SpecTcl 1.1 User's Guide

SUN MICROSYSTEMS, INC. THROUGH ITS SUN MICROSYSTEMS LABORATORIES DIVISION (“SUN”) WILL LICENSE THIS SOFTWARE AND THE ACCOMPANYING DOCUMENTATION TO YOU (a “Licensee”) ONLY ON YOUR ACCEPTANCE OF ALL THE TERMS SET FORTH BELOW.

Sun grants Licensee a non-exclusive, royalty-free right to download, install, compile, use, copy and distribute the Software, modify or otherwise create derivative works from the Software (each, a “Modification”) and distribute any Modification in source code and/or binary code form to its customers with a license agreement containing these terms and noting that the Software has been modified. The Software is copyrighted by Sun and other third parties and Licensee shall retain and reproduce all copyright and other notices presently on the Software. As between Sun and Licensee, Sun is the sole owner of all rights in and to the Software other than the limited rights granted to Licensee herein; Licensee will own its Modifications, expressly subject to Sun’s continuing ownership of the Software. Licensee will, at its expense, defend and indemnify Sun and its licensors from and against any third party claims, including costs and reasonable attorneys’ fees, and be wholly responsible for any liabilities arising out of or related to Licensee’s development, use or distribution of the Software or Modifications. Any distribution of the Software and Modifications must comply with all applicable United States export control laws.

THE SOFTWARE IS BEING PROVIDED TO LICENSEE “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED. IN NO EVENT WILL SUN BE LIABLE HEREUNDER FOR ANY DIRECT DAMAGES OR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES OF ANY KIND.

Please
Recycle

Contents

Figures

Preface

The SpecTcl User's Guide and Reference describes SpecTcl and how to use it to produce cross-platform applications with graphical user interfaces.

Who Should Use This Book

This guide is written for you if you are a Tcl/Tk programer and want to use SpecTcl or are a programmer and want to learn Tcl/Tk and SpecTcl.

Before You Read This Book

Although you do not have to be a professional programmer, many parts of this guide depend on a familiarity and comfort with programming languages and concepts. If you do not already know Tcl/Tk, we assume that you have used other procedural languages, such as C or Pascal, or scripting languages, such as Perl, C shell, Bourne shell, or Korn shell.

How This Book Is Organized

Here is a brief description of the chapters in this book:

Chapter 1, "Introduction to SpecTcl," describes the product briefly.

Chapter 2, "Getting Started with SpecTcl," guides you, step by step, through:

- Design, save, build, test, and execute, using a very small application.
- The widget layout process, using a somewhat larger application.

Chapter 3, "Basics," introduces the tool palette, command tools, the grid, and other facilities that you use each time you use SpecTcl.

Chapter 4, "Managing Layout," explains how to lay out widgets in SpecTcl applications and why this process differs from the layout process in traditional GUI builders.

Chapter 5, "Common Properties of Widgets," provides information about certain properties that are important the layout process in SpecTcl or have some other special significance. (Tcl/Tk documentation covers most properties in greater depth.)

Chapter 6, "Labels, Buttons, and Menus," explains these basic widgets and provides examples to demonstrate their use.

Chapter 7, "Other Widgets," continues with last chapter's coverage of specific widgets, this time with more complex widgets, such as listboxes.

Chapter 8, "Tcl and Tk," takes a minimalist approach to describing Tcl/Tk. Although most SpecTcl users already know Tcl/Tk, if you happen to be new to Tcl/Tk, this might serve as a stop gap. We recommend you acquire something more substantial on the subject; see "Related Books," below.

Chapter 9, "Advanced Topics," provides information that is only slightly more advanced than the material that precedes it.

Related Books

For information about Tcl/Tk we recommend the following books:

Practical Programming in Tcl and Tk
(Second Edition)
by Brent B. Welch
Prentice Hall PTR, 1995
ISBN 0-13-616830-2

Tcl and the Tk Toolkit
John K. Ousterhout
Addison Wesley, 1994
ISBN 0-201-63337-X

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	**Meaning**	**Example**
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. system% You have mail.
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
`Menu=>Cmd`	Select the Cmd command from the Menu menu	`Edit=>Copy` means to select the `Copy` command from the `Edit` menu.
Code samples are included in boxes and may display the following:		
`%`	UNIX C shell prompt	`system%`
`$`	UNIX Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`

Acknowledgments

This guide is written at the SunScript group of SunLabs, a division of Sun Microsystems. SunScript is directed by John Ousterhout.

All members of the SpecTcl project contributed to this guide; specifically:

- Ray Johnson, manager
- Ioi Lam, developer
- Bryan Surles, developer
- Allan Pratt, writer

My thanks to the project for contributing written material, review comments, and suggestions. Thanks also to Ken Corey, one of the original SpecTcl developers, who spent time getting me up to speed on SpecTcl and to Steve Uhler, the developer originally responsible for the design of SpecTcl.

A special thanks to Shlomtzi Shaham for testing the widget-layout tutorial.

— Allan Pratt

Introduction to SpecTcl

1

SpecTcl provides a development environment to build applications with graphical user interfaces that run on multiple platforms.

The figure, below, shows a simple application in execution (foreground) and the design environment in which it was developed (background).

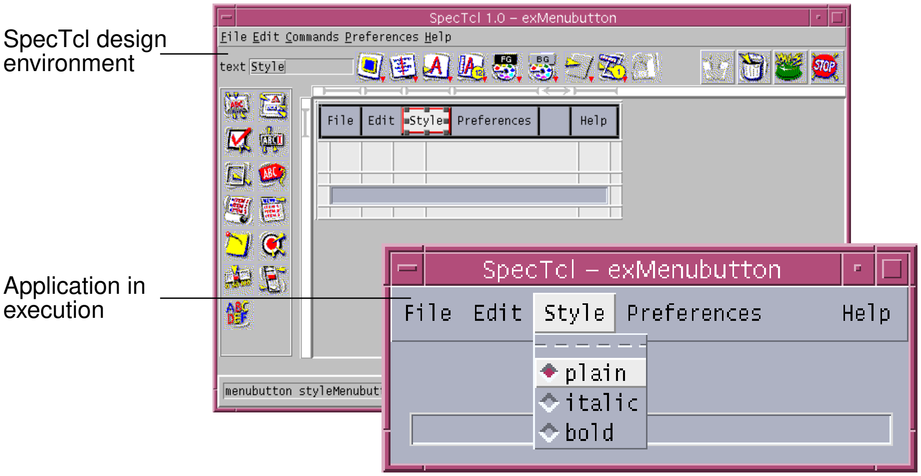


Figure 1-1 A SpecTcl Application - Design and Execution

SpecTcl Features

These are some of SpecTcl's important features:

- Lets you design graphical user interfaces interactively and graphically.
- Creates applications on one platform—Unix, Windows, or Macintosh—that can run on any of the other platforms with no changes to the application.
- Uses a geometry "smart" enough to keep elements aligned across all platforms.
- Enables fast development of applications that require many more lines of code in traditional procedural languages.
- Provides more flexibility at run time than many languages, which simplifies many tasks; for example, generating menus at run time.
- Lets you integrate Tcl/Tk scripts with scripts generated in SpecTcl.
- Lets you alternate quickly between design and execution, without waiting for long compilations; you can add something new and get immediate feedback on how it works.
- Produces executable files that use Tcl/Tk; your users don't require SpecTcl.
- Lets you develop and test simple applications separately, and combine them later into subassemblies of a larger application.

A Technical Note on SpecTcl

If you are interested in the technical issues addressed by SpecTcl's designers, this section is written for you; otherwise, you can skip the section without missing anything you will need to develop SpecTcl applications.

SpecTcl and its Grid Geometry Manager

SpecTcl uses a grid geometry manager that can be described as constraint based. Some of this is hidden from you, as a programmer, because its generally more convenient to work with abstractions such as rows and columns and their widgets, than to work directly with constraints. Some constraints have been mapped onto other entities; for example, the widget sticky property is an abstraction that ties the way a widget is sized to the way its column is sized.

Rather than specifying the size and location of a widget by itself, we constrain the widget with respect to other widgets. The notion of a grid, the sticky property of widgets, and the attributes of columns and rows are just high-level abstractions of low-level mathematical constraints.

For further information on SpecTcl's grid and a general explanation of the widget-layout process, see Chapter 4, "Managing Layout."

SpecTcl Versus Other Constraint-Based Builders

Constraint-based GUI builders are not new, but many are overly general and allow the programmer to specify constraints that are either ambiguous or unsolvable. By using a manageable subset of constraint-based concepts, we have designed SpecTcl as a more reliable programming tool.

To demonstrate the potential for ambiguity in specifying constraints, let's look at an example. Suppose we want to constrain the width of window A to be equal to the sum of the widths of windows B and C. Or, we can write the constraint as (A = B + C). If we make B bigger, we know that the system must either make A bigger or C smaller to satisfy the constraint. The ambiguity is that we don't know which.

It's very easy to specify either an ambiguous or unsolvable constraint in a general constraint-based system.

The grid geometry manager avoids these problems by allowing only a limited number of constraints all of which we know how to solve in an efficient manner. The result is a mechanism with most of the power of a general constraint-based system but with none of its pitfalls.

Getting Started with SpecTcl

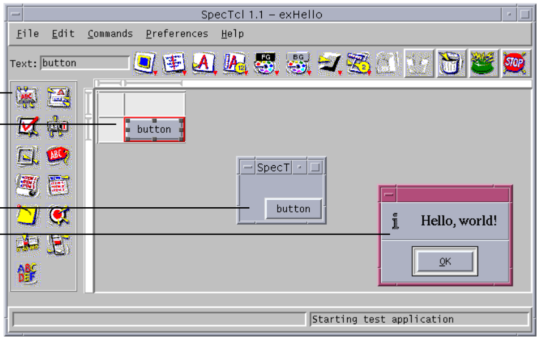
2

This chapter presents the following material to get you started using SpecTcl:

- To introduce you to SpecTcl's tools and the phases of the development process, the "Hello, world" tutorial builds a very basic application.
- To introduce you to the example applications, provided with your release, we show and describe one of the example applications.
- To demonstrate the widget layout process, a second tutorial builds a still simple, but more typical application.

The "Hello, world" Tutorial

Let's begin the first tutorial with a picture of the design and execution environments of "Hello, world."



Hello, world!

Starting test application

At design time:
Drag this palette button
to here in the grid:

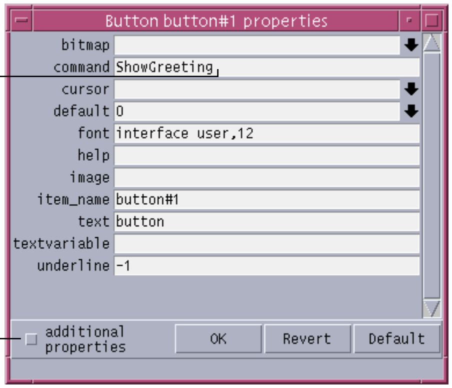
At execution time:
when you press this button,
this dialog box appears

Figure 2-1 "Hello, world" Design and Execution Environment

Starting a New Applica

To create the button,
drag this widget
to this grid cell

To open the button's property sheet,
double-click on the button.



bitmap	
command	ShowGreeting
cursor	
default	0
font	interface user,12
help	
image	
item_name	button#1
text	button
textvariable	
underline	-1

Edit command property here
and press Return.

To see more properties,
click here.

Double-clicking on *any* widget opens its property sheet, so you can view and edit its properties.

Note – To see or edit a property value that's wider than its property-sheet entry, use the left and right arrow keys to scroll left and right.

---

3. To edit this button's command property: a) click in the command entry in the property sheet, b) enter the command `ShowGreeting`, and c) press the `Return` key.

   Pressing `Return` makes the new (or changed) entry part of the interface. `ShowGreeting` is a user-defined Tcl command, described in the next section.

Editing Code

The `Edit=>Edit Code` command provides a very simple ASCII editor that lets you enter a script.

```
proc ShowGreeting {} {
   tk_messageBox -message "Hello, world!" -type ok
}
```

See text for definitions of the Dismiss, Apply, and Revert buttons.

Figure 2-4 Edit Code Window

Use the edit-code editor to:

- Enter proc definitions—that is, user-defined Tcl commands that you can invoke throughout your application.

- Enter code, outside any proc, that you want executed once—after the interface is loaded but before the user works with the interface.
- Source-in Tcl statements from another file; for example:

  ```
  uplevel #0 "source foo.tcl"
  ```

  Here's why you need the `uplevel` statement. Code that you enter in the Edit Code window that is not in any proc is, nonetheless, placed in a `proc` by SpecTcl: a `proc` that is called when the application is started. The `uplevel` statement ensures that your statements are evaluated at the outermost (global) level.

To demonstrate the edit code feature, let's continue with the "Hello, world!" example:

1. Select `Edit=>Edit Code`.

   At first, the edit code window comes up empty.

2. To define the Tcl `ShowGreeting` command mentioned in the last section, enter these commands in the edit-code window:

   ```
   proc ShowGreeting {} {
       tk_messageBox -message “Hello, world!” -type ok
   }
   ```

   The `tk_messageBox` command is a built-in Tk command that posts its message in a dialog box.

3. Click on the Dismiss button.

The buttons along the bottom of the edit-code window do the following:

- The Dismiss button confirms your edits and closes the editor window.
- The Apply button confirms the edits you have made so far, but leaves the window open for more changes.
- The Revert button "undoes" any changes (even confirmed ones), and returns the interface to its state when the edit-code window opened.

Saving the Application

When you're done with additions and changes, save your application. To demonstrate, let's save the "Hello, World" application as `hello.ui`:

1. Select `File=>Save As`.

2. Enter `hello.ui` in the dialog box and press Save.

   SpecTcl always saves the application in a file with a `.ui` (user interface) suffix. If you omit the `.ui`, SpecTcl appends it.

SpecTcl saves your application in the `.ui` file in a form it can read and update when you next select `File=>Open`... .

Use `File=>Save As`... when you need to specify a new file name. Use `File=>Save` if you are saving successive changes to the same file.

Quitting SpecTcl

When you are finished with SpecTcl, select `File=>Quit`. If you have not saved your changes, SpecTcl prompts you to see whether you want to do so, so they are not lost when SpecTcl terminates. Then SpecTcl stops executing.

Building an Executable

When you select `Commands=>Build`, SpecTcl creates an executable file—a Tcl version of your `.ui` file:

file-name`.ui.tcl`

Note – SpecTcl cannot read or reprocess the `.ui.tcl` file. So, any changes that you make to the `.ui.tcl` file are lost the next time you do a build.

For further information on application execution, see “Running the Application Stand-Alone” on page 28.

For the Macintosh only there is an additional Build command. We suggest:

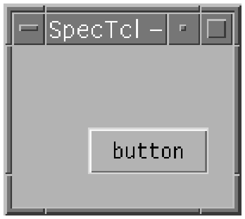
- Use `Commands=>Build` while you are developing an application.
- Use `Commands=>Build Application`... when you are ready to release your application; see “Building a Macintosh Application” on page 115.

Building and Testing

To combine the build and execute phases, use the `Build and Run Test` command—a convenient way to alternate between developing your application and trying it out. (Save your application before the build.)

To demonstrate the command, let’s continue with the “Hello, world!” example:

1. Select `Commands=>Build and Test`



Your application’s main window appears.

2. Press the button in your application.

The “Hello, world!” dialog box appears:

When a button is pressed, SpecTcl executes the Tcl commands in the button’s command property.

3. To stop your application, select `Commands=>Stop Test`.

This also returns SpecTcl to design mode.

