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: Tree (tree.html) | Contents (index.html) | Single Page (onepage.html) | Next: Case Study: IDLE Modernization (idle.html)

Styles and Themes

The "themed" aspect of the new Ttk widgets is one of the most powerful and exciting aspects of the new widget set. Yet because it does things quite differently from how Tk has traditionally worked, and because in trying to be flexible it does a *lot* of things, it's certainly the most confusing for many people.

Definitions

We'll first define a few concepts and terms that Ttk themes and styles rely on.

Widget Class

A *widget class* is used by Tk to identify the type of a particular widget; essentially, whether it is a button, a label, a canvas, etc. In classic Tk, all buttons had the same class ("Button"), all labels had the same class ("Label"), etc.

You could use this widget class both for introspection and for changing options globally via the option database. This could let you say, for example, that all buttons by default had a red background.

There were a *few* classic Tk widgets, including frame and toplevel widgets, which would allow you to *change* the widget class of a particular widget when the widget was first created, by passing it a `"class"` configuration option. So while normally frames would have a widget class of "Frame", you could specify that one particular frame widget had a widget class of "SpecialFrame".

Because of that, you could use the option database to define different looks for *different types* of frame widgets (not just all frame widgets, or frame widgets located at a particular place in the hierarchy).

What Ttk does is take that simple idea and give it rocket boosters.

Widget State

A *widget state* allows a single widget to have more than one appearance or behavior, depending on things like mouse position, different state options set by the application, and so on. In classic Tk, several widgets had `"state"` configuration options which allowed you to set them to "normal" or "disabled"; a "disabled" state for a button, for example, would draw its label greyed out. Some used an additional state, "active," again which represented a different behavior.

The widget itself, or more typically, the widget's class bindings, controlled how the appearance of the widget changed in different states, typically via consulting widget configuration options like `"foreground"`, `"activeforeground"`, and `"disabledforeground"`.

Ttk again extends and generalizes this basic idea of widget state, in two important ways. First, rather than being widget-specific, all Ttk widgets have state options, and in fact, all have exactly the same state options, accessed by the `"state"` and `"instate"` widget commands.

A Ttk widget state is actually a set of independent state flags, containing zero or more of the following flags: `"active"`, `"disabled"`, `"focus"`, `"pressed"`, `"selected"`, `"background"`, `"readonly"`, `"alternate"`, or `"invalid"` (see the widget (http://www.tcl.tk/man/tcl8.6/TkCmd/ttk_widget.htm) page in the reference manual for exact meanings).

Note that while all of these state flags are available for every widget, they may not be used by each widget. For example, a label widget is likely to ignore an `"invalid"` state flag, and so no special appearance would be associated with that flag.

The second major change that Ttk makes is that it takes the decision of what to change when the state is adjusted out of the widget's control. That is, a widget author will no longer hardcode logic to the effect of "when the state is disabled, consult the disabledforeground configuration option and use that for the foreground color." With that logic hardcoded, not only did it make coding widgets longer (and more repetitive) but also restricted how a widget could be changed based on its state. That is, if the widget author hadn't coded in logic to change the font when the state changed, you as the user of the widget were out of luck.

Instead of hardcoding these decisions within each widget, Ttk moves the decisions into a separate location: styles. This means that the widget author doesn't need to provide code for every possible appearance option, which not only simplifies the widget but paradoxically ensures that a wider range of appearances *can* be set, including those the widget author may not have anticipated.

Style

That brings us then to define a widget style. Very simply, a *style* describes the appearance (or appearance*s*) of a Ttk widget class. All widgets created with that widget class will have the same appearance(s). While each themed widget has a default class (e.g. "TButton" for "ttk::button" widgets), you can, unlike in classic Tk, assign a different widget class for any themed widget you create. This is done using Ttk's `"style"` configuration option, which all themed widgets support.

