to insert the new text (just before the first line, first character, i.e. at the very start of the widget). There are a variety of ways to specify these indices. You've also seen another one: the `"end"` (from the "get" example) means just past the end of the text ("just past" because text insertions go right before the given index, so inserting at "end" will add text to the end of the widget). Note that Tk will *always* add a newline at the very end of the text widget.

Here are a few additional examples of indices, and what they mean:

`3.end`	The newline at the end of line 3.
`1.0 + 3 chars`	Three characters past the start of line 1.
`2.end -1 chars`	The last character before the new line in line 2.
`end -1 chars`	The newline that Tk always adds at the end of the text.
`end -2 chars`	The actual last character of the text.
`end -1 lines`	The start of the last actual line of text.
`2.2 + 2 lines`	The third character (index 2) of the fourth line of text.
`2.5 linestart`	The first character of line 2.
`2.5 lineend`	The position of the newline at the end of line 2.
`2.5 wordstart`	First char. of the word with the char. at index 2.5.
`2.5 wordend`	First char. after the word with the char. at index 2.5.

Some additional things to keep in mind:

- The term "chars" can be abbreviated as "c", and "lines" as "l."
- You can omit the spaces between the terms, e.g. "1.0+3c".
- If you specify an index past the end of the widget (e.g. "end + 100c") it will be interpreted as the end.
- Adding characters will wrap to the next lines as needed; e.g. "1.0 + 10 chars" on a line with only five characters will end up being on the second line.
- When using indices containing multiple words, make sure to quote them appropriately so that Tk sees the entire index as a single argument.
- When moving up or down a certain number of lines, this is interpreted as *logical* lines, where each line is terminated only by the "\n." With long lines and wrapping enabled, this may represent multiple lines on the display. If you'd like to move up or down a single line on the display, you can specify this as, e.g. "1.0 + 2 display lines".
- To determine the actual canonical position of an index, use the `"index"` method, passing it the index expression, and it will return the corresponding index in the form `"line.char."`
- You can compare two indices using the `"compare"` method, which lets you check for equality, whether one index is later in the text than the other, etc.

Deleting Text

While the `"insert"` method adds new text anywhere in the widget, the `"delete"` method removes it. You can specify either a single character to be deleted (by index) or a range of characters specified by the start and end index. In the latter case, characters from (and including) the start index through to *just before* the end index will be deleted (so the character at the end index is not deleted). So this would remove a single line of text (including its trailing newline) from the start of the text:

```
.text delete 1.0 2.0
```

```
text.delete(1.0, 2.0)
```

```
$text->delete("1.0", "2.0");
```

```
text.delete('1.0', '2.0')
```

There is also a `"replace"` method, taking a starting index, and ending index and a string as parameters. It does the same as a delete followed by an insert at the same location.

Example: Logging Window

Here's a short example illustrating how to use a text widget as an 80x24 logging window for your application. The user doesn't edit the text widget at all; instead, your program will write log messages to it. You'd like to keep the content to no more than 24 lines (so no scrolling), so as you add new messages at the end, you'll have to remove old ones from the top if there are already 24 lines.

```
package require Tk
grid [text .log -state disabled -width 80 -height 24 -wrap none]

proc writeToLog {msg} {
    set numlines [lindex [split [.log index "end - 1 line"] "."] 0]
    .log configure -state normal
    if {$numlines==24} {.log delete 1.0 2.0}
    if {[.log index "end-1c"]!="1.0"} {.log insert end "\n"}
    .log insert end "$msg"
    .log configure -state disabled
}
```

```
require 'tk'
root = TkRoot.new
@log = TkText.new(root) {state 'disabled';width 80;height 24;wrap 'none'}.grid

def writeToLog(msg)
    numlines = @log.index("end - 1 line").split('.')[0].to_i
    @log['state'] = :normal
    @log.delete(1.0, 2.0) if numlines==24
    @log.insert('end', "\n") unless @log.index('end-1c')=='1.0'
    @log.insert('end', msg)
    @log['state'] = :disabled
end
```

```
use Tkx;
$mw = Tkx::widget->new(".");

$log = $mw->new_tk__text(-state => "disabled", -width => 80, -height => 24, -wrap => "none");
$log->g_grid;

sub writeToLog {
    my ($msg) = @_;
    $numlines = $log->index("end - 1 line");
    print $numlines . "\n";
    $log->configure(-state => "normal");
    if ($numlines==24) {$log->delete("1.0", "2.0");}
    if ($log->index("end-1c")!="1.0") {$log->insert_end("\n");}
    $log->insert_end($msg);
    $log->configure(-state => "disabled");
}
```

```
from tkinter import *
from tkinter import ttk

root = Tk()
log = Text(root, state='disabled', width=80, height=24, wrap='none')
log.grid()

def writeToLog(msg):
    numlines = log.index('end - 1 line').split('.')[0]
    log['state'] = 'normal'
    if numlines==24:
        log.delete(1.0, 2.0)
    if log.index('end-1c')!='1.0':
        log.insert('end', '\n')
    log.insert('end', msg)
    log['state'] = 'disabled'
```