Note – To see the same application centered, with a three-dimensional button, select `File=>Open...` of `exHello.ui` in the `examples` directory. See “The Example Applications” on page 30.

Running the Application Stand-Alone

Whether you use UNIX, Windows, or the Macintosh, you can build a `ui.tcl` file that is executable, but there are some differences between the different platforms:

- In MS Windows, the application's `.ui.tcl` file is executable because every Tcl file is registered with Windows as executable. You can double-click on the icon for a `ui.tcl` file to execute the application.
- In Unix, the `.ui.tcl` file is executable and `wish` is executed with your application as a script. So you can also double-click on a `ui.tcl` file to execute the application. For further information and options, see "Execution Options in UNIX" on page 116.
- With the Macintosh, you can choose either of these alternatives:
  - Select `Commands=>Build`, which builds a `ui.tcl` file, but you cannot execute it by double-clicking.
  - Select `Commands=>Build Application...`, which creates a double-clickable `.ui.tcl` file. This the best way to distribute an application that you are ready to release. For further information, see "Building a Macintosh Application" on page 115.

Inserting Debugging Information

You can often debug a Tcl script by adding `puts` commands strategically, writing out variables as they change. In all platforms, the puts command writes a string to output. In UNIX, the string is written to standard output; in Windows or the Macintosh, a console window appears, which displays the string output.

To demonstrate, let's modify the "Hello, World" application, described earlier:

1. To display the property sheet, double-click on the button.
2. Edit the command property, replacing it with:

```
puts "Hello, World, from %W"
```

Note – The `%W` in the `puts` string, above, is a directive to SpecTcl. Within a widget's command property, SpecTcl replaces `%W` with the widget's name. For further information, see "Substitutions in Commands" on page 114.

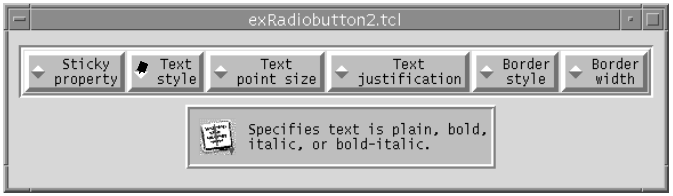
---

3. Select `File=>Save`.
4. Select `Commands=>Build and Test`.

When the application appears, press the button as before. Instead of a dialog box, you see the output: `Hello, World from .button#1`.

The Example Applications

A name such as *appName*`.ui` in this guide refers to an application in the **`examples`** directory, provided in your release materials. Here is an example application that shows you something unique about SpecTcl widgets.



Specifies text is plain, bold, italic, or bold-italic.

Figure 2-5 Executing `exRadiobutton2.ui` - Text Style Selected

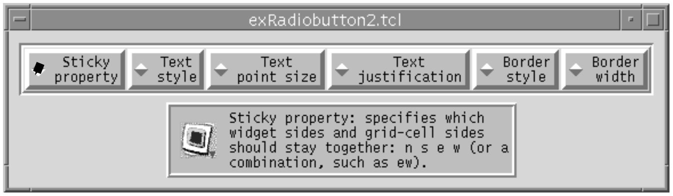
This particular example shows how the label automatically becomes taller to accommodate larger amounts of text. This is an integral part of the window geometry used by SpecTcl, as explained in Chapter 4, "Managing Layout."

We recommend you try this now to acquaint yourself with these examples:

1. Select `File=>Open`... and, in the dialog box, enter `examples` for the directory and `exRadiobutton2.ui` for the file name.
2. Select `Commands=>Build` and `Test`.
3. Click on the Text Style button.

   The application should now look like Figure 2-5.
4. Now, click on the Sticky property button.

   The application should now look like Figure 2-6:



Sticky property: specifies which widget sides and grid-cell sides should stay together: n s e w (or a combination, such as ew).

Figure 2-6 Executing `exRadiobutton2.ui` - Sticky Selected

You can try various things at execute-time and then examine the properties of widgets. Then, you can look at the code by selecting `Edit=>Edit Code`.

The Layout Tutorial

This section demonstrates the widget-layout process in SpecTcl, by guiding you through the creation of an example application step by step. The figure below shows the completed application in execution:

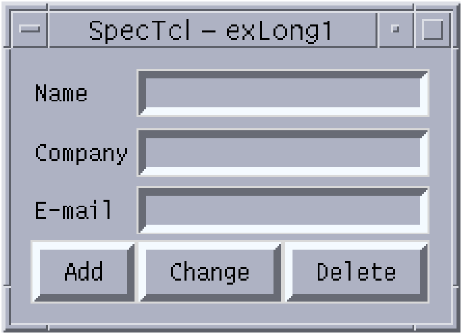
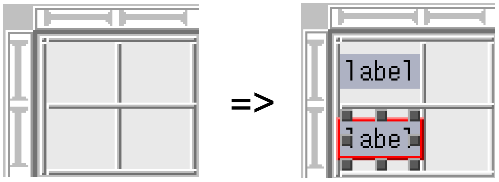


Figure 2-7 The Layout Example in Execution

In this section, each subsection shows a few steps in the design process for the `exLong.ui` application. Each subsection begins with a figure. The left and right parts of the figure show how the application looks at the beginning and the end of the section, respectively.

Adding Labels to an Empty Grid



When you start SpecTcl or select `File=>New`, you begin with an empty grid. To add two label widgets:

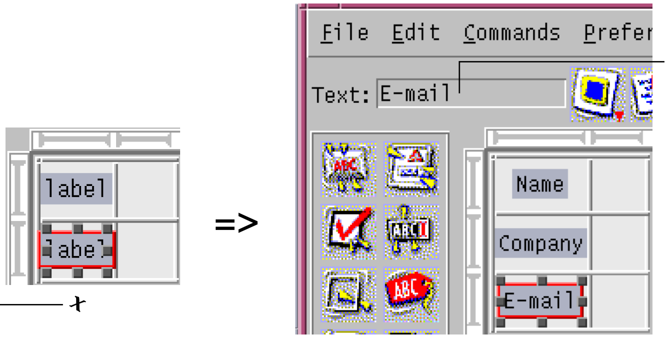
1. Click on the palette's label widget:

   When you click on it (rather than dragging it), the palette widget stays selected so you can easily create several labels. For further information, see "Clicking or Dragging on Palette Widgets" on page 41.

2. To create two labels, as shown above, click on each grid cell of the first column.

   Each time you click in a cell, SpecTcl creates a label widget in that cell.

Completing the Labels



Text: E-mail

To create another label, and a new row to contain it, click about here.

Edit the text property in this text area.

Make sure the palette widget is still selected. Let’s continue by finishing the labels:

1. To create the last label, click below the second one, as shown in the figure, above, left.

   If a palette widget is selected and you click below the grid, SpecTcl makes room for the widget by creating a new row. Then, it places the new label in the new row.

2. To deselect the palette widget, click on it once again.

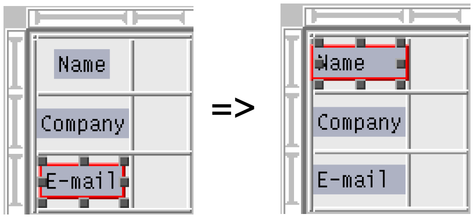
   Otherwise, you’ll continue to create label widgets.

3. To change the labels to read `Name`, `Company`, and `E-mail`, as shown in the figure, click on the first label, to select it, and then edit the text area, as shown in the figure.

   At first, the text area contains `label`, the default text property of the selected widget.

4. Similarly, select the second and third labels and edit their text properties to read `Company` and `E-mail`, respectively.

Improving the Labels’ Appearance



To align the labels:

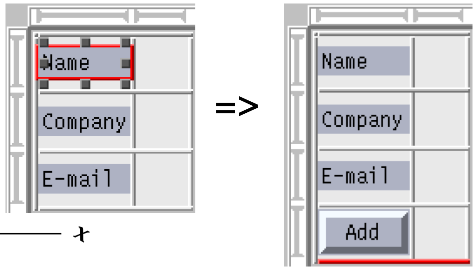
1. Double-click on the Name label, to bring up its property sheet.
2. In the property sheet, edit the Sticky entry to be **ew** (that is, East West).

   This constrains the label to be (and stay) the width of its column.
3. Edit the anchor entry to be **w**.

   This position the text to the left within the label widget.
4. Make the same changes to properties of the `Company` and E-`mail` labels.

Note – There is also a justify property, but justify aligns multiple lines of text; it doesn’t affect the placement of text within a widget.

Creating the Add Button



Drag a palette button
to about here

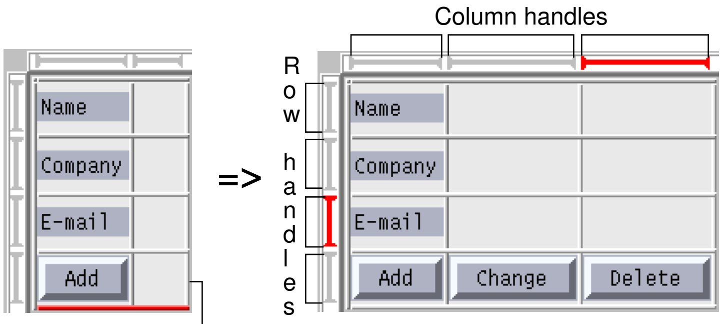
To create the Add button:

1. Drag a palette button widget, , to a place below the E-mail label.

   This creates a new row and a new button—the way it did for the third label.

2. To open the button's property sheet, double click on the new button.
3. In the property sheet, edit these properties (to values as specified in parentheses): anchor (`w`), borderwidth (`4`), sticky (ew), and text (`Add`).

Creating Change and Delete Buttons and Another Column



Click on this gridline; then, to create another column, double-click on it.

To add a new column and the `Change` and `Delete` buttons:

1. Click on the rightmost gridline, as shown in the figure, and then double-click on it.

   The first click selects the gridline; double-clicking creates another column.

2. To copy the `Add` button, first click on it, then select `Edit=>Copy`.
3. To create the `Change` button, first click on the grid cell to the right of the Add button, then select `Edit=>Paste`.

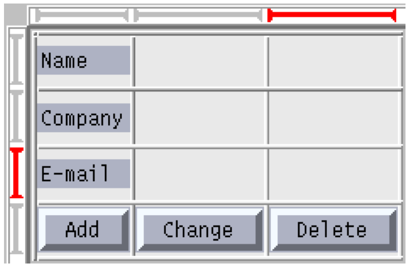
   Clicking on the grid cell selects it, to prepare for the paste operation. You can tell which cell is selected by noting which column and row handles are highlighted. Note the highlighted handles in the figure above, which show that a paste would insert a widget in row 3, column 3.

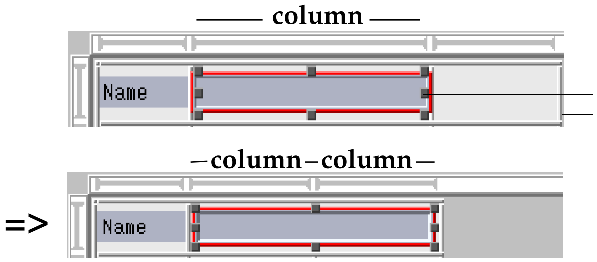
4. Similarly, click in the grid cell that receives the `Delete` button and select `Edit=>Paste`.

   The paste operation gives you a button widget that has all the properties of the Add button, except for a generated item_name.

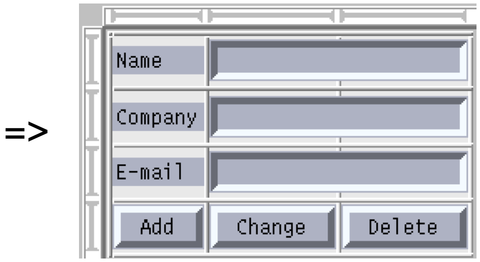
5. To change the text on the two new buttons, select each button, in turn, and edit the text area.

Creating the Entry Widgets





Drag this
to here.



To add the entry widgets:

1. Drag an entry widget, , to the cell beside the `Name` label.

   At this point, the entry widget occupies one column.

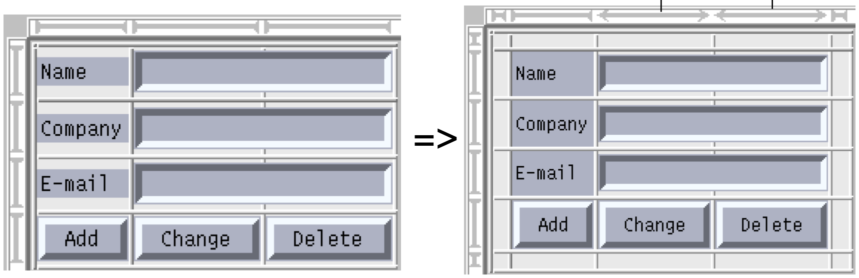
2. To resize the Name entry widget, drag its middle resize handle to the right. To locate the exact place to drag, move the cursor over the entry widget and watch the cursor change from the text cursor, `I`, to a right-side cursor, →▌. When you see the right-side cursor, start to drag.

   Drag the resize handle to the right until the entry widget occupies two columns. See the middle portion of the figure above.

3. Do the same thing to create `Company` and `E-mail` entry widgets; that is, create them and change them to span two columns.
4. To edit properties of the entry widget, double-click on the widget.

   The property sheet appears.
5. In the property sheet, edit these properties to these values (specified in parentheses): sticky (`ew`) and borderwidth (`4`).

Adding Finishing Touches



Arrowheads on column handles indicate automatic resizeability

To add the finishing touches:

1. To place a border around the application, add a row or column, as appropriate, by clicking and double-clicking on gridlines on the border.
2. To set the minimum size for the border, drag the right gridline of the last column, and watch the message area at the bottom of the window.

   The message area displays the size of rows and columns as you move a gridline. The border of `exLong.ui` in the examples directory is 10. You can continue this process for each row or column that's part of the border.

   Note that there's a trick to resizing some of these rows and columns. For a column, always move the *right* gridline. For a row, always move the *bottom* gridline.
3. For columns 2 and 3, click repeatedly on the column handles, until they show as arrowheads, as shown in the figure, above, right. Note that column 1 does *not* have arrowheads.

At execution time, when the user resizes the application window, columns or rows that have arrowheads are automatically resized. For further information, see “Setting Resizeability of Rows and Columns” on page 61.

Examining Run-Time Actions and Resizeability

The version of `exLong.ui` in the Examples directory has a script. To see it, open the application and then select `Edit=>Edit Code`. We suggest you run it.

When you press a button, a procedure does a `puts` identifying the button; when you press return; a procedures does a `puts` identifying the entry and giving it contents.

For a complete explanation of the interaction between the script and the entry widgets, see “The Entry Widget” on page 85.

The figure shows two views of the `exLong.ui` application in execution, before and after the user resizes the application window.

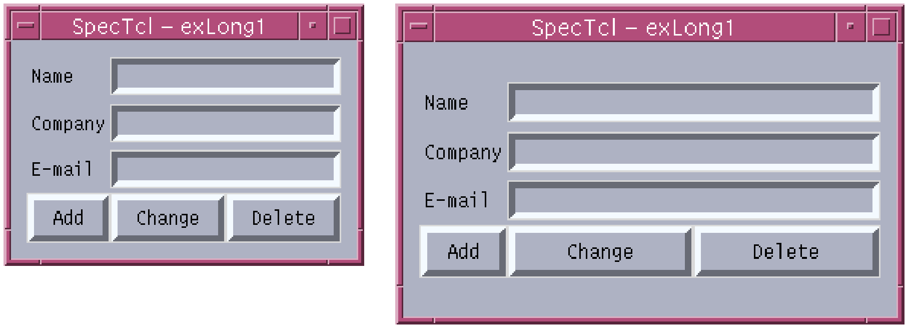


Figure 2-8 Resizing the Application Window During Execution

Note that although the window on the right is definitely larger, some elements have changed size and some have not.