So a style defines the normal appearance of widgets of a certain widget class, but can also define variations of that appearance that depend on the current state flag. For example, a style can specify that when the "pressed" state flag is set, the appearance should change in a particular way. Because of this, a style can describe one or more ways for the widget to appear, depending on the state.

The rest of this chapter will delve into far more detail of what a style actually is, but at least now you know the responsibility it has.

Themes

You can think of a *theme* as a collection of styles. While each style is widget-specific (one for buttons, one for entries, etc.) a theme will collect many styles together. Typically, a theme will then define one style for each type of widget, but each of those styles will be designed so that they visually "fit" together with each other — though perhaps, unfortunately, Ttk doesn't technically restrict bad design or judgement!

To use a particular theme for an application is really to say that you'd like to have a set of styles defined so that by default all the different type of widgets will have some common appearance, and fit in well with each other.

Using Styles and Themes

So now we know what styles and themes are supposed to do, but how exactly do we use them? To do this, we need to know how to refer to styles and themes, and how to apply them to a widget or user interface.

Style Names

Every style has a name. If you're going to modify a style, create a new one, or use a style for a widget, you need to know its name.

How do you know what the names of the styles are? If you have a particular widget, and you want to know what style it is currently using, you can first check the value of its `"style"` configuration option. If that is empty, it means the widget is using the *default* style for the widget. You can retrieve that via the widget's class. For example:

```
% ttk::button .b
.b
% .b cget -style     # empty string as a result
% winfo class .b
TButton
```

```
>> b = Tk::Tile::Button.new(parent)
=> #
>> b['style']
=> []
>> TkWinfo.classname(b)
=> "TButton"
```

```
Perl> $b = $mw->new_ttk__button()
.b
Perl> $b->cget(-style)    # empty string as a result
Perl> Tkx::winfo('class', $b)
TButton
```

```
>>> b = ttk.Button()
>>> b['style']
''
>>> b.winfo_class()
'TButton'
```

So in this case, the style that is being used is "TButton". The default styles for other themed widgets are named similarly, e.g. "TEntry", "TLabel", "TSizeGrip", etc. It's always wise to check specifics though; for example, the treeview widget's class is "Treeview", not "TTreeview".

Beyond the default styles though, styles can be named pretty much anything. You might create your own style (or use a theme that has a style) named "FunButton", "NuclearReactorButton", or even "GuessWhatIAm" (not a smart choice). More often, you'll find names like "Fun.TButton" or "NuclearReactor.TButton", which suggest variations of a base style; as you'll see, this is something Ttk supports for creating and modifying styles.

The ability to retrieve a list of all currently available styles is currently not supported.

Using a Style

To use a style means to apply that style to an individual widget. If you know the name of the style you want to use, and which widget to apply it to, it's easy. Setting the style can be done at creation time:

```
ttk::button .b -text "Hello" -style "Fun.TButton"
```

```
b = Tk::Tile::Button.new( parent ) {text "Hello"; style "Fun.TButton"}
```

```
$b = $parent->new_ttk__button(-text => "Hello", -style => "Fun.TButton");
```

```
b = ttk.Button( parent , text='Hello', style='Fun.TButton')
```

As well, you can change the style of a widget at any time after you've created it with the `"style"` configuration option:

```
.b configure -style "NuclearReactor.TButton"
```

```
b['style'] = "NuclearReactor.TButton"
```

```
$b->configure(-style => "NuclearReactor.TButton");
```

```
b['style'] = 'NuclearReactor.TButton'
```

Using Themes

While styles control the appearance of individual widgets, themes control the appearance of the entire user interface. The ability to switch between themes is one of the significant features of the themed widgets.