Note that because the widget was disabled, we had to reenable it to make any changes, even from our program.

Formatting with Tags

Up until now, we've just dealt with plain text. Now it's time to look at how to add special formatting, such as bold, italic, strikethrough, background colors, font sizes, and much more. Tk's text widget implements these using a feature called *tags*.

Tags are objects associated with the text widget. Each tag is referred to via a name chosen by the programmer. Each tag can have a number of different configuration options; these are things like fonts, colors, etc. that will be used to format text. Though tags are objects having state, they don't need to be explicitly created; they'll be automatically created the first time the tag name is used.

Adding Tags to Text

Tags can be associated with one or more ranges of text in the widget. As before, these are specified via indices; a single index to represent a single character, and a start and end index to represent the range from the start character to just before the end character. Tags can be added to ranges of text using the "`tag add`" method, e.g.

```
.text tag add highlightline 5.0 6.0
```

```
text.tag_add('highlightline', 5.0, 6.0)
```

```
$text->tag_add("highlightline", "5.0", "6.0");
```

```
text.tag_add('highlightline', '5.0', '6.0')
```

Tags can also be provided when inserting text by adding an optional parameter to the "`insert`" method, which holds a list of one or more tags to add to the text you're inserting, e.g.

```
.text insert end "new material to insert" "highlightline recent warning"
```

```
text.insert('end', 'new material to insert', 'highlightline recent warning')
```

```
$text->insert_end("new material to insert", "highlightline recent warning");
```

```
text.insert('end', 'new material to insert', ('highlightline', 'recent', 'warning'))
```

As the text in the widget is modified, whether by the user or your program, the tags will adjust automatically. For example, if you had tagged the text "the quick brown fox" with the tag "nounphrase", and then replaced the word "quick" with "speedy," the tag would still apply to the entire phrase.

Applying Formatting to Tags

Formatting is applied to tags via configuration options; these work similarly to configuration options for the entire widget. As an example:

```
.text tag configure highlightline -background yellow -font "helvetica 14 bold" -relief raised
```

```
text.tag_configure('highlightline', :background=>'yellow', :font=>'helvetica 14 bold', :relief=>'raised')
```

```
$text->tag_configure("highlightline", -background => "yellow", -font => "helvetica 14 bold", -relief => "raised");
```

```
text.tag_configure('highlightline', background='yellow', font='helvetica 14 bold', relief='raised')
```

The currently available configuration options for tags are: "`background`", "`bgstipple`", "`borderwidth`", "`elide`", "`fgstipple`", "`font`", "`foreground`", "`justify`", "`lmargin1`", "`lmargin2`", "`offset`", "`overstrike`", "`relief`", "`rmargin`", "`spacing1`", "`spacing2`", "`spacing3`", "`tabs`", "`tabstyle`", "`underline`", and "`wrap`". Check the reference manual for detailed descriptions of these. The "`tag cget`" method allows you to query the configuration options of a tag.

Because multiple tags can apply to the same range of text, there is the possibility for conflict (e.g. two tags specifying different fonts). A priority order is used to resolve these; the most recently created tags have the highest priority, but priorities can be rearranged using the "`tag raise`" and "`tag lower`" methods.

More Tag Manipulations

To delete a tag altogether, you can use the "`tag delete`" method. This also, of course, removes any references to the tag in the text. You can also remove a tag from a range of text using the "`tag remove`" method; even if that leaves no ranges of text with that tag, the tag object itself still exists.

The "`tag ranges`" method will return a list of ranges in the text that the tag has been applied to. There are also "`tag nextrange`" and "`tag prevrange`" methods to search forward or backward for the first such range from a given position.