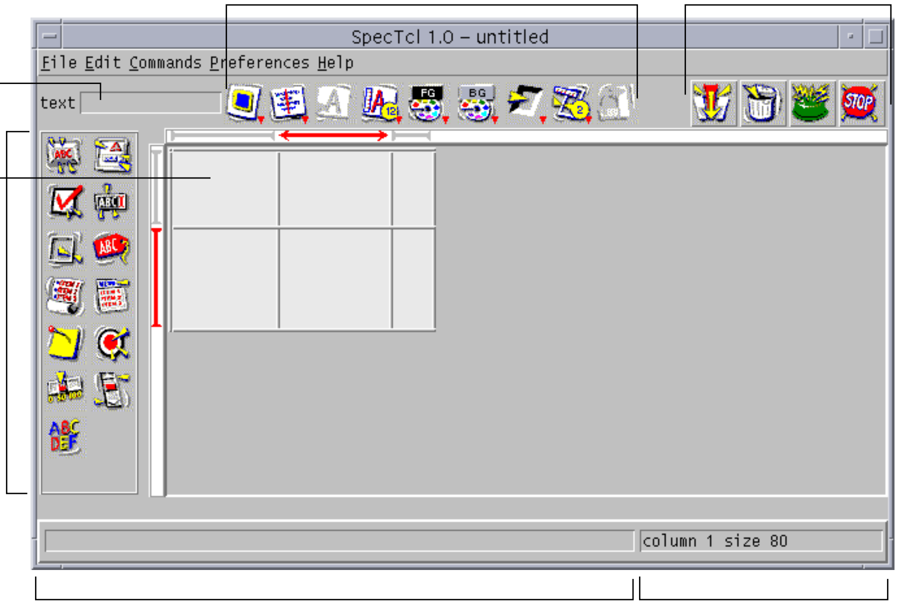
Compare the two pictures with the design goals:

- To provide the entry widgets with more horizontal space.
- To keep the labels the same size.
- To keep both the left and right sides of the row of buttons aligned with the widgets above them.

Basics

3

This chapter describes basic features of SpecTcl. To understand the design window and its tools, consider this pictorial overview:



column 1 size 80

Toolbar: Click on these tools to change property of selected widget.

Click to insert/delete item and start/stop test run.

Text area

Main grid

Palette:
To create widgets, drag any icon to the main grid.

Help area: identifies items as you move the cursor.

Message area: provides feedback.

Figure 3-1 Overview of SpecTcl's Main Window

The Big Picture

Some people prefer to gain familiarity with basic tools and features first and then go on to concepts; others prefer to start with a conceptual overview. To start with the overview, skip to Chapter 4, “Managing Layout,” which explains why the layout process in SpecTcl may be different than what you’re used to.

About Help

SpecTcl provides a help facility and some contextual help.

In addition, there is a Tcl/Tk HTML help facility you can view with your network browser at:

```
http://sunscript.sun.com/man/tcl8.0/contents.html
```

This contains the Tcl/Tk Manual, including Tcl and Tk commands and keywords.

Help Facility

For on-line Help in SpecTcl, select Help on the Help menu; then click on one of the following entries when the table of contents appears:

- Quick Tips
- Glossary of Terms
- Widget options
- Interfacing the user interface with an application
- Known Problems
- Tour of the SpecTcl user interface
- SpecTcl Tutorial
- Miscellaneous
- Changes since the last release

If you are an experienced SpecTcl user, you will find helpful reminders; if you are new to SpecTcl there is also introductory information.

Help Area

As you move the mouse within the design window, the help area at the bottom, left of the window (see Figure 3-1 on page 39) provides help for the item beneath the cursor. For example, “Select the point size for the font” appears when the cursor is over the font tool. Help is available for these item categories: the palette icons, tools in the toolbar, command tools, gridlines, row and column handles, and widgets in the grid.

Message Area

At the bottom, right of the SpecTcl window is the message area, which provides feedback on your interaction with SpecTcl; see Figure 3-1 on page 39. Information provided by the message area include:

- The name of the widget and its grid position after widget creation.
- The height or width of a row or column, respectively, as you move a gridline.

Widget Basics

This section describes creating and selecting widgets.

Creating Widgets

You can create widgets in the current grid or extend the grid by clicking outside it.

Clicking or Dragging on Palette Widgets

To create a widget, do either:

- Drag a widget from the palette to the a particular grid cell.

  When drag the palette widget, the palette widget becomes unselected as soon as the new widget is created.

- Click on a palette widget, then click in one or more grid cells.

  When you click, rather than drag, the palette widget stays selected; each time you click on the grid it creates a new widget. To turn this off, click again on the palette widget.

Creating Widgets Outside the Current Grid

If you drag a widget from the palette to an area to the right of the grid, SpecTcl creates a new column and places the new widget in that column. Similarly, if you drag the palette widget and drop it below the grid, SpecTcl creates a new row and places the new widget in that row.

This works similarly for the other way of creating widgets: clicking on a palette widget and then clicking to the right of, or below, the grid.

Selecting a Widget

To work with a widget, it must be selected. To select it, click on it.

Figure 3-2 A Selected Widget

In the figure, the checkbutton on the left is selected, which you can tell because the handles appear. Note the other, unselected checkbutton.

Selecting a widget shows:

- The grid area that the widget currently occupies. Note the handles in the figure; they mark the periphery of the occupied area.
- Which widget you are currently working with.

  When you create or paste a widget, it is automatically selected.

Here are examples of what you can do with a selected widget:

- Use widget menu commands, such as `Edit=>Widget Properties`.
- Edit its text property in the text area.
- Change widget properties with a toolbar tool.

Note – The notation `Edit=>Widget Properties` means "Select `Widget Properties` from the Edit menu."

Navigating and Selection

The following commands let you move the selection from one widget to another:

- `Commands=>Navigate=>Next Widget`
- `Commands=>Navigate=>Previous Widget`
- `Commands=>Navigate=>Select Parent`
- `Commands=>Navigate=>Select 1st Child`

You can use the first two commands (`Next` and `Previous Widget`) to move systematically through a series of buttons in a frame so as to repeat the action of a command tool. Or you could move through all the widgets contained directly in the main grid.

You can use the last two commands (`select 1st child` and `select parent`) to move the selection from a widget in a subgrid (the frame) to the frame itself (the parent) and back again (to the first child of the frame).

For further information on frames and their widgets, see “Selecting a Widget's Parent or Child” on page 97.

Copying, Cutting, Pasting, and Deleting Widgets

Both `Copy` (that is, `Edit=>Copy`) and `Cut` place a widget on the clipboard, so it can be subsequently pasted. `Delete` just discards the selected widget. These commands are all on the Edit menu.

To copy or cut and then paste a widget, use steps such as these:

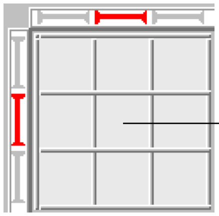
1. Select the widget to be copied or cut by clicking on it.

2. Choose `Edit=>Copy` or `Edit=>Cut`.

3. Click on the cell to receive the widget, which selects the cell.

4. To paste it into the selected cell, choose `Edit=>Paste`.

This figure shows two row and column handles are highlighted, which indicates the selected grid cell:



Selected grid cell, at intersection of highlighted handles

Figure 3-3 A Selected Grid Cell

To delete a widget, first select the widget. Then select `Edit=>Delete`, or press the Delete key if you have one.

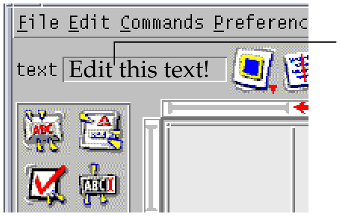
Editing Widget Properties

This section details the various ways to edit (or set) widget properties:

- Editing the text area is a convenient way to set the text property.
- Editing the widget's property sheet lets you modify any property.
- Editing properties with the toolbar is easy with certain properties of the selected widget; see "Editing Properties through Tools on the Toolbar" on page 47.

Editing the Text Area

Buttons and labels typically display text, and this text is considered to be a property (or attribute) of the widget. The text property is quite common, so SpecTcl provides the text area, shown below, as a convenience.



Text area -to edit the text property of the selected widget.

Figure 3-4 A Text Area for Editing the Text Property

To edit a widget's text property easily, a) select the widget and b) edit the text in the text-entry area.

To select the text in the text area, for easier editing, do one of these:

- Double-click on the text in the text area.
- Select `Edit=>Edit Text Property`.

This changes the text property directly, without a Return.

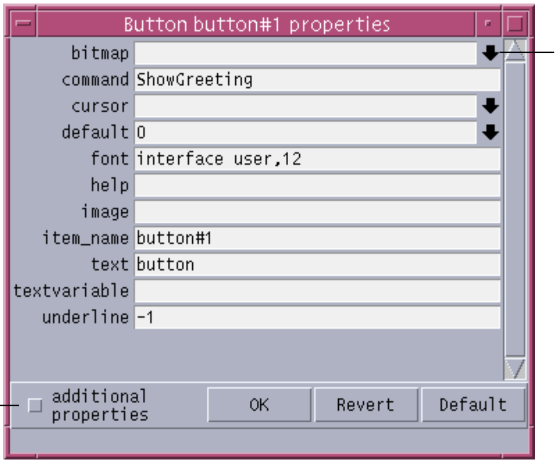
If you do press Return, SpecTcl inserts a newline character in the text property, providing multi-line text.

Editing the Property Sheet

You can view and edit all widget's properties through its property sheet. To display the property sheet do either:

- Double-click on the widget, or
- Click on the widget and select the menu command: `Edit=>Widget Properties` ...

Either action displays the widget's properties; for example:



bitmap:
command: ShowGreeting
cursor:
default: 0
font: interface user,12
help:
image:
item_name: button#1
text: button
textvariable:
underline: -1

Press arrow to see a menu with: `warning`, `question`, and other bitmap values.

Click here to see more properties:

Figure 3-5 A Property Sheet

Editing a Property Entry

To modify an entry in the property sheet:

1. Set the cursor in the entry.
2. Enter or edit an entry.
3. Press the Return key to confirm the new value.

Note the red print as you edit an entry. When you press the Return key, the new property becomes part of the interface and the print returns to black.

Clicking on the OK Button

By clicking on the OK button, you confirm the last property change, and close the dialog box.

Clicking on the Default Button

Resets the properties to the default properties for this class of widget (for example, button or label) and this project. For further information, see "Editing Widget Default Properties" on page 48.

Clicking on the Revert Button

When you click on the Revert button, you reset the property sheet to the values it last loaded—that is, the properties displayed when you 1) opened the property sheet or 2) selected another widget. This enables you to edit several property values and then have an "undo" for those changes.

Editing Properties through Tools on the Toolbar

With each tool on the toolbar, you can set one property of the selected widget.

Sticky; see “Sticky Property” on page 39.

Justifies multi-line text: left, right, or center. Displays current justification; cycles between left (shown here), right, and center.

Font style: plain, bold, italic, or bold, italic.

Font size: 8, 10, 12, ... 36. Current size is displayed in the small circle.

Foreground; displays a panel of colors.

Background; displays a panel of colors.

Relief: plain, raised, sunken, ridge, and groove.

Borderwidth: 0, 1, 2, 4, 8, 12. Current borderwidth is displayed in the circle.

Orientation: toggles scrollbars and sliders between vertical and horizontal.

Tip – To reapply the last action of the toolbar, select `Commands=>Reapply the Toolbar`, or enter Ctrl-r. To apply the same property to multiple widgets, set the first widget’s property with a tool, then repeat the tool action by selecting another widget and reapplying it, as just described. To move quickly through a group of widgets, see “Navigating and Selection” on page 43.

Using and Changing Widget Names

When you create a widget, SpecTcl generates a name for it, using the widget's class name and a serial number; for example, label#1 and radiobutton#2. When you move the mouse, the help area displays the name of each widget as the cursor passes over it.

The widget name is in the property sheet—as the item_name property, where you can view and edit it. Widget names can contain letters, digits, and underscores (_). SpecTcl reserves the pound sign (#) for names it generates.

Names in a script begin with a period. If item_name is label#1, you refer to it as `.label#1` in a script; for example:

```
.label#1 config -background red
```

This naming convention is derived from Tk, but also differs from it, as explained later in this guide.

If your application loads multiple user interfaces (`.ui` files), there are additional widget-name conventions; for further information, see "Using Multiple Assemblies" on page 111 and "Widget Names in SpecTcl Scripts" on page 112 in Chapter 9, "Advanced Topics."

Editing Widget Default Properties

You can set default properties for any palette widget. These default properties override the system default properties that appear when you create a widget; for example, `button` on new buttons, which is a system default text property.

SpecTcl saves the default properties that you set in the application's `.ui` file, which means:

- Default properties are available across multiple SpecTcl sessions.
- Default properties are set on a per-project basis; that is, the defaults you set on one project do *not* apply to another project.

To set default properties, do one of the following:

- Select `Edit=>Default Properties=>`*widget-class;* for example, `Edit=>Default Properties=>button`.
- Double-click on any palette widget.

When the property sheet appears, use it the way you do standard property sheets. Newly set default properties are available for use immediately—when you create the next widget of the given type or when you click on `Default` button in its property sheet.

Grid Basics

This section starts you using the grid—shows how to do work with it without much commentary. For a more systematic presentation of the grid and its geometry, see Chapter 4, “Managing Layout.

In SpecTcl, you always work within a grid structure. When you create a widget, it always go in a particular grid cell. The figure shows the way rows and columns are numbered.

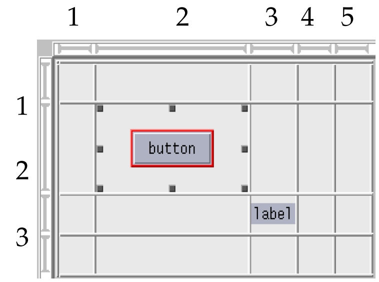


Figure 3-6 Numbering Rows and Columns in the Grid

In each cell of the main grid, you can place at most one widget.

Inserting a Row or Column

To insert a row or column to the grid:

1. Select a gridline by clicking on it.

   The gridline turns red, showing it’s selected.

2. To create the new row or column: click on the insert tool on the toolbar, press the `Insert` key, or select `Edit=>Insert`.

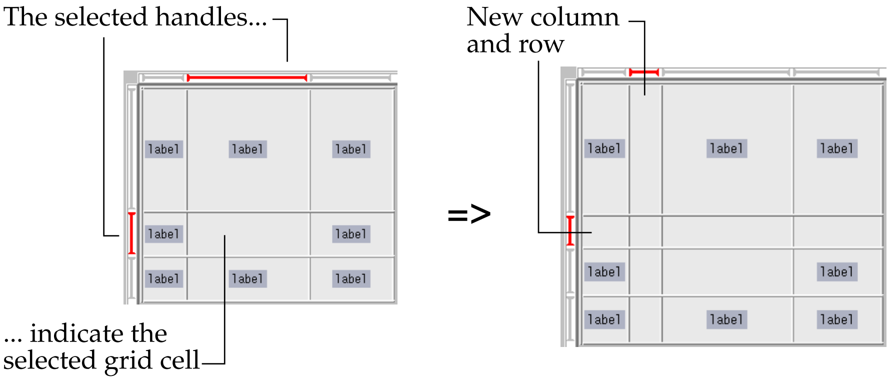
   If you clicked on a column gridline, it adds a column to the right of the selected gridline. If you clicked on a row gridline, it adds the row below the selected gridline.