Like styles, themes are identified by a name. You can obtain the names of all available themes:

```
% ttk::style theme names
aqua clam alt default classic
```

```
>> Tk::Tile::Style.theme_names
=> ["aqua", "step", "clam", "alt", "default", "classic"]
```

```
Perl> Tkx::ttk__style_theme_names()
aqua clam alt default classic
```

```
>>> s = ttk.Style()
>>> s.theme_names()
('aqua', 'step', 'clam', 'alt', 'default', 'classic')
```

Only one theme can ever be active at a time. To obtain the name of the theme currently in use, you can use the following:

This API, which was originally targeted for Tk 8.6, was back-ported to Tk 8.5.9. If you're using an earlier version of Tk getting this info is a bit trickier.

```
% ttk::style theme use
aqua
```

```
>> Tk.tk_call('ttk::style', 'theme', 'use')
=> "aqua"
```

As of Ruby 1.9.3-p0, the Tk::Tile::Style.theme_use API hadn't been updated to allow querying the current style. This will probably change soon.

```
Perl> Tkx::ttk__style_theme_use()
aqua
```

```
>>> s.theme_use()
'aqua'
```

Switching to a new theme can be done with:

```
ttk::style theme use themename
```

```
Tk::Tile::Style.theme_use "themename"
```

```
Tkx::ttk__style_theme_use("themename");
```

```
s.theme_use('themename')
```

What does this actually do? Obviously, it sets the current theme to the indicated theme. Doing this, therefore, replaces all the currently available styles with the set of styles defined by the theme. Finally, it refreshes all widgets, so that they take on the appearance described by the new theme.

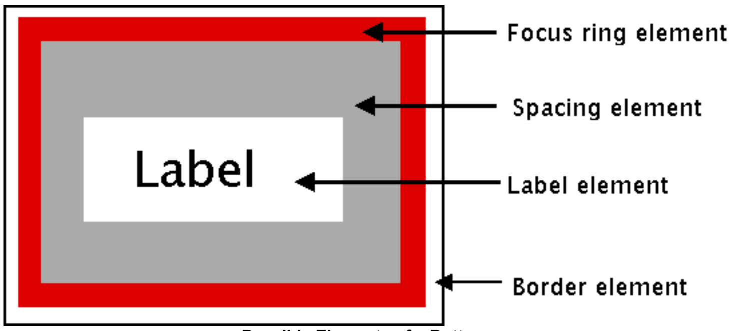
What's Inside a Style?

If all you want to do is *use* a style, you now know everything you need. If however, you want to create your own styles or modify an existing one, now it gets "interesting".

Elements

While each style represents a single widget, each widget is normally composed of smaller pieces, called *elements*. It's the job of the style author to construct the entire widget out of these smaller elements. What these elements are depends on the widget.

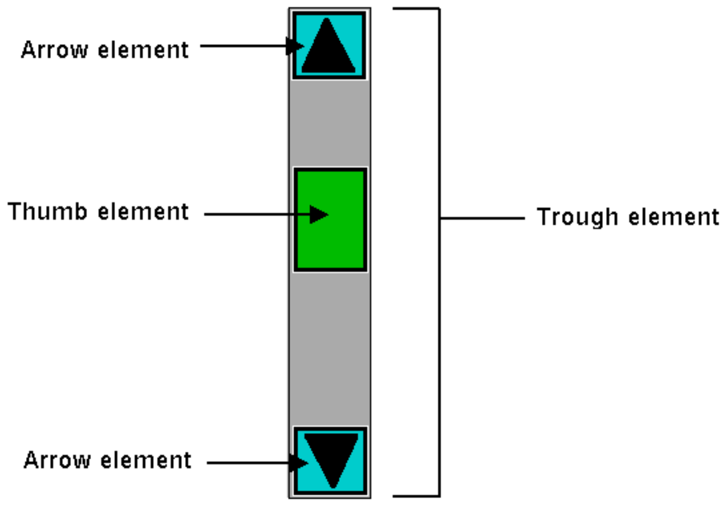
Here's an example of a button. It might have a border on the very outside, which is one element. Just inside that, there may be a focus ring, which is normally just the background color but may be highlighted when the user tabs into the button. So that's a second element. Then there might be some spacing between that focus ring and the button's label. So that spacing would be a third element. Finally, there is the label of the button itself, a fourth element.