The "`tag names`" method, called with no additional parameters, will return a list of all tags currently defined in the text widget (including those that may not be presently used). If you pass the method an index, it will return the list of tags applied to just the character at the index.

Finally, you can use the first and last characters in the text having a given tag as indices, the same way you can use "end" or "2.5". To do so, just specify "*`tagname`*`.first`" or "*`tagname`*`.last`".

Differences between Tags in Canvas and Text Widgets

While both canvas and text widgets support "tags" which can be used to apply to several objects, style them, and so on, these tags are not the same thing. There are important differences to take note of.

In canvas widgets, individual canvas items have configuration options that control their appearance. When you refer to a tag in a canvas, the meaning of that is identical to "all canvas items presently having that tag". The tag itself doesn't exist as a separate object. So in the following snippet, the last rectangle added will *not* be colored red.

```
.canvas itemconfigure important -fill red
.canvas create rectangle 10 10 40 40 -tags important
```

```
canvas.itemconfigure('important', :fill => 'red')
TkcRectangle.new(canvas, 10, 10, 40, 40, :tags => 'important')
```

```
$canvas->itemconfigure("important", -fill => "red");
$canvas->create_rectangle(10, 10, 40, 40, -tags => "important");
```

```
canvas.itemconfigure('important', fill='red')
canvas.create_rectangle(10, 10, 40, 40, tags=('important'))
```

In text widgets by contrast, it's not the individual characters that retain the state information about appearance, but tags, which are objects in their own right. So in this snippet, the newly added text *will* be colored red.

```
.text insert end "first text" important
.text tag configure important -foreground red
.text insert end "second text" important
```

```
text.insert('end', 'first text', 'important')
text.tag_configure('important', :foreground => 'red')
text.insert('end', 'second text', 'important')
```

```
$text->insert_end("first text", "important");
$text->tag_configure("important", -foreground => "red");
$text->insert_end("second text", "important");
```

```
text.insert('end', 'first text', ('important'))
text.tag_configure('important', foreground='red')
text.insert('end', 'second text', ('important'))
```

Events and Bindings

One quite cool thing is that you can define event bindings on tags. That allows you to do things like easily recognize mouse clicks just on particular ranges of text, and popup up a menu or dialog in response. Different tags can have different bindings, so it saves you the hassle of sorting out questions like "what does a click at this location mean?". Bindings on tags are implemented using the "`tag bind`" method:

```
.text tag bind important <1> "popupImportantMenu"
```

```
text.tag_bind('important', '1', proc{popupImportantMenu})
```

```
$text->tag_bind("important", "<1>", sub{popupImportantMenu});
```

```
text.tag_bind('important', '<1>', popupImportantMenu)
```

Widget-wide binding to events works as it does for every other widget. Besides the normal low-level events, there are also two virtual events that will be generated: <Modified> whenever a change is made to the content of the widget, and <Selection> whenever there is a change made to which text is selected.

Selecting Text

Your program may want to know if a range of text has been selected by the user, and if so, what that range is. For example, you may have a toolbar button to bold the selected text in an editor. While you can tell when the selection has changed (e.g. to update whether or not the bold button is active) via the <Selection> virtual event, that doesn't tell you what has been selected.

The text widget automatically maintains a tag named "sel," which refers to the selected text. Whenever the selection changes, the "sel" tag will be updated. So you can find the range of text selected using the "`tag ranges`" method, passing it "sel" as the tag to report on.

Similarly, you can change the selection by using "`tag add`" to set a new selection, or "`tag remove`" to remove the selection. You can't actually delete the "sel" tag of course.

> *Though the default widget bindings prevent this from happening, "sel" is like any other tag in that it can support multiple ranges, i.e. disjoint selections. To prevent this from happening when changing the selection from your code, make sure you remove any old selection before adding a new one.*

The text widget manages the concept of the insertion cursor (where newly typed text will appear) separate from the selection. It does so using a new concept called a *mark*.

Marks

Marks are used to indicate a particular place in the text. In that respect, they are like indices, except that as the text is modified, the mark will adjust to be in the same relative location. In that way, they resemble tags, but refer to a single position rather than a range of text. Marks actually don't refer to a position occupied by a character in the text but specify a position *between* two characters.

Tk automatically maintains two different marks. The first is named "insert," and is the present location of the insertion cursor. As the cursor is moved (via mouse or keyboard), the mark moves with it. The second mark is named "current," and reflects the position of the character underneath the current mouse position.