Inserting a Row and Column

To simultaneously create a new row and column:

1. Select a grid cell, by clicking on an empty grid cell.
2. Click on the insert tool or select `Edit=>Insert`.

In Figure 3-7, the left side shows a selected grid cell—its row and column handles show it's selected. The right side shows the same grid after the insert operation. A new grid cell is selected: at the intersection of the new row and column, to the left and above the previously selected row and column.



The selected handles...

New column and row

... indicate the selected grid cell

Figure 3-7 Inserting a Row and Column

Resizing a Row or Column

To resize a column, drag its right gridline left or right. As you drag the gridline, note that the column size is displayed, as it changes, in the message entry in the lower right of the main window. Similarly, you can resize a row, by dragging its lower gridline.

Resizing a row or column can affect many things, such as the size of the widgets it contains; see Chapter 4, "Managing Layout, for further information.

Deleting a Row or Column

You cannot delete a column or row that contains widgets; therefore, to delete either one:

1. First, move any widgets out of the column or row that you want to delete.

   To move a widget, just drag it to another place in the grid; to delete a widget, select it and then select `Edit=>Delete` (or press the `Delete` key if you have one).

2. Click in any cell within the empty column (or row) you wish to delete.
3. On the toolbar, click on the Delete tool or (press the `Delete` key).

Beyond the Main Grid

Although one grid cell can only accommodate a single widget, the widget can be a container widget or **frame**, which can hold several widgets. For example, to group radiobuttons, place them in a frame.

Frames have rows and columns and many other characteristics of the main grid. In fact, you can think of frames as subgrids. For further information, see "The Frame Widget" on page 95 .

Managing Layout

4

Very likely, you'll find using SpecTcl to build an application to be quite different than what you're used to, because SpecTcl uses a `grid` geometry manager. Geometry managers arrange widgets on the screen, and they definitely affect the way you layout the widgets of your application. This chapter explains the `grid` geometry and provides a conceptual model of the layout process it supports.

WYSIWYG versus Portable

With current platforms—UNIX, Windows, and the Macintosh—you can create a graphical user interface (GUI) builder that is either WYSIWYG or portable across those platforms, but not both. And, we decided portability was, and is, SpecTcl's most important design objective.

Traditional GUI Builders

Traditional GUI builders use a `place` geometry, which is WYSIWYG. When a user positions a widget at design time, the coordinates of the widget are saved and used to position the widget at run-time. This means that the position and size of widgets are set and fixed at design time.

Advantages

Such GUI builders vary, but they typically share these advantages:

- They are easy to learn and to use, because you more or less “draw” the interface the way you draw with a graphical editor.
- They have the easiest possible conceptual model, because there’s no difference between their design-time and run-time appearance.
- They impose few restrictions in how and where to place widgets.

Disadvantages

And, traditional builders typically have these disadvantages:

- Applications that look good on the platform on which they’re built, cannot be executed on another platform unless the interface is realigned. In other words, the interface is not portable.

  Widgets on different platforms are roughly comparable, but the differences are large enough to create an out-of-focus look if you mechanically translate applications from one platform to another. This comes from differences in widget shape, style, the fonts they display, and placement strategies. Of course, some builders don’t support all these platforms with or without realignment.
- An application’s interface is usually static and not very flexible. That is, a font change, transposing two widgets, or a change in border style can force you to realign the widgets of the application interface.
- The interface cannot usually resize itself automatically in response to the user resizing the application window.

SpecTcl

The following subsections look at SpeTcl’s design goals, explain why a constraint-based system was chosen, and describe some of the implications of that choice.

Design Goals

When SpecTcl was designed, its primary design goal was, and is, portability. Since portability was deemed more important and WYSIWYG doesn’t allow for easy portability we decided to sacrifice WYSIWYG for portability.

SpecTcl uses a `grid` geometry manager, described next, to work around the problems found in traditional GUI builders.

A Geometry Based on Constraints

At a conceptual level, the grid geometry manager is constraint-based. Instead of specifying widget positions as fixed screen positions, widgets are located with respect to each other. In SpecTcl's grid, when you say two widgets are in the same column, you are stating a relationship, but are *not* specifying fixed positions. The size and position of that column varies in ways you, as a programmer, can control.

Similarly, widget sizes vary according to what the widget currently displays (the size of the text or image that it displays). Widget size can also vary with the size of its row or column, if the widget's properties make this constraint.

Constraints in Disguise

Here are some aspects of widgets and their grid that are actually constraints on widget size and location:

- Row size - this specifies a minimum vertical distance between two horizontal gridlines and has secondary affects on the widgets placed in the row.
- Column size - this specifies a minimum horizontal distance between the two vertical gridlines that form the column and has secondary affects on widgets placed in that column.
- Sticky property - a sticky property of `ew` ties the widget width to the width of its column, `ns` ties the widget height to the height of its rows, and `nsew` ties widget size to its grid-cell size. The term "ties" signifies a dynamic relationship. If, at run-time, one entity changes, any entities tied to it also change.
- Row resizeability - if a row is set to be resizeable, widgets tied to it, as described above are resized when the row is automatically resized.
- Column resizeability - if a column is set to be resizeable, widgets tied to it, as described above are resized when the column is automatically resized.

Characteristics of Grid-Based Applications

Because of the grid, SpecTcl applications have these characteristics:

- An application that you create on one platform can run on another platform without modification and its interface remains aligned.
- An application's interface is not static. In fact, it adjusts automatically to many changes to maintain its alignment dynamically.

  As a programmer, you can take this one step further—to make your application responsive to real-time changes, a particularly useful feature for a web application. For example, you can let users choose between reading an English or Spanish display, while the interface stays aligned in both cases.
- A well-designed interface can respond appropriately when the user resizes the application window.

  At design time, as a programmer, you can control the way the application allocates additional space to widgets at run-time.

About the Grid

Normally, you might think of a grid as a regular, rigid entity, like the grid in a spreadsheet application. But SpecTcl uses a *smart grid* that adapts itself flexibly to real-time changes. The smart grid is different than a spread sheet in the following ways:

- You can reposition and resize each column and row to fit your situation.
- Columns and rows can change size and position in response to real-time changes.
- You can create subgrids—with the frame widget—and nest them to any level.
- The grid is *not* WYSIWYG; in fact, it is visible only at design time.

  What the grid provides is a conceptual model of the way widgets are related to each other; for example, their alignment.

In short, this is not your average grid; it responds intelligently to many situations as they arise.

More on Dynamic Alignment and Resizing

To begin the conceptual model, this section presents further information on dynamic alignment and control of dynamic resizing.

Dynamic Alignment

You can align widgets, horizontally or vertically along gridlines and have SpecTcl maintain that alignment dynamically. Although widgets and gridlines might change in size and position, they can be constrained to do so in ways that retain their alignment.

Applications that change in real time can especially benefit from dynamic alignment. For example, suppose your application displays stock-market quotes for the top 5 most volatile stocks on a particular market. Then, column headers and values can adjust when a new stock enters the display—to accommodate a new stock and new values, either of which might require a change in column size. Dynamic alignment enables the new column to retain the alignment used in other columns.

Most applications need this feature to adjust across different platforms, which usually display slightly different fonts and require other minor adjustments.

Dynamic Resizeability

With SpecTcl dynamic resizing, users can size the application window to suit their own situation and find that the user interface responds in an intelligent way—expanding (or contracting) some areas when it benefits the user and leaving areas as is when it doesn’t.

As a programmer, you don’t, in general, know your users’ window resources, which might vary substantially from user to user. So, the amount of screen an application uses can only be right for everyone if it changes dynamically, under user-directed program control.

Rows and columns provide a vehicle for you, as a programmer, to express which screen areas get extra space and which don’t. Your decision will be influenced by the type of widgets in that area. For example, a wider entry widget can accommodate more text, but a larger button might just look silly.

The following sections lead you through the layout process, showing the choices you can make and how SpecTcl reacts to them. The layout process is presented in these broad categories:

- Placing widgets in cells of the grid.
- Controlling columns and rows.
- Positioning widgets within their cells.

Placing Widgets in Grid Cells

When you drag a new widget to the user interface, you drag it to a particular row and column of the grid. The area occupied by the widget, is called a **cell**, When you select a widget, the handles delimit its cell, as shown in the figure, below.

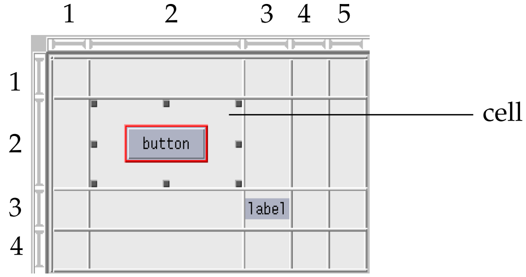


Figure 4-1 Placing Widgets in the Grid

You can place one widget, and only one, in a given cell. A frame (container) widget lets you circumvent this restriction; see “The Frame Widget” on page 95.

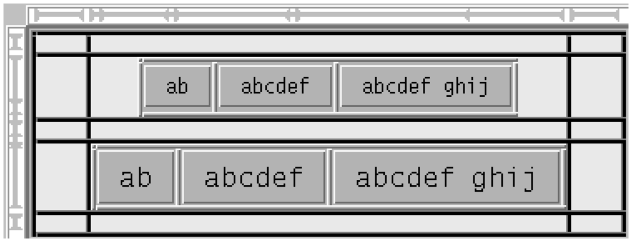
Controlling Widget Size

SpecTcl enables you to specify the size of application widgets in several ways.

Automatic Sizing

When you leave a widget’s height and width zero (the default), widgets are sized automatically to accommodate what they display—text or image. With text, widget size depends on the length and font size of the text—and on padding, as shown in Figure 4-3.

For example, in Figure 4-2, the six buttons (in frame containers), would be the same size if they displayed the same characters and font:



From top to bottom, these buttons differ only in font size.

From left to right, these buttons differ only in text length.

Figure 4-2 Self-Sizing Buttons

As text on labels changes in production applications, the user interface can adjust automatically, so that no text is crowded or truncated. Automatic resizing of widgets is especially convenient if your user interface displays labels in multiple languages.

The Effect of padx and pady On Widget Size

You can pad the size of a button, which is primarily based on the text (or image) that it displays, through its `padx` and `pady` properties, as shown here:

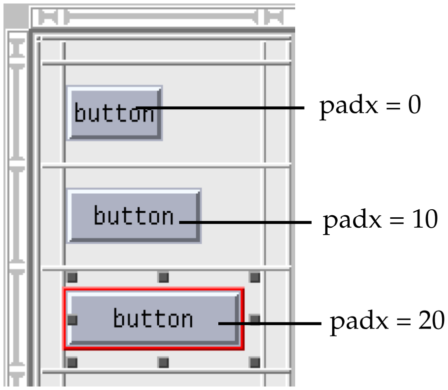
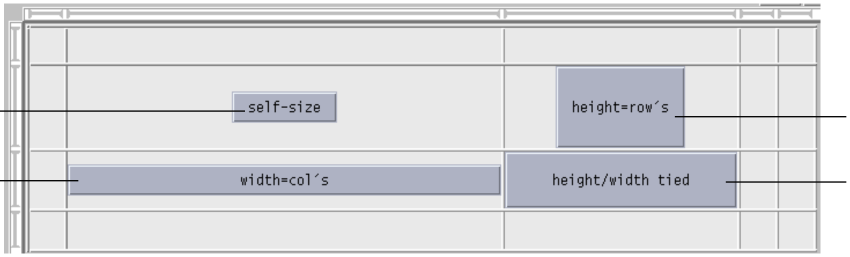


Figure 4-3 The Effect of `padx` and `pady` on Widget Size

To give a better visual comparison, a sticky value of `w` (West) keeps the buttons against the left cell wall.

Tying Widget Size to Cell Size

There is a useful alternative to letting widgets self-size: you can constrain the height or width of a widget to that of its row or column. The figure shows widgets with each of these alternatives:



Self-Sized.

Constrained to column width.

Constrained to row height.

Constrained to row and column sizes.

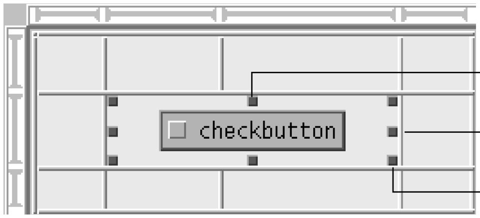
Figure 4-4 Widgets and Various Sizing Constraints

To place such constraints on a widget, set its sticky property, as explained in "Sticky Property" on page 70.

You can then resize the widget by resizing its row or column, as shown later in this chapter.

Changing a Widget's Row or Column Span

You can extend a widget's cell across column and row boundaries. To do so, select the widget and drag a handle, as shown in the figure:



Drag this handle to change the number of rows spanned.

Drag this handle to change the number of columns spanned.

Drag corner handle to change both rows and columns spanned.

Figure 4-5 Changing Widget Row and Column Spans

Setting Specific Sizes

It’s rarely better to set height and width explicitly, but in exceptional situations it might be appropriate. For widgets that display text, set the width property to the number of average-sized characters to be displayed in the specified font. If you display more characters than the explicit width specifies, truncation is likely.

Controlling Rows and Columns

Two important user-interface parameters that you set by column and row are:

- Minimum sizes for columns and rows
- The resizeability of columns and rows

  You can also extend this column and row resizeability to the widgets they contain, on a widget-by-widget basis, as explained later in this chapter.

Establishing Minimum Sizes for Rows and Columns

When you move a column gridline, the widths of the newly positioned columns establish minimum column widths. If the width of a self-sizing widget exceeds the width of its cell, the column expands automatically to accommodate the widget. But the columns won’t automatically contract to less than these minimum widths. Similarly, new row positions establish minimum row heights.

Setting Resizeability of Rows and Columns

As previously mentioned, you set the resizeability of a SpecTcl application on a row-by-row and column-by-column basis. You specify whether the column (or row) size is to vary or stay fixed, when the application window is resized. You can also set the resizeability of rows and columns in the subgrid within a frame (a container widget).

Arrowheads in this figure show the rows and columns that are resizeable:

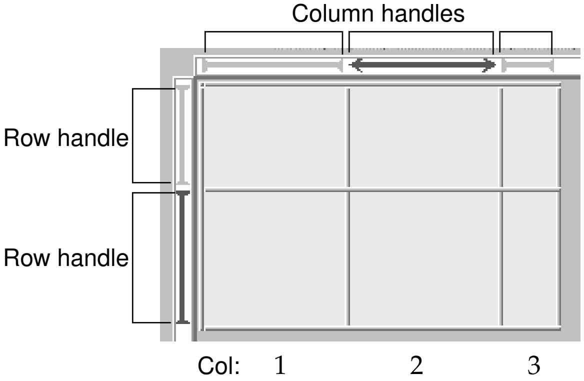


Figure 4-6 Resizeability of Rows and Columns

The grid in the figure above demonstrates how SpecTcl indicates resizeability:

- Column 2 is resizeable, as indicated by *arrowheads* on its column handles.
- Columns 1 and 3 are not resizeable.
- The rows are not resizeable.

To make a column resizeable:

1. Click on the column handle.

   This selects it, turning it red, as shown by the darker handle in column 2 above.

2. Click again on the column handle.

   Each click, after the column handle is selected, toggles the column—resizeable and non-resizeable.

You toggle row resizeability similarly, by clicking repeatedly on the appropriate row handle.