Possible Elements of a Button.

Why might the style author have divided it up that way? If you have one part of the widget that may be in a different location from another or might be a different color than another, it may be a good candidate for an element. Note that this is just one example of how a button could be constructed from elements. Different styles and themes could (and do) accomplish this in different ways.

Here is a second example of a vertical scrollbar, containing a "trough" element containing the rest, which includes the up and down arrow elements at either end and a "thumb" element in the middle.



Possible Elements of a Scrollbar.

Layout

Besides the choice of which elements are part of a widget, a style also defines how those elements are arranged within the widget, or in other words, their layout. In the button example, we had a label element inside a spacing element, inside a focus ring element, inside a border element. So the logical layout is like this:

```
border {
    focus {
        spacing {
            label
        }
    }
}
```

We can ask Ttk what the layout of the TButton style is like this:

```
% ttk::style layout TButton
Button.border -sticky nswe -border 1 -children {Button.focus -sticky nswe
-children {Button.spacing -sticky nswe -children {Button.label -sticky nswe}}}
```

```
>> Tk::Tile::Style.layout('TButton')
=> [["Button.border", {"children"=>[["Button.focus", {"children"=>[["Button.spacing",
{"children"=>[["Button.label", {"sticky"=>"nswe"}]], "sticky"=>"nswe"}]],
"sticky"=>"nswe"}]], "sticky"=>"nswe", "border"=>"1"}]]
```

```
Perl> Tkx::ttk__style_layout('TButton')
Button.button -sticky nswe -border 1 -children {Button.padding -sticky nswe
-children {Button.spacing -sticky nswe -children {Button.label -sticky nswe}}}
```

```
>>> s.layout('TButton')
[("Button.border", {"children": [("Button.focus", {"children": [("Button.spacing",
{"children": [("Button.label", {"sticky": "nswe"})], "sticky": "nswe"})],
"sticky": "nswe"})], "sticky": "nswe", "border": "1"})]
```

If we clean this up and format it a bit, we get something with this structure:

```
Button.border -sticky nswe -border 1 -children {
    Button.focus -sticky nswe -children {
        Button.spacing -sticky nswe -children {
            Button.label -sticky nswe
        }
    }
}
```

This starts to make sense; we have four elements, named "Button.border", "Button.focus", "Button.spacing", and "Button.label". Each of these has different element options, such as "-children", "-sticky", and "-border" that here specify layout or sizes. Without getting into two much detail at this point, we can clearly see the nested layout, based on the "-children" and "-sticky" attributes; Ttk uses a simplified version of Tk's "pack" geometry manager to specify element layout.

Element Options

Each of these different elements may have a number of different options. For example, the thumb of the scrollbar may have an option to set its background color, or another to provide the width of a border, if any. These can be customized to adjust how the elements within the widget look.

What options are available for each element? Here's an example of checking what options are available for the label inside the button (which we know from the "layout" command is identified as "Button.label"):

```
% ttk::style element options Button.label
-compound -space -text -font -foreground -underline -width -anchor -justify
-wraplength -embossed -image -stipple -background
```

```
>> Tk::Tile::Style.element_options("Button.label")
=> ...
```

```
Perl> Tkx::ttk__style_element_options("Button.label")
-compound -space -text -font -foreground -underline -width -anchor -justify
-wraplength -embossed -image -stipple -background
```

```
>>> s.element_options('Button.label')
('-compound', '-space', '-text', '-font', '-foreground', '-underline', '-width', '-anchor', '-justify',
'-wraplength', '-embossed', '-image', '-stipple', '-background')
```

Presumably these options shouldn't have the leading dash on them.

In the next sections, we'll take a look at the not-entirely-straightforward way that you can work with element options.

Changing Style Options

In this section, we'll look at how you can change the style's appearance via modifying style options. You can do this either by modifying an existing style, or more typically, by creating a new style.