To create your own marks, use the widget's `"mark set"` method, passing it the name of the mark and an index (the mark is positioned just before the character at the given index). This is also used to move an existing mark to a different position. Marks can be removed using the `"mark unset"` method, passing it the name of the mark. If you delete a range of text containing a mark, that also removes the mark.

The name of a mark can also be used as an index (in the same way "1.0" or "end-1c" are indices). You can find the next mark (or previous one) from a given index in the text using the `"mark next"` or `"mark previous"` methods. The `"mark names"` method will return a list of the names of all marks.

Marks also have a *gravity*, which can be modified with the `"mark gravity"` method, which affects what happens when text is inserted at the mark. Suppose we have the text "ac", with a mark in between that we'll symbolize with a pipe, i.e. "a|c." If the gravity of that mark is "right" (the default) the mark will attach itself to the "c." If the new text "b" is inserted at the mark, the mark will remain stuck to the "c," and so the new text will be inserted before the mark, i.e. "ab|c." If the gravity is instead "left," the mark will attach itself to the "a," and so new text will be inserted after the mark, i.e. "a|bc."

Images and Widgets

Like canvas widgets, text widgets can contain not only text but also images and any other Tk widgets (including a frame itself containing many other widgets). In some senses, this allows the text widget to work as a geometry manager in its own right. The ability to add images and widgets within the text opens up a world of possibilities for your program.

Images are added to a text widget at a particular index, with the image normally specified as an existing Tk image. There are also other options that allow you to fine-tune padding and so on.

```
image create photo flowers -file flowers.gif
.text image create sel.first -image flowers
```

```
flowers = TkPhotoImage.new(:file => 'flowers.gif')
TkTextImage.new(text, 'sel.first', :image => flowers)
```

```
Tkx::image_create_photo( "flowers", -file => "flowers.gif");
$text->image_create("sel.first", -image => "flowers");
```

```
flowers = PhotoImage(file='flowers.gif')
text.image_create('sel.first', image=flowers)
```

Widgets are added to a text widget pretty much the same way as images. The widget you're adding should be a descendant of the text widget in the overall window hierarchy.

```
ttk::button .text.b -text "Push Me"
.text window create 1.0 -window .text.b
```

```
b = Tk::Tile::Button.new(text) {text 'Push Me'}
TkTextWindow.new(text, 1.0, :window => b)
```

```
$b = $text->new_ttk__button(-text => "Push Me");
$text->window_create("1.0", -window => $b);
```

```
b = ttk.Button(text, text='Push Me')
text.window_create('1.0', window=b)
```

Even More

There are many more things that the text widget can do; here we'll briefly mention just a few more of them. For details on using any of these, see the reference manual.

Search

The text widget includes a powerful `"search"` method which allows you to locate a piece of text within the widget; this is useful for a "Find" dialog, as one obvious example. You can search backwards or forwards from a particular position or within a given range, specify your search using exact text, case insensitive, or using regular expressions, find one or all occurrences of your search term, and much more.

Modifications, Undo and Redo

The text widget keeps track of whether or not changes have been made to the text (useful to know whether you need to save it for example), which you can query (or change) using the `"edit modified"` method. There is also a complete multi-level undo/redo mechanism, managed automatically by the widget when you set its `"undo"` configuration option to true. Calling `"edit undo"` or `"edit redo"` then will modify the current text using information stored on the undo/redo stack.

Eliding Text

You can actually include text in the widget that is not displayed; this is known as "elided" text, and is made available using the `"elide"` configuration option for tags. You can use this to implement an outliner, a "folding" code editor, or even just to bury some extra meta-data intermixed with your text. When specifying positioning with elided text you have to be a bit more careful, and so commands that deal with positions have extra options to either include or ignore the elided text.

Introspection

Like most Tk widgets, the text widget goes out of its way to expose information about its internal state; we've seen most of this in terms of the `"get"` method, widget configuration options, `"names"` and `"cget"` for both tags and marks, and so on. There is even more information available that you can use for a wide variety of tasks. Check out the `"debug"`, `"dlineinfo"`, `"bbox"`, `"count"` and `"dump"` methods in the reference manual.

Peering

The Tk text widget allows the same underlying text data (containing all the text, marks, tags, images, and so on) to be shared between two or more different text widgets. This is known as *peering* and is controlled via the `"peer"` method.

Previous: Canvas (canvas.html) | Contents (index.html) | Single Page (onepage.html) | Next: Tree (tree.html)