Controlling Widget Resizeability

A widget can get extra horizontal (or vertical) space only if its column (or row) gets extra space. So, a first step for the widget is to set the resizeability of its row or column appropriately. The next step is to set its sticky property appropriately. The widget can get extra horizontal space (with sticky = `ew`), extra vertical space (with sticky = `ns`), or both (with sticky = `nsew`). When the user resizes the application window, widget resizing depends on all the factors just mentioned. For further information on the sticky property, see "Sticky Property" on page 70. The next section demonstrates these issues.

Resizing the Application Window

To demonstrate resizeability in an application, Figure 4-7 shows the `exResize.ui` application, in the `examples` directory, at design time:

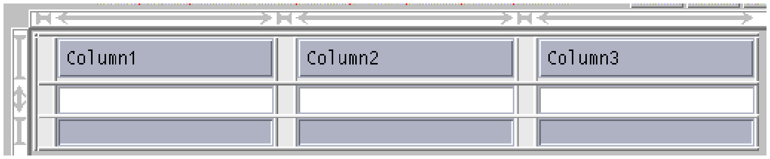


Figure 4-7 Designing `exResize.ui` for Resizeability

Figure 4-8 shows the `exResize.ui` application in execution. On the left, the user has resized the application window to use minimal space; on the right, the user has resized it to use more space.

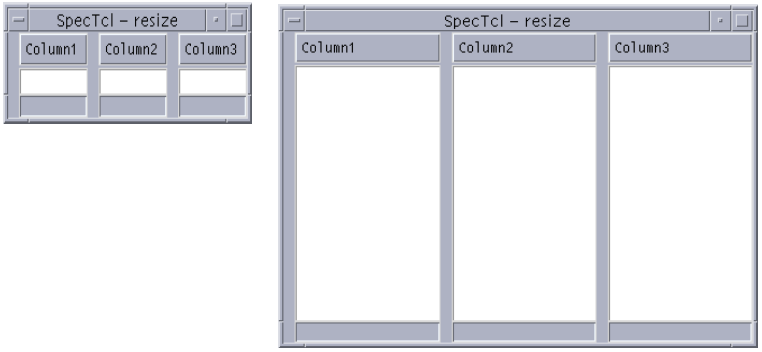


Figure 4-8 Executing `exResize.ui` - Resizing the Application Window

Resizeability Considerations

When you design for resizeability, consider that widgets vary widely in how (and whether) their expansion benefits the user. For example, row 2 in Figure 4-7 is set resizeable because the text widgets can display significantly more text when they have more vertical space.

On the other hand, larger buttons might just look awkward. And, remember that when a button expands, the font size of its text stays the same, unless you change it. To determine what's best, experiment, and note the visual effect.

The three columns (also in Figure 4-7) are set expandable, again because of the text widgets. However, this also keeps column headers (labels) and entries (at the bottom) aligned with the text widgets as they expand.

Consider opening `exResize.ui`, in SpecTcl, and then note the following elements which make it work:

- For the labels, the sticky property has been set explicitly to `ew`, so that they expand horizontally if their grid column expands. For text and entry widgets, the sticky property is set by default to enable expansion.
- Note that the columns (and the center row) have been set resizeable by clicking on column and row handles (as shown in the previous section).

Positioning a Widget within its Cell

This section describes properties that affect widget position in their grid cells.

The wadx and waxy Properties

You can specify values that maintain a minimum distance between widgets and their grid cells through the wadx and wady properties.

Figure 4-9 demonstrates horizontal minimums by showing buttons with wadx values of `0`, `10`, and `20`:

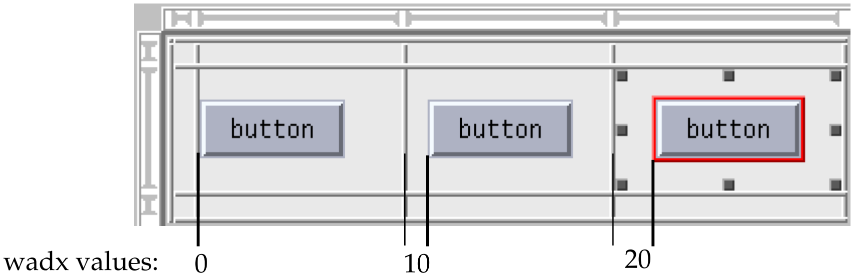


Figure 4-9 The Effect of `wadx` and `wady` on Widget Position

The wady property is similar—specifying the minimum number of pixels between a widget and its grid cell in the vertical direction.

The Sticky Property

Figure 4-10 shows ways the sticky property can position the widget within its grid cell. From the left, the buttons have sticky properties `n` (North West), `s` (South), and `sw` (South East):

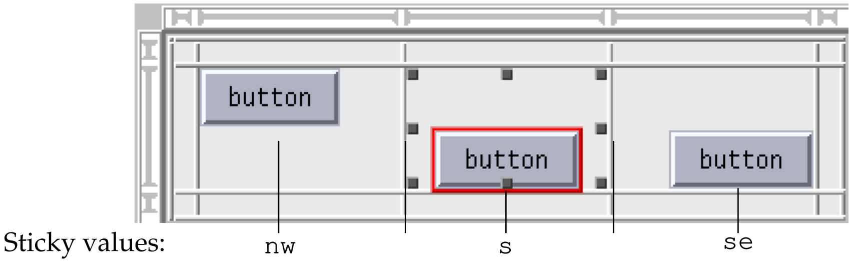


Figure 4-10 The Effect of the Sticky Property on Widget Position

For further information on the sticky property, see “Sticky Property” on page 70.

Aligning Widgets

Suppose you have a column with entries of various widths that you want to align (either left or right). To do so, set the sticky property of each entry—to `w` for left-alignment; to `e` for right-alignment. The entries stay aligned, then, even if individual entries are resized, or if the column is moved or resized.

Similarly, rows of widgets can be aligned by setting their sticky property to `n` or `s`.

If you set the sticky property to `ew`, the widgets will be constrained to be the same width, and both left and right aligned. Similarly, If you set the sticky property to `ns`, the widgets will be constrained to be the same height, which keeps both tops and bottoms of the widgets in alignment.

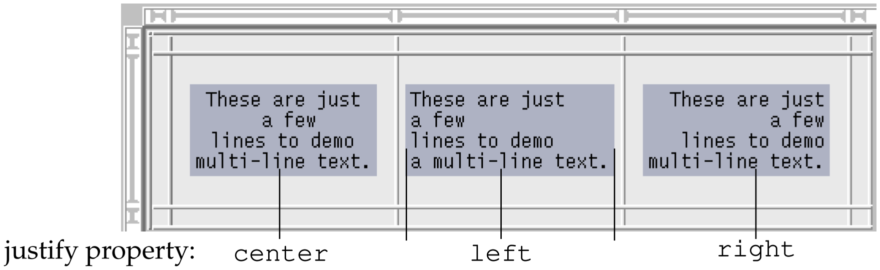
Note that this is alignment with a difference. In SpecTcl, the sticky property is, in effect, a *stay aligned* command.

Aligning Multi-Line Text within a Widget

The justify and anchor properties both affect the way multi-lined text is displayed.

Using the Justify Property

You can align multi-line text in a label (or button) by setting its justify property to `center` (the default), `left`, or `right`, as shown in Figure 4-11:



These are just
a few
lines to demo
multi-line text.

These are just
a few
lines to demo
a multi-line text.

These are just
a few
lines to demo
multi-line text.

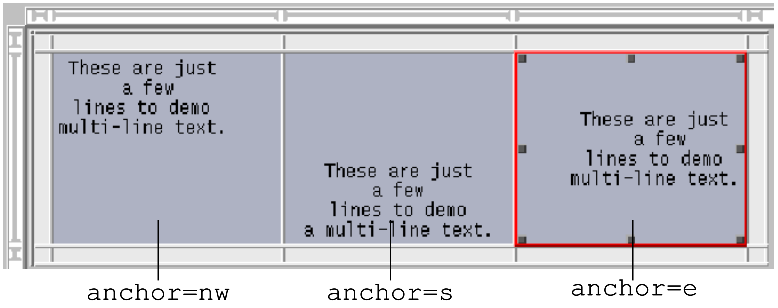
Figure 4-11 The Effect of the Justify Property on Multi-line Text

For further information on the justify property, see "Justify Property" on page 68.

Using the Anchor Property

You can also position the text within the widget if the widget is large enough for this to show. To be more precise, you are positioning a imaginary rectangle that surrounds the text. To do this type of positioning, use the anchor property.

Figure 4-12 shows ways the anchor property can position the "rectangle" that holds the text within a label. From the left, the buttons have anchor properties of `nw` (North West), `s` (South), and `e` (East).



These are just
a few
lines to demo
multi-line text.

These are just
a few
lines to demo
a multi-line text.

These are just
a few
lines to demo
multi-line text.

Figure 4-12 The Effect of the Anchor Property on Text Position

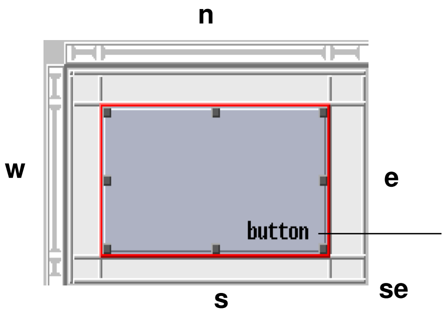
For further information, see "Anchor Property" on page 67.

Common Properties of Widgets 5

This chapter describes certain properties that apply to several widgets. Properties that you don't find here might be described in the section that describes the individual widget. For further information on most properties, see the documentation for Tcl/Tk.

Anchor Property

The anchor property positions text within a button or label; for example, note the position of the word `button` within the outsized button, below.



The anchor property uses compass points to position text or image.

Anchor property: **se**.

Figure 5-1 Positioning Text or Image with the Anchor Property

Anchor can be n, s, e, w (compass points North, South, East, West), c (centered), and intermediate compass points ne (North-East) and so forth. The button in the figure has an anchor property of se for South-East. To left justify text, use **w**; to right justify, use **e**, to center, use **c**.

Borderwidth Property

The borderwidth property specifies the width of the widget's border, in pixels. The figure shows buttons with a borderwidth of 1, 2, and 4, respectively.

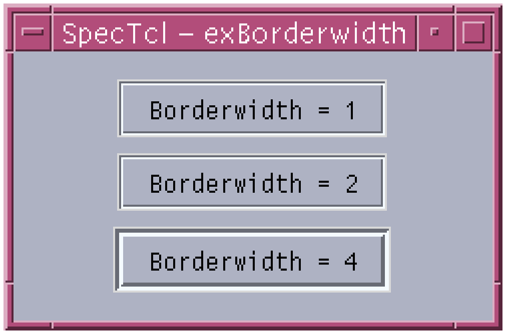
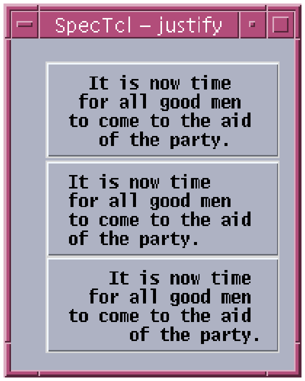


Figure 5-2 Effect of Borderwidth Property

Note – When you use a 3-dimensional border, the display is effective only for a borderwidth of 4 (or more). See also, "Relief Property" on page 69.

Justify Property

The justify property applies only to multi-line text, like the text you see in the buttons, below. Those lines are centered, left justified, or right justified, depending on the justify property: `center`, `left`, or `right`, respectively.



It is now time
for all good men
to come to the aid
of the party.

It is now time
for all good men
to come to the aid
of the party.

It is now time
for all good men
to come to the aid
of the party.

The justify property aligns text lines:

center

left

right

Figure 5-3 Aligning Multi-Line Text with the Justify property

See also, “Anchor Property” on page 67, which positions the block of text within the widget.

Relief Property

All widgets have a relief property that provides alternatives for border style: plain, raised, sunken, ridge and groove. To see the design window for the application below, select `File=>Open` `exRelief.ui`. The application demonstrates the relief property alternatives, as applied to various widgets:

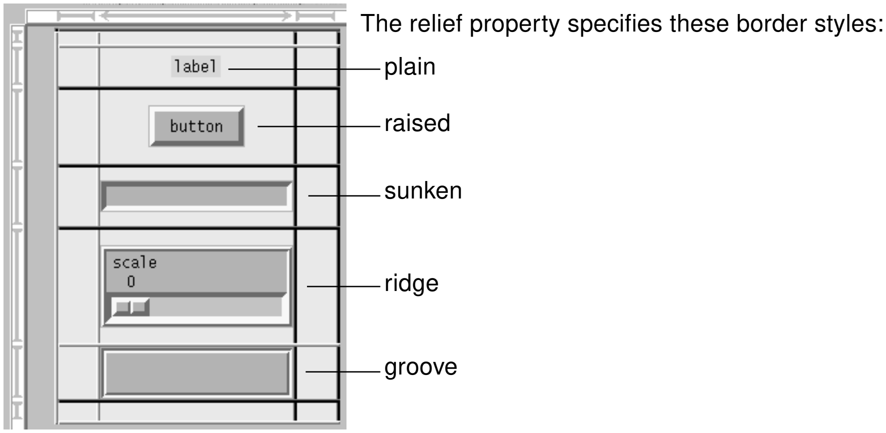


Figure 5-4 Setting Border Style with the Relief Property

To provide the space for a 3-dimensional effect, set the borderwidth property to 4 or higher. To see a figure that shows the difference, see “Borderwidth Property” on page 68.

Sticky Property

If you place a widget in a grid cell larger than itself, the widget is centered in the grid cell, away from the sides, like `label1` in the figure. To examine the design window, below, for the application, select `File=>Open exSticky.ui`.

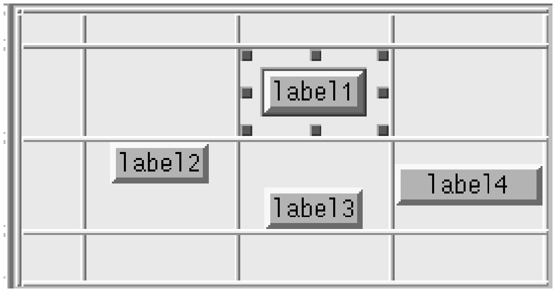


Figure 5-5 Widgets with Different Sticky Properties

The **sticky** property controls this placement, enabling you to "stick" the widget to any of the grid-cell walls, which are described as North, South, East and West, and represented in the property as: `n`, `s`, `e`, and `w`. In the figure, above, label2 is stuck to the top; it has a sticky property of n (North). Label3 and label4 have a sticky property of s and ew, respectively.

Setting the Sticky Property

When you click on the sticky tool, shown below, it displays a panel of selection alternatives, also shown:

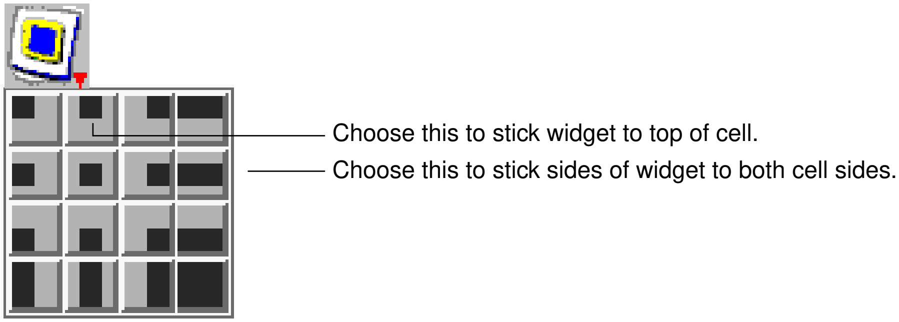


Figure 5-6 Using the Sticky Tool

To use the sticky tool:

1. First, click on the widget you want to change.
2. Click on the sticky tool to display a panel of alternatives.
3. Click on the alternative that shows the way the widget should be positioned in its grid cell.