Modifying a Style Option

Modifying a configuration option for an existing style is done in a similar fashion as modifying any other configuration option, by specifying the style, name of the option, and new value:

```
ttk::style configure TButton -font "helvetica 24"
```

```
Tk::Tile::Style.configure('TButton', {"font" => "helvetica 24"})
```

```
Tkx::ttk__style_configure("TButton", -font => "helvetica 24");
```

```
s.configure('TButton', font='helvetica 24')
```

You'll learn more about what the valid options are shortly.

If you need to retrieve the current value of an option, this can be done with the `"lookup"` method.

```
% ttk::style lookup TButton -font
helvetica 24
```

```
>> Tk::Tile::Style.lookup('TButton', 'font')
=> "helvetica 24"
```

```
Perl> Tkx::ttk__style_lookup("TButton", "-font")
helvetica 24
```

```
>>> s.lookup('TButton', 'font')
'helvetica 24'
```

Creating a New, Derived Style

If you modify an existing style, such as "TButton", that modification will apply to all widgets using that style (so by default, all buttons). That may well be what you want to do.

More often, you're interested in creating a new style that is similar to an existing one but varies in a certain aspect. For example, you'd like to have most of the buttons in your application keep their normal appearance, but create certain special "emergency" buttons, which will be highlighted in a different way. In this case, creating a new style (e.g. "Emergency.TButton") derived from the base style ("TButton") would be the appropriate thing to do.

By prepending another name ("Emergency") followed by a dot onto an existing style, you are implicitly creating a new style derived from the existing one. So in this example, our new style will have exactly the same options as a regular button except for the indicated differences:

```
ttk::style configure Emergency.TButton -font "helvetica 24"
-foreground red -padding 10
```

```
Tk::Tile::Style.configure('Emergency.TButton', {"font" => "helvetica 24",
"foreground" => "red", "padding" => 10})
```

```
Tkx::ttk__style_configure("Emergency.TButton", -font => "helvetica 24",
-foreground => "red", -padding => 10);
```

```
s.configure('Emergency.TButton', font='helvetica 24',
foreground='red', padding=10)
```

State Specific Style Options

Besides the normal configuration options for the style, the widget author may have specified different options to use when the widget is in a particular widget state. For example, when a button is disabled, you'd like to have the color of the button's label greyed out.

To do this, you can specify a "map", which allows you to specify variations for one or more of a style's configuration options. For each configuration option, you can specify a list of widget states, along with the particular value the option should be assigned when the widget is in that state.

> *Remember that the state is composed of one or more state flags (or their negation), as set by the widget's `"state"` method, or queried via the `"instate"` method.*

The following example provides for the following variations from a button's "normal" appearance:

- when the widget is in the disabled state, the background color should be set to "#d9d9d9"
- when the widget is in the active state (mouse over it), the background color should be set to "#ececec"
- when the widget is in the disabled state, the foreground color should be set to "#a3a3a3" (this is in addition to the background color change we already noted)
- when the widget is in the state where the button is pressed and the widget is not disabled, the relief should be set to "sunken"

```
ttk::style map TButton \
        -background [list disabled #d9d9d9  active #ececec] \
        -foreground [list disabled #a3a3a3] \
        -relief [list {pressed !disabled} sunken] \
        ;
```

```
Tk::Tile::Style.map ("TButton",
    "background" => ["disabled"=>"#d9d9d9","active=>"#ececec"],
    "foreground" => ["disabled"=>"#a3a3a3"],
    "relief => ["pressed !disabled"=>"sunken"]);
```

PERLTODO

```
s.map('TButton',
    background=[('disabled','#d9d9d9), ('active','#ececec)],
    foreground=[('disabled','#a3a3a3')],
    relief=[('pressed', '!disabled', 'sunken')])
```

Remember that in the past, with classic Tk widgets, exactly what changed when the widget was in each state would have been determined solely by the widget author. With themed widgets, it is the style itself that determines what changes, which could include things that the original widget author had never anticipated.