The second way is to use the property sheet:

1. Double-click on the label you want to set—to bring up the property sheet.
2. Change the sticky property—to **`n`** for label2 and to **`ew`** for label4.

Labels, Buttons, and Menus

6

This chapter presents information that applies to specific widgets—labels, buttons, radiobuttons, checkbuttons, and menubuttons. All references to specific application, such as `exLabel1.ui`, refer to applications in the `examples` directory.

For information common to all widgets, see Chapter 5, "Common Properties of Widgets."

The Label Widget

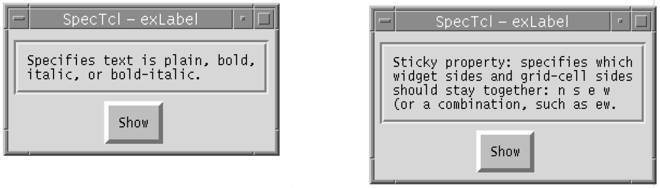
A label widget typically *labels* something else, as the "Name" label, below, identifies the entry widget. In addition, SpecTcl labels perform other display services explained later in this section.

Figure 6-1 Executing `exLabel1.ui`

A label can display text or an image, but not both.

Displaying Multiple Lines of Text

Labels can display multiple lines of text, as demonstrated in the figure by `exLabel2.ui.tcl` in execution:



Specifies text is plain, bold, italic, or bold-italic.

Sticky property: specifies which widget sides and grid-cell sides should stay together: n s e w (or a combination, such as ew.

Figure 6-2 A Label Displaying Multiple Lines of Text

The figure shows `exLabel2.ui` displaying two strings. With more text to display, a label automatically expands, in this case mostly in height.

The following properties are key in making the application executes as it does:

- The label's textvariable property is `ltext` (a Tcl variable).
- The label's wraplength property is 250, to constrain the display width.
- The label's justify property is `left`; see also, "Justify Property" on page 68.
- The button's command property is `ShowText`, so the `ShowText` proc is called when the button is pushed.

The figure shows `exLabel2.ui` in the design window (excerpt):

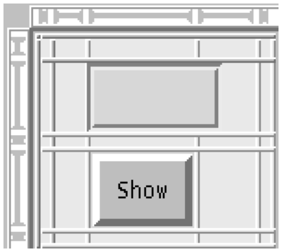


Figure 6-3 Design Window of `exLabel2.ui`

To view the script, open `exLabel2.ui` and select `Edit=>Edit Code`:

```
proc ShowText { } {
   global ltext toggle

   append sticky \
   "Sticky property: specifies which widget sides and grid-cell " \
   "sides should stay together: n s e w or a combination, such as ew."
```

```
   set tStyle \
   "Specifies text is plain, bold, italic, or bold-italic."

    # Flip/flop between displaying long and short strings
    set toggle [expr 1 ^ $toggle]
    if {$toggle} {
       set ltext $sticky
    } else {
       set ltext $tStyle
    }
}

global toggle
set toggle 0
```

The `set ltext` commands causes text to be displayed in the label, because `ltext` is the label's textvariable property. To create the long `sticky` string, the script uses the append statement, which concatenates its arguments. At execution, when you press the button, `ShowText` determines which string it's displaying by checking the string length, and toggles between the two strings.

Displaying an Image

Labels can also display an image file; for further information, see "Displaying an Image" on page 76, which describes buttons but applies equally to labels.

Important properties: anchor, justify, image, and textvariable.

About Buttons

Although the next section is titled "The Button Widget," in effect it describes characteristics common to buttons, checkbuttons, and radiobuttons. See also, "The Checkbutton Widget" on page 77 and "The Radiobutton Widget" on page 79.

The Button Widget

A button lets a user request an action, as specified by the button's command property. Specifically, when the user presses a button, radiobutton, or checkbutton, the Tcl commands in the widget's command property are executed.

Buttons typically display one or two words, such as “Save” or “OK,” but they can also display an image or multi-line text.

Displaying Multi-Line Text

Buttons can display multi-line text; for an example, see the multi-line label described in “The Label Widget” on page 73, which functions similarly.

Displaying an Image

A button can display an image, as demonstrated by running `exButton.tcl`:

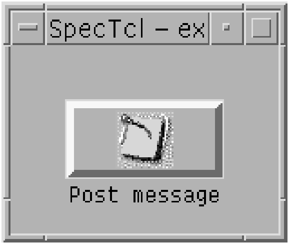
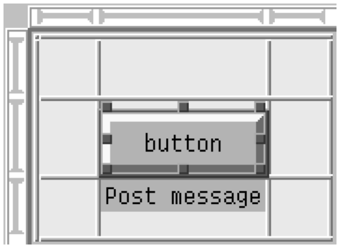


Figure 6-4 A Button with an Image

`Post message`, in the figure above, is in a separate label, because buttons can display text or an image, but not both simultaneously.

Here is the design window for `exButton.ui`, followed by its script:



```
proc ShowImage {w} {
    set iw [image create photo -file exButton.gif]
    $w config -image $iw
}

ShowImage .button#2
```

Figure 6-5 Design Window and Script of `exButton.ui`

The `ShowImage` proc, above, first creates an image attribute, using a `.gif` file, then it reconfigures the button with the image attribute.

Because the call to ShowImage (the last line of the script) is executed as the application is loaded, the user first sees the application with the image already loaded (as shown in Figure 6-4 on page 76).

Types of Images

There are two types of images:

- photo images, as shown above
- bitmap images

For photo images, only GIF and PPM/PGM formats are currently supported. For bitmap images, X11 bitmap format (e.g., as generated by the bitmap program).

Important properties: command, image, and textvariable.

The Checkbutton Widget

Checkbuttons let the user toggle options on or off, as demonstrated in the figure by `exCheckbutton.ui.tcl` (in the `examples` directory) in execution.

Options:
☑ Binary
☐ Hash
☑ Verbose

Figure 6-6 Executing `exCheckbutton.ui`

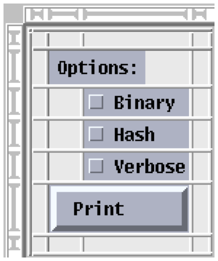
Variable Property - On/Off State

The variable property of checkbuttons specifies a Tcl variable that holds the on-off state of the checkbutton (usually 1 and 0).

If you have several checkbuttons, make certain the variable property of each one is unique, to prevent your buttons from turning each other on and off.

Showing Checkbutton Values

The figure shows the design window for `exCheckbutton.ui` (left) and its script (right):



```
proc ShowSw { } {
    global sw
    set swList [list Binary Hash Verbose]

    foreach i {0 1 2} {
       puts "[lindex $swList $i] is $sw($i)"
    }

    puts "\n"
}
```

Figure 6-7 Design Window and Script of `exCheckbutton.ui`

To view this directly, open `exCheckbutton.ui.tcl` in SpecTcl and then select `Edit=>Edit Code`.

These properties are key to operation of the script:

- The command property of each radiobutton is `ShowSw`, so that `ShowSw` is called when the checkbutton is pressed.
- The variable property of the checkbuttons is sw(0), sw(1), and sw(2), respectively; so that the on/off states of the checkbuttons are saved as elements of the sw array.

The `global` statement in the proc makes the sw array in the proc refer to the array elements in the variable properties of the checkbuttons.

When you press a checkbutton, the `puts` statement writes out the on/off values; for example:

```
Binary is 1
Hash is 0
Verbose is 1
```

For descriptions of generic button characteristics, see “About Buttons” on page 75.

Important properties: command, onvalue, offvalue, and variable.

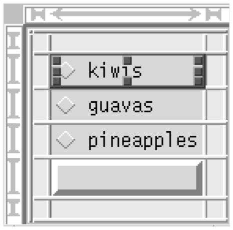
The Radiobutton Widget

Radiobuttons let the user select one alternative from a set, as demonstrated in the figure by `exRadiobutton.ui` in execution. Selecting one radiobutton turns the others off.

Figure 6-8 Executing `exRadiobutton.ui`

Referencing Radiobutton Values

Here is the design window for `exRadiobutton`.ui (left) and its script (right).



```
proc ShowButtons {} {
  global rbutton displayText

  # Display user's choice in a label,
  # using button's value as index into a list.
  set fruit_list [list kiwis guavas pineapples]
  set displayText [lindex $fruit_list $rbutton]
}
```

Figure 6-9 Design Window and Script of `exRadiobutton.ui`

Demonstrating the Radiobuttons

These properties are key to operation of the script:

- The variable property of each radiobutton is `rbutton`, which ties the radiobuttons together.

- The value property of the radiobuttons is 0, 1, and 2, respectively. One of these values is placed in rbutton when a radiobutton is pressed.
- The command property of each radiobutton is `ShowButtons`, so that `ShowButtons` is called when any radiobutton is pressed.
- The textvariable property of the label is `displayText`. When you set `displayText` to a string, the label displays the string.

At execution, when the user presses a radiobutton, `ShowButtons` is called. `ShowButtons` uses `rbutton`, the variable property, as an index into a list; the value of rbutton is 0, 1, or 2, depending on the radiobutton.

Another global, displayText, is the textvariable of the label. `ShowButtons` sets displayText to element `$rbutton` of the list.

For descriptions of generic button characteristics, see “About Buttons” on page 75.

Important properties: command, value, and variable.

About Menus

To create menus, you use the frame widget, menubutton widgets, and add commands, as described in the following sections.

The Menubutton Widget

A menubutton displays a menu when you press it, as demonstrated in the figure by `exMenubutton.ui` (in the `examples` directory) in execution:

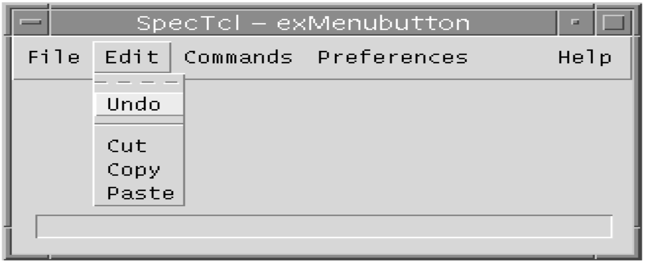


Figure 6-10 Menu Application

For generic button characteristics, see “About Buttons” on page 75.

Important properties: indicatorOn, menu, and textvariable.

The Menubar

At the top level of most applications with menus is a menu bar: a frame widget containing several menubuttons. The figure demonstrates this with the design window of `exMenubutton.ui`:

Figure 6-11 Design Window of `exMenubutton.ui`

To explain similar applications, we present the steps to recreate `exMenubutton.ui` (in the `examples` directory):

1. Drag a frame widget from the tool palette to the grid.

   This creates a subgrid with a single grid cell. You need five more cells. (This is a brief description of the process, but see "The Frame Widget" on page 95, for a better description.)

2. To create more subgrid cells, 1) click on the right wall of the subgrid cell to select the gridline, then 2) double-click on the gridline to create another cell.

3. Create five menubuttons, by dragging a palette menubutton to each subgrid cell (except for the empty one before `Help`).

   One subgrid cell is left empty so `Help` is right-adjusted, as customary.

4. Change the item_name property of each menubutton; for example, change menubutton#1 to fileMenubutton, menubutton#1, to editMenubutton, and so forth (names are in the script shown below).

5. Set the menu property of each menubutton to `m`.

6. Drag an entry widget from palette to grid.

Standard Button Menu Entries

To create a menu, use the menu command to create a menu object as the child of one of the menubuttons. Then add entries to it, as explained next.

Here are the commands to create the File and Edit menus (the other menus, which use checkbuttons and radiobuttons, are described later):

```
menu .fileMenubutton.m
     .fileMenubutton.m add command -label “Open”  \
          -command {puts “Open”}
     .fileMenubutton.m add command -label “Close”  \
          -command {puts “Close”}

menu .editMenubutton.m
     .editMenubutton.m add command -label "Undo"  -command \
           {puts "Undo"}
     .editMenubutton.m add separator
     .editMenubutton.m add command -label "Cut"   -command \
           {tk_textCut   .entry#1}
     .editMenubutton.m add command -label "Copy"  -command \
           {tk_textCopy  .entry#1}
     .editMenubutton.m add command -label "Paste" -command \
           {tk_textPaste .entry#1}
```

The `-command` option on `add command` is the command that is executed when the entry is selected. Most menu entries included here just identify themselves by writing out their names, but a few do more.

The `Copy`, `Cut`, and `Paste` commands transfer information between the clipboard and the entry widget, so you can try it out.

Checkbutton Menu Entries

Here are the commands to create the Preferences menus, which create menu entries that are checkbuttons:

```
menu .preferencesMenubutton.m
     .preferencesMenubutton.m add check -label “Opt1” \
          -variable opt1 \
          -command {puts “Opt1 is $opt1”}
     .preferencesMenubutton.m add check -label “Opt2” \
          -variable opt2 \
          -command {puts “Opt2 is $opt2”}
```

These differences characterize menus with checkbuttons:

- The command that adds entries is `add check`.
- When you add an entry, you specify a different variable property for each entry, as with other checkbuttons.

Radiobutton Menu Entries

Here are the commands to create the Style menus, which create menu entries that are radiobuttons:

```
menu .styleMenubutton.m
     .styleMenubutton.m add radio -label "plain" \
          -variable stylevar -value 0 \
          -command {puts "Style is $stylevar"}
     .styleMenubutton.m add radio -label "italic" \
          -variable stylevar -value 1 \
          -command {puts "Style is $stylevar"}
     .styleMenubutton.m add radio -label "bold" \
          -variable stylevar -value 2 \
          -command {puts "Style is $stylevar"}
```

These differences characterize menus with radiobuttons:

- The command that adds entries is `add radio`.
- When you add an entry, you specify the same variable property for all radiobutton, as with non-menu radiobuttons.
- Also, for each entry, you specify a unique value property.

Other Widgets

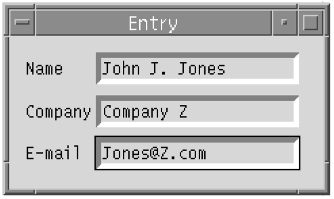
7

This chapter continues where the last chapter left off. It provides information that applies to other specific widgets—the entry, listbox, scale, text, frame, scrollbar, and canvas widgets. Specific applications mentioned, such as `exEntry.ui`, refer to applications in the `examples` directory.

If this chapter doesn't describe a property of one of these widgets, try Chapter 5, "Common Properties of Widgets."

The Entry Widget

The entry widget provides a one-line place for the user to enter text, as demonstrated in the figure by `exEntry.ui` in execution:



Name: John J. Jones
Company: Company Z
E-mail: Jones@Z.com

Figure 7-1 Executing `exEntry.ui`

In each of the three entry widgets, above, the user can enter text directly and use the usual editing commands.

When entering text, users must be able to signal when they're finished. As programmer, you can either supply a button for this, or have users press the Return key, or both. To connect the events for pressing the Return key to your script, you write bind commands, described later in this section.

Setting Properties for the Application

These properties are key to operation of the script:

- The item_name property of the entries are `entryName`, `entryCompany`, and `entryEmail`, respectively.
- The textvariable property of the entries are `ename`, `ecompany`, and `email`, respectively.

Here is the design window for `exEntry.ui` (left) and an outline of the script (right - details later):

```
bind Entry <Key-Return> {ShowEntry %W}
bind .entryEmail <Key-Return>
{ShowEmailEntry %W}

proc ShowEntry {w} {
    ...
}

proc ShowEmailEntry {w} {
    ...
}
```

Figure 7-2 Design Window and Script for `exEntry.ui`

To see the full script select `File=>Open`... for `exEntry.ui`, and then select `Edit=>Edit Code`.

When the User Presses Return

These bind commands transfer control to the `ShowEntry` and `ShowEmailEntry` procs when the user presses Return:

```
bind .entryEmail <Key-Return> {ShowEmailEntry %W}
bind Entry <Key-Return> {ShowEntry %W}
```

A bind command connects an event to the statements that process the event. To show two different ways to bind, let's connect the Key-Return event to:

- `ShowEntry` for all entry widgets.
- `ShowEmailEntry` for the .entryEmail widget.

In the `bind Entry` statement, `Entry` (a bindtag) refers to all entry widgets. This statement binds the Key-Return event for *any* entry widget to the ShowEntry proc. (There is also an `All` bindtag, with which you could bind the Key-Return event for *any* widget, since all widgets are referenced by the `All` bindtag.)

The bind `.entryEmail` statement connects the Key-Return event for the `.entryEmail` widget to the `ShowEmailEntry` proc.

The `%W` in the argument to either proc, means something special to SpecTcl. SpecTcl replaces `%W` with the name of the widget associated with the event.

For further information on the bind command and binding, see one of the Tcl books recommended in "Related Books" on page xvi.

Retrieving the Entry Text

The two procs just write out the widget that invoked them, the proc name, and the text that the user typed. Both procs are designed to let you enter text and press Return in the various entries and track what happens.

Here is the ShowEmailEntry proc

```
proc ShowEmailEntry {w} {
    global email
    append s "Widget name: $w \n"
    append s "proc name: ShowEmailEntry \n"
    append s "text: $email \n"
    puts $s
}
```

The global statement connects the `email` in the proc with the `email` that is the textvariable property of the entryEmail widget. The proc builds a string `s` with the information mentioned and writes it out.

Here is the ShowEntry proc:

```
proc ShowEntry {w} {
    append s "Widget name: $w \n"
    append s "proc name: ShowEntry \n"
```

```
    append s “text: [$w get] \n”
    puts $s
}
```

This proc is similar, but we don’t know the name of the widget, because pressing Return in any entry widget transfers control here. So, the widget name parameter, w, is used, and the `$w get` command fetches the text.

Processing Events Twice

A single event can trigger more than one action, if more than one bind statement is involved. When you try out the `exEntry.ui` application, note that when you press return in the E-mail-address entry, *both* procs are called.

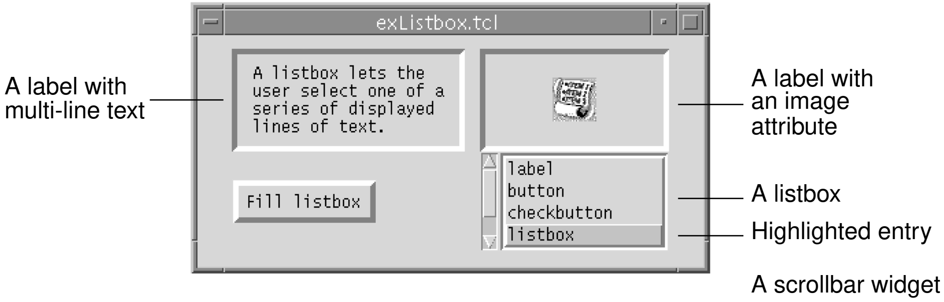
When this is inappropriate, you can avoid it; either: 1) bind each widget to a particular proc (that is, avoid the bind Entry statement), or 2) use only the bind Entry statement.

There is another way to avoid multiple event-handling calls, because the bind statements are executed in a particular order, with the more general ones executed last. So, you can place a `break` statement at the end of the proc that handles the event for the individual widget. This stops event processing for this event and avoids calling.

Important properties: exportselection and takefocus.

The Listbox Widget

The listbox widget lets the user select one of a number of displayed entries, as demonstrated by `exListbox.ui` (in the `examples` directory) in execution.



A listbox lets the user select one of a series of displayed lines of text.

Figure 7-3 Executing `exListbox.ui` Application

The user has clicked on the “listbox” entry, which is highlighted, and the application displays related text and a .gif-file image. If you open `exListbox.ui` and execute it, resize the application window if some of the text is not visible.

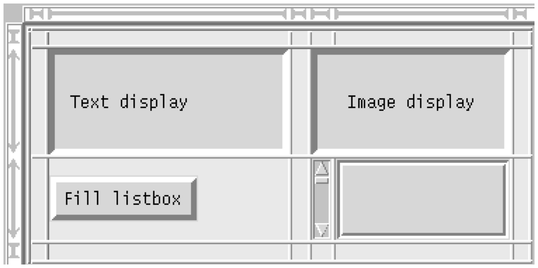
Important properties: xscrollcommand and yscrollcommand.

Setting Properties for the Application

To connect the widgets to the script, set these properties:

- For the button, set `command` to `ListboxInit`.
- For the labels, set item_name to `textLabel` (left) and `imageLabel` (right).

Here is the design window for `exListbox.ui` followed by a sketch of its script.



```
bind .listbox#1 <ButtonRelease-1> {ShowSel}
proc ListboxInit {} {
   ...
}
proc ShowSel { } {
   ...
}
```

Figure 7-4 Design Window and Script of `exListbox.ui`

When the User Selects a Listbox Entry

The bind command transfers control to the `ShowSel` proc when the user releases the first mouse button over any listbox entry:

```
bind .listbox#1 <ButtonRelease-1> {ShowSel}
```

Reacting to the User's Choice

As mentioned, when a user clicks on an entry in listbox#1, control passes to the `ShowSel` proc, to react to this event. The script includes one proc to initialize the listbox and another to determine the user's choice and react. Here is the first:

```
proc ListboxInit {} {
    # ListboxInit places a list of names in the listbox
    global txt

    set txt(label) "A label widget typically ..."
    set txt(button) "A button lets the user..."
    set txt(checkbutton) "A checkbutton lets the user..."
    set txt(listbox) "A listbox lets the user..."

    .listbox#1 delete 0 end

    foreach fname { label button checkbutton listbox } {
        .listbox#1 insert end $fname
    }
}
```

The `set txt(...)` statements place text to be displayed in an array. Note that the in array elements, such as `txt(button)`, the string indexes such as `button` are not predefined. The txt array is declared global so the other proc can use it.

The `.listbox#1 delete` command empties the listbox. Then `.listbox#1 insert` commands append `label`, `button`, and so forth in the `foreach` loop.

And here is the proc that reacts to the user's listbox selection:

```
proc ShowSel { } {
    # ShowSel displays text and an image file that correspond to
    # a user's listbox choice

    global txt

    # Find the user's choice and display the related text
    set i [.listbox#1 curselection]
    set choiceName [.listbox#1 get $i]
    .textLabel config -text $txt($choiceName)

    # Now display the related image file
    set fname exListbox.${choiceName}.gif
```

```
    set iw [image create photo -file $fname]
    .imageLabel config -image $iw
}
```

`ShowSel` is called after the user clicks on a listbox entry. The `.listbox#1 curselection` command returns an index (i, between 0 and n-1) into the listbox entries. The `.listbox#1 get $i` command gets the text of the i-th entry.

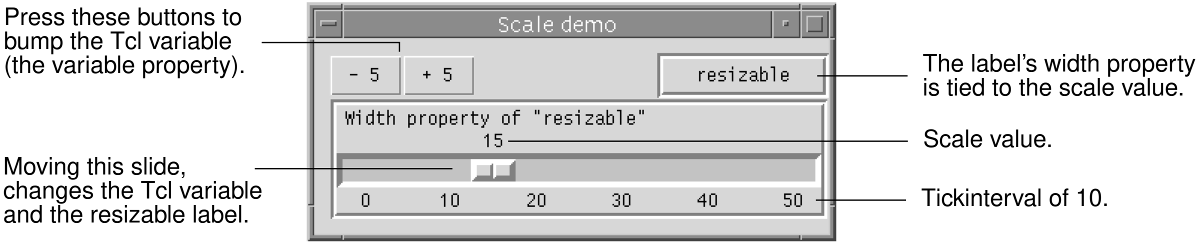
The i-th element of txt is the display text set in the first proc. The .gif files are conveniently named exListbox.label.gif, exListbox.button.gif, and so forth.

When you execute this example, be sure to expand the application window if some of the text doesn't fit at first.

Important properties: selectMode.

The Scale Widget

The scale widget, with its moveable slider, provides a way to view, and change, the value of a variable—graphically, as demonstrated by `exScale.ui` in execution:



Press these buttons to bump the Tcl variable (the variable property).

Moving this slide, changes the Tcl variable and the resizable label.

The label's width property is tied to the scale value.

Scale value.

Tickinterval of 10.

Figure 7-5 Executing `exScale.ui`

To demonstrate the scale widget, the application ties the scale widget's Tcl variable (its variable property) to the width property of a label (marked "resizeable"). So the scale's slide shows the width of the label, and when you move the slide, you resize the label.

Important properties: bigincrement, command, from, label, orient, showvalue, sliderlength, sliderrelief, takefocus, tickinterval, troughcolor, and variable.

Here is the design window for `exScale.ui` (in the `examples` directory):

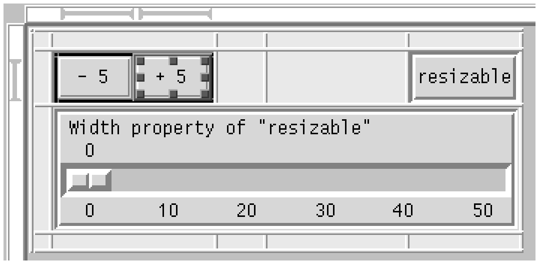


Figure 7-6 Design Window and Script of `exScale.ui`

Adding a Script

You can look at the command properties of the examples, but here are the main points:

- For the scale widget, the command property is `ShowVal`, so that when the scale value changes, the `ShowVal` proc is called.
- For the -5 button, the command property is:

  ```
  set x [.scale#1 get]; .scale#1 set [expr $x - 5]
  ```

  The scale widget (.scale#1) has its own `set` and `get` commands. They are used here to get the current scale value, subtract 5, and set the scale to the new value.
- The plus button is similar.
- If you select `Edit=>Edit Code`, you see:

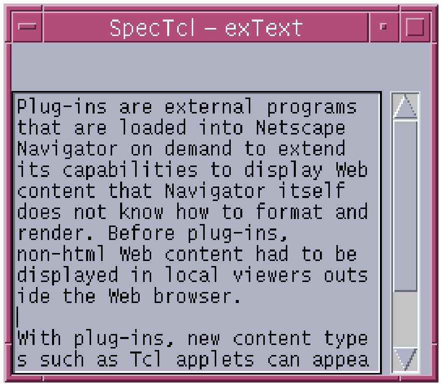
  ```
  proc ShowVal { } {
      global val

      .resizeLabel config -width $val
  }
  ```

  This proc reconfigures the “resizable” label so that it’s width changes directly with the value of the scale.

Note that as you move the scale to 0, the label grows wider, because the value 0 has a special meaning. It means the label should size itself to display its text.

The Text Widget

The text widget provides an easy, versatile way to display text to the user in specified fonts, sizes, and colors, as demonstrated by `exText.ui` in execution, below, which has both text widget and scrollbar widgets.



```
Plug-ins are external programs
that are loaded into Netscape
Navigator on demand to extend
its capabilities to display Web
content that Navigator itself
does not know how to format and
render. Before plug-ins,
non-html Web content had to be
displayed in local viewers outs
ide the Web browser.
|
With plug-ins, new content type
s such as Tcl applets can appea
```

Figure 7-7 Executing `exText.ui`

And here is the design window for `exText.ui`:

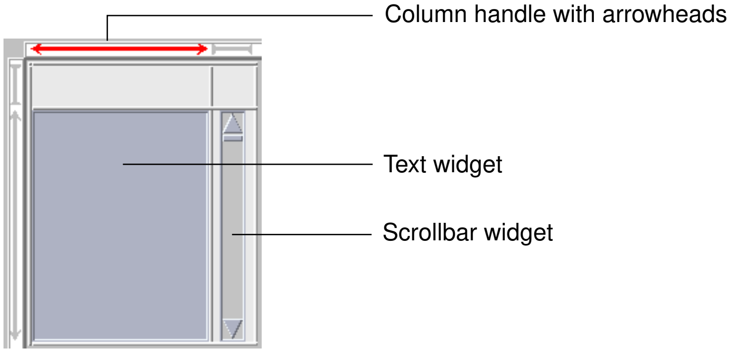


Figure 7-8 Design Window of `exText.ui`

You can open `exText.ui` in the `examples` directory or create it as follows:

1. Drag a text widget from the palette to one of the cells in column1.

2. To resize the text widget, drag the right column gridline of its grid cell to the right and the bottom gridline downward.

   When you move the gridlines, the widget sides move too, because of the widget’s default sticky property: `nsew`.

3. From the palette, drag a scrollbar widget to the cell that’s to the right of the text widget.

4. From the `Commands` menu, select `Attach Scrollbars`.

   This enables scrollbar movements to control the text widget; for further information, see “Attaching Scrollbars” on page 100.

5. Click once on the column handle, at the top of column 1, to select it, then click on it again to change the column handle to show arrowheads, as shown in the figure.

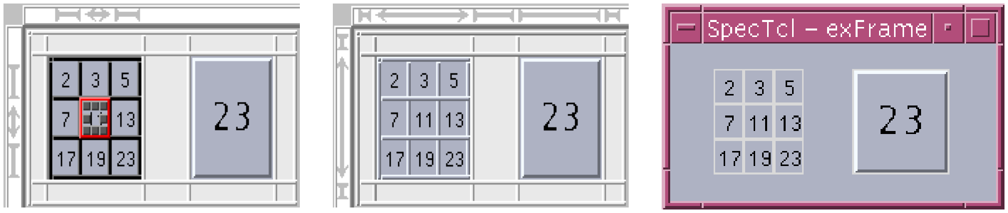
   The arrowheads indicate that resizeability has been turned on. Then, during execution, the text widget can grow wider—to display more text—when you widen the application window.

When you save the application and place it in execution:

- Copy and paste some text into the text widget; too much text to display at one time.
- Verify that the scrollbars work.
- Widen the application window, and verify that the text widget also widens.

The Frame Widget

The `exFrame.ui` application, shown below, demonstrates the frame's grouping capabilities. The application has a frame that's a 3 by 3 subgrid, with numbered buttons in subgrid cells. The large button (with 23) to the right of the frame is there for contrast: to show that the frame is subdividing a cell of the main grid.



Design time
Button in frame selected

Design time
Frame selected

Run-time

Figure 7-9 Designing and Executing `exFrame.ui`

Starting at the left, the figure shows two views of the application at design time: 1) with a button selected (a child of the frame) and 2) with the frame itself selected. At the right, it shows the application at run-time.

As a subgrid, the frame widget shares many features of the main grid:

- The frame has rows and columns, which you can add more of or delete.
- You can resize rows and columns to establish new minimum heights and widths, respectively.
- Rows and columns are resized automatically as widgets with different space requirements enter or leave.
- Each cell of the frame can contain at most one widget, which can also be another frame.
- If the frame has the appropriate sticky property (combinations that include `ns` or `ew`), the frame can pass extra space to those rows and columns that are set resizeable, as discussed later in this section.