> *Because widget states can contain multiple flags, it's possible that more than one state will match for an option (e.g. "pressed" and "pressed !disabled" will both match if the widget's "pressed" state flag is set). The list of states is evaluated in the order you provide in the map command, with the first state in the list that matches being used.*

Sound Difficult to you?

So you now know that styles are made up of elements, which have a variety of options, and are composed together in a particular layout. You can change various options on styles, to make all widgets using the style appear differently. Any widgets using that style take on the appearance that the style defines. Themes collect an entire set of related styles, making it easy to change the appearance of your entire user interface.

So what makes styles and themes so difficult in practice? Three things. First:

> **You can only modify options for a style, not element options (except sometimes).**

We talked earlier about how to discover what elements were used in the style by examining the style's layout, and also how to discover what options were available for each element. But when we went to make changes to a style, we seemed to be configuring an option for the style, without specifying an individual element. What's going on?

Again, using our button example, we had an element "Button.label", which among other things had a `"font"` configuration option. What happens is that when that "Button.label" element is drawn, it looks at the `"font"` configuration option set on the *style* to determine what font to draw itself in.

> *To understand why, you need to know that when a style includes an element as a piece of it, that element does not maintain any (element-specific) storage. In particular, it does not store any configuration options itself. When it needs to retrieve options, it does so via the containing style, which is passed to the element. Individual elements, therefore, are "flyweight" objects in GoF pattern parlance.*

Similarly, any other elements will lookup their configuration options from options set on the style. What if two elements use the same configuration option (like a background color)? Because there is only one background configuration option, stored in the style, that means both elements will use the same background color. You can't have one element use one background color, and the other use a different background color.

> *Except when you can. There are a few nasty, widget-specific things called "sublayouts" in the current implementation which let you sometimes modify just a single element, via configuring an option like "TButton.Label" (rather than just "TButton", the name of the style). Are the cases where you can do this documented? Is there some way to introspect to determine when you can do this? No to both. This is one area of the themed widget API that I definitely expect to evolve over time.*

The second difficulty is also related to modifying style options:

> **Options that are available don't necessarily have an effect, and it's not an error to modify a bogus option.**

You'll sometimes try to change an option that is supposed to exist according to element options, but it will have no effect. As an example, you can't modify the background color of a button in the "aqua" theme used by macOS. While there are valid reasons for these cases, at the moment it's not easy to discover them, which can make experimenting frustrating at times.

Perhaps more frustrating when you're experimenting is that specifying an "incorrect" style name or option name does not generate an error. When doing a "configure" or "lookup" you can, in fact, specify any name at all for a style, and specify any name at all for an option. So if you're bored with the "background" and "font" options, feel free to configure a "dowhatimean" option. It may not do anything, but it's not an error. Again, it may make it hard to know what you should be modifying and what you shouldn't.

> *This is one of the downsides of having a very lightweight and dynamic system. You can create new styles by just providing their name when configuring style options; this means you don't need to explicitly create a style object. At the same time, this does open itself to errors. It's also not possible to find out what styles currently exist or are used. And because style options are really just a front end for element options, and the elements in a style can change at any time, it's not necessarily obvious that options should be restricted to those referred to by current elements alone, which may themselves not all be introspectable.*

Finally, here is the last thing that makes styles and themes so difficult:

> **The elements available, the names of those elements, which options are available or have an effect for each of those elements, and which are used for a particular widget can be different in every theme.**

So? Keep in mind among other things that the default theme for each platform (Windows, macOS, and Linux) are different, (which is a good thing). Some implications of this:

1. If you want to define a new type of widget (or more likely a variation of an existing widget) for your application, you're going to need to do it separately and differently for each theme your application uses (so at least three for a cross-platform application).
2. The elements and options available will differ for each theme/platform, meaning you may have to come up with a quite different customization approach for each theme/platform.
3. The elements, names, and element options available with each theme are not typically documented (outside of reading the theme definition files themselves), but are generally identified via theme introspection (which we'll see soon). Because all themes aren't available on all platforms (e.g. "aqua" will only run on macOS), you'll need ready access to every platform and theme you need to run on.