Creating a Multi-Cell Subgrid

This figure shows the steps to create a frame with two columns:

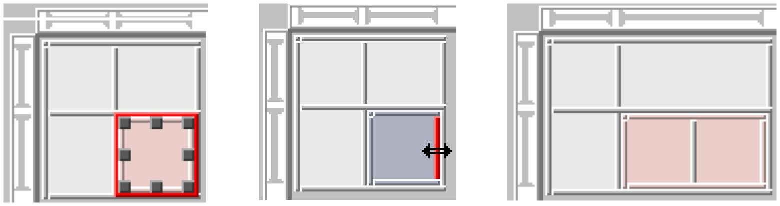


Figure 7-10 A Multi-Cell Subgrid

1. Drag a frame widget from the palette to the grid, as shown, above left.
2. With your mouse, if you move the cursor from left to right over the frame, you see a double-arrow and the gridline turns green, as shown above middle.
3. While it is green, double-click on it to create the additional column, as shown, above, right.

   Or, single clicking while it is green selects the gridline, so you can resize the column, as you would in the main grid.

Selection with a Subgrid Present

If you click repeatedly in a cell of the subgrid, the selection toggles between two states, shown in the figure:

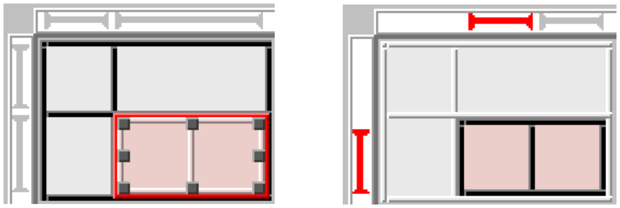


Figure 7-11 Selecting within the Grid and Subgrid

- In the figure on the left, you can tell that the entire frame is selected because its resize handles are visible.

  Use this selection to edit properties of the frame itself.

- In the figure on the right, you can tell that the first cell of the frame is selected, because its row and column handles are dark, delineating that cell.

  Use this selection to paste a widget into a cell of the subgrid.

When a frame is in the SpecTcl window, dark lines show whether the selection is within the main grid or the subgrid. When the selection is in the main grid, the main grid has dark lines; when the selection is within the subgrid, the subgrid has dark lines.

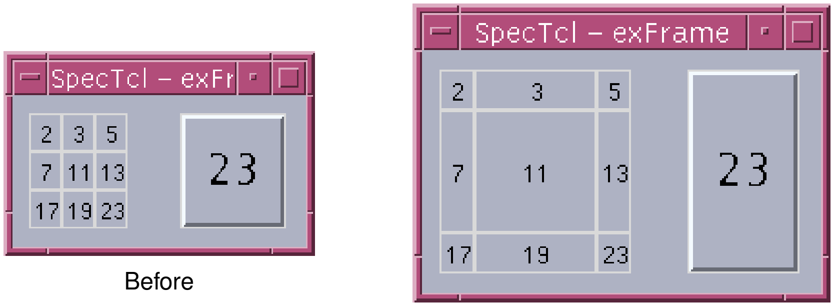
Selecting a Widget's Parent or Child

It's not always obvious how to select a frame. You can click on an empty cell in the frame *if there is one*. Otherwise, click on a widget within a cell, and press the *up-arrow key*, which selects the parent—the frame itself. Similarly, to select the first child, press the down-arrow key. You can navigate up and down a number of nested frames by using the up- and down-arrow keys. You can also move between parent and child widgets by selecting one of these menu commands:

- `Commands=>Navigate=>Select Parent`
- `Commands=>Navigate=>Select 1st Child`

Passing Window Space to Children

The figure shows the `exFrame.ui` application in execution before and after the user expands the application window:



Before

After

Figure 7-12 Executing `exFrame.ui` and Resizing the Application Window

Look at the design window for `exFrame.ui` again, as shown in Figure 7-9 on page 95. Resizeability was set on for both column 2 and row 2 and off elsewhere. Note that Figure 7-12 on page 97 shows that the application's expansion is consistent with these settings:

- The button at (2, 2), with the 11, is expanded both horizontally and vertically.
- Other buttons are expanded horizontally or vertically, but not both.

To prepare an application to work this way, do the following:

1. To set the resizeability of a row or columns on, click on the row or column handle involved, until you see the arrowheads that signal resizeability.

   You must do this for each row and column that is to change size as the user resizes the application window. Other rows and columns stay fixed.

2. Set buttons that are to expand horizontally to a sticky property that includes `ew`.

   In `exFrame.ui`, all buttons have a sticky property of `nsew`, which includes this step and the next.

3. Set buttons that are to expand vertically to have a sticky property that includes `ns`.

4. To select any button, click on it.

   This is a step towards selecting the frame.

5. To select the frame, select `Commands=>Navigate=>Select Parent`.

   Because the frame is, by definition, the parent of the widgets it contains.

6. To set the frame's sticky property, first select `Edit=>Widget Properties`...

   If there were an empty cell in the frame, you could double-click on it, but there isn't.

7. When the property sheet appears, set the sticky property to `nsew`.

   Or, you can use the sticky tool and select the largest element, in the lower right hand corner. Caution: If you use the sticky tool, you must choose the order specified here; if you do not have the resizeability of any row or column set on, your choice of sticky for the frame is restricted.

If the user expands the application window, the frame and its elements might get extra space, depending first on the resizeability of the frame's row and column. Extra space depends also on the frame's sticky property:

- Extra height - If the frame's sticky property contains `n` and `s`, the height of the frame expands to fill its grid cell and the frame can get extra height.
- Extra width - Similarly, if the frame's sticky property contains `e` and `w`, the width of the frame expands to fills its grid cell and the frame can get extra width.

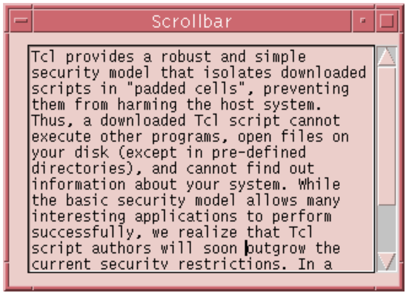
If the frame as a whole can get extra space, the widgets within the frame can also get extra. You can set each row and column within the frame as resizeable or not, the way you do for the main grid. For further information, see "Using Multiple Assemblies" on page 111.

The figure below shows the application

Important properties: selectMode.

The Scrollbar Widget

The scrollbar widget provides a scrolling capability for another widget. For example, you can use a scrollbar to scroll through lines of text in the text widget, as demonstrated by `exScrollbar.ui.tcl` in execution:



```
Tcl provides a robust and simple
security model that isolates downloaded
scripts in "padded cells", preventing
them from harming the host system.
Thus, a downloaded Tcl script cannot
execute other programs, open files on
your disk (except in pre-defined
directories), and cannot find out
information about your system. While
the basic security model allows many
interesting applications to perform
successfully, we realize that Tcl
script authors will soon outgrow the
current security restrictions. In a
```

Figure 7-13 Executing `exScrollbar.ui`

For large movements through the text, you can drag the scrollbar up or down. For small movements, click one of the arrows: a click moves by one line of text.

Attaching Scrollbars

To attach scrollbar to another widget, do the following:

1. Create a scrollbar widget next to a widget that works together with a scrollbar.
2. If the orientation of the scrollbar widget is wrong, use the orientation tool to change it.
3. Select `Commands=>Attach` Scrollbars.

   SpecTcl searches for scrollbar widgets that are adjacent in the grid to widgets that can accept scrollbars. It then changes the properties of the widgets concerned so that they work together.

Following the design window for `exScrollbar.ui`, shown below, we provide detailed steps.

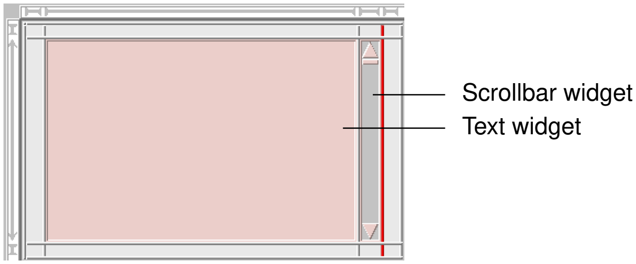


Figure 7-14 Designing `exScrollbar.ui`

To create the application:

1. Drag a text widget to the grid.
2. Drag a scrollbar widget to the grid cell next to the text widget.
3. Select `Command=>Attach Scrollbars`.

   SpecTcl links the two widgets for you.

Note – In case you have to undo it, here's some more information on `Attach Scrollbars`: 1) SpecTcl sets the command property of the scrollbar to refer to the text widget, `%B.text#1 yview`, and 2) it sets the yscrollbar property of the text widget to refer to the scrollbar, `%B.scrollbar#1 set`. If you delete the scrollbar widget, clear the text widget’s yscrollbar property to avoid an undefined reference to the scrollbar. (For further information, see the Tk documentation.)

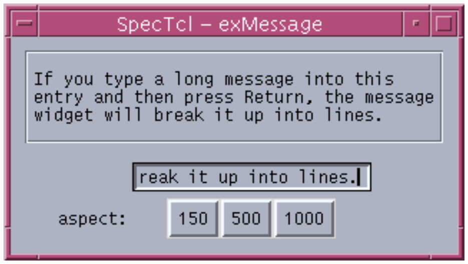
Important properties: jump and orient

The Canvas Widget

The canvas widget is a general-purpose widget that you can program to display a number of different objects, such as lines, polygons, images, and so forth. For further information on the canvas widget, see one of the Tcl/Tk books described in “Related Books” on page xvi.

The Message Widget

The message widget displays a long text string by breaking it up into several lines, as shown by the `exMessage.ui` application in execution:



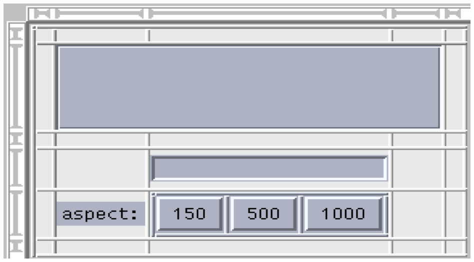
```
If you type a long message into this
entry and then press Return, the message
widget will break it up into lines.
```

```
reak it up into lines.
```

Figure 7-15 Executing `exMessage.ui`

The message widget’s aspect property controls the dimensions of the formatted text. When you execute this application, click on the different buttons to see the formatting affect with aspects of 150, 500, and 1000.

Here is the design window for `exMessage.ui` (left) and its script (right). Select `Edit=>Edit Code` to see (or edit) the code after opening `exMessage.ui`:



```
bind .entry#1 <Key-Return> {ShowText}

proc ShowText {} {
global entryText msgText
    set msgText $entryText
}
```

Figure 7-16 Design Window and Script for `exMessage.ui`

When you type a long text string into the entry widget and press Return, the message widget display the reformatted text. The `bind` statement in the script causes the `ShowText` proc to be called. `ShowText` simply sets one variable to the value of the other, but these global variables are the textVariable properties of the message widget and the entry widget, respectively.

The buttons reconfigure the message widget to have the various aspects.

Important properties: aspect, textVariable.

Tcl and Tk

8

This chapter gives you a quick start learning Tcl and shows a few examples of the way SpecTcl uses Tk. If you already know Tcl, you might prefer to skip it.

If you are new to Tcl, we include this chapter to:

- Provide you with enough basic information to understand the sample scripts quoted in this guide.
- Give you the flavor of the language, so you can decide whether to learn more.

If you plan to use Tcl very much, consider getting one of the excellent books on Tcl and Tk available through bookstores or publishers; for titles, see “Related Books” on page xvi in the preface.

About Tcl

Tcl is all about strings—in the form of commands, constants, variables, lists, and expressions, but still strings. Tcl determines what to do with different strings by their context within Tcl commands.

Tcl is interpreted, rather than compiled. This provides a lot of flexibility and makes it easy to try something, correct it, and try it again—without having to wait for compilation.

Entering Commands Interactively

One of the best features of Tcl is the ability to enter commands interactively and get immediate feedback as to whether you understand the command. We recommend you enter example scripts as you read this chapter.

Provided with Tcl is tclsh, an interactive shell; to start it, enter:

```
tclsh
```

Or, you can start `wish`, which is an interactive shell (released with Tcl/Tk) for building Tk applications:

```
wish
```

An interface window will appear. This is intended to display a Tk interface, but you can ignore it and Tcl commands are processed as with tclsh.

Using either tclsh or wish, you can enter a set command, like the following:

```
set x 123
```

and the shell responds:

```
=> 123
```

Note – The notation => is used after Tcl commands to indicate the result of the command, the string `123` in this case; => is not part of the result itself.

Tcl Commands

To discuss Tcl commands, we need a definition for word: A **word** is one or more contiguous "printing" characters. For example, here are three words:

- this_is_a_word
- 123.5
- /

Words are separated from each other by non-printing characters called **white space**: characters that don't print, such as space characters, tabs, and newlines. And by means of grouping, you can, in effect, include white space within words; for further information, see "Grouping" on page 107.

A Tcl command is a series of words. The first word in a command is the command name; subsequent words are command arguments. For example, here are `set`, `append`, and `puts` commands, respectively:

```
set x 5
append foo a b c
puts {Hello, World!}
```

A command typically ends at the end of the line. You can also end a Tcl command with a semicolon(;); for example:

```
    Set x 5.0; set y 7.5
```

Command Syntax

In summary, Tcl commands consists of a series of words interpreted as follows:

command-name arg1 arg2 arg3 ...

You can include white space in an argument by grouping; see “Grouping” on page 107.

The Tcl interpreter:

- Separates the words of a command into its name and arguments.
- Performs $ variable substitution, explained below.
- Passes command and command arguments to other procedures which interpreted the arguments on a command-by-command basis.

Commands that Span Lines

Commands end at the end of the line, unless the last character of the line is a backslash; for example,

```
set x \
5
```

Sets a variable x to 5, as expected.

Comments

Comments begin with a pound sign (#). The # must be the first word of the command. For example, this is *not* a valid comment, because the # becomes part of the set command:

set x 5 ~~# Begin initialization~~

Something similar, however, does work:

```
set x 5; # Begin initialization
```

A commands doesn’t usually need a semicolon at its end; the semicolon, above, signals that the # begins a new command and is therefore a comment.

Setting Variables

You don’t have to declare Tcl variables. They are defined when their values are first set—often in `set` commands such as this one:

```
set w .label#1
=> .label#1
```

The variable w now has as its value the name of the label.

```
set num 469
=> 469
```

```
set compound_rate 57.9
=> 57.9
```

```
set st "This is a string"
=> "This is a string"
```

These variables all contain character strings; the values 469 and 57.9 are *not* automatically converted to binary values as in some languages. You can nonetheless use numeric values in arithmetic expressions, as explained later.

Getting the Value of a Variable

To embed the value of a variable into a command, prefix the variable with a dollar sign; for example:

```
set i 5
=> 5
expr $i + 3
=> 8
```

In the `expr` (expression) command, above, the Tcl interpreter replaces $i by the value of i, 5, before the expression is evaluated.

Getting the Result of a Command

To embed the result of one command in another, enclose it in brackets; for example:

```
set i 5
```

```
=> 5
set j [expr $i + 3]
=> 8
```

Within the set j command, the `expr` command is evaluated first, as 8 and becomes:

```
set j 8
```

Grouping

Since white space usually separates Tcl command arguments, to include white space in argument requires quoting:

- Enclose the characters in double quotes; for example: "one word".
- Enclose the characters in braces {}; for example: {Also one word}.

To group characters *and enable* $ substitution, use double quotes; for example:

```
puts "The name of the widget