As an example, here is what the layout of the "TButton" style looks like on the theme used by default on three different platforms, as well as the advertised options for each element (not all of which have an effect):

Mac OS X	*Button.button -sticky nswe -children {Button.padding -sticky nswe -children {Button.label -sticky nswe}}*	
	Button.button	-
	Button.padding	padding, relief, shiftrelief
	Button.label	compound, space, text, font, foreground, underline, width, anchor, justify, wraplength, embossed, image, stipple, background
Windows	*Button.button -sticky nswe -children {Button.focus -sticky nswe -children {Button.padding -sticky nswe -children {Button.label -sticky nswe}}}*	
	Button.button	-
	Button.focus	-
	Button.padding	padding, relief, shiftrelief
	Button.label	compound, space, text, font, foreground, underline, width, anchor, justify, wraplength, embossed, image, stipple, background
Linux	*Button.border -sticky nswe -border 1 -children {Button.focus -sticky nswe -children {Button.padding -sticky nswe -children Button.label -sticky nswe}}}*	
	Button.border	background, borderwidth, relief
	Button.focus	focuscolor, focusthickness
	Button.padding	padding, relief, shiftrelief
	Button.label	compound, space, text, font, foreground, underline, width, anchor, justify, wraplength, embossed, image, stipple, background

The bottom line is that in classic Tk, where you had the ability to modify any of a large set of attributes for an individual widget, you'd be able to do something on one platform and it would sorta kinda work (but probably need tweaking) on others. In themed Tk, the easy option just isn't there, and you're pretty much forced to do it the right way if you want your application to work with multiple themes/platforms. It's more work up front.

Advanced: More on Elements

While that's about as far as we're going to go on styles and themes in this tutorial, for curious users and those who want to delve further into creating new themes, we can provide a few more interesting tidbits about elements.

Because elements are the building blocks of styles and themes, it begs the question of "where do elements come from?" Practically speaking, we can say that elements are normally created in C code, and conform to a particular API that the theming engine understands.

At the very lowest level, elements come from something called an *element factory*. At present, there is a "default" one, which most themes use, and uses Tk drawing routines to create elements. A second allows you to create elements from images and is actually accessible at the script level using the "`ttk::style element create`" method (from Tcl). Finally, there is a third, Windows-specific engine using the underlying "Visual Styles" platform API.

If a theme uses elements created via a platform's native widgets, the calls to use those native widgets will normally appear within that theme's element specification code. Of course, themes whose elements depend on native widgets or API calls can only run on the platforms that support them.

Themes will then take a set of elements, and use those to assemble the styles that are actually used by the widgets. And given the whole idea of themes is so that several styles can share the same appearance, it's not surprising that different styles share the same elements.

So while the "TButton" style includes a "Button.padding" element, and the "TEntry" style includes a "Entry.padding" element, underneath these padding elements are more than likely one and the same. They may appear differently, but that's because of different configuration options, which as we recall, are stored in the style that uses the element.

It's also probably not surprising to find out that a theme can provide a set of common options which are used as defaults for each style, if the style doesn't specify them otherwise. This means that if pretty much everything in an entire theme has a green background, the theme doesn't need to explicitly say this for each style. This uses a root style named "."; after all, if "Fun.TButton" can inherit from "TButton", why can't "TButton" inherit from "."?

Finally, it's worth having a look at how existing themes are defined, both at the C code level in Tk's C library, but also via the Tk scripts found in Tk's "library/ttk" directory. Search for "`Ttk_RegisterElementSpec`" in Tk's C library to see how elements are specified.

Previous: Tree (tree.html) | Contents (index.html) | Single Page (onepage.html) | Next: Case Study: IDLE Modernization (idle.html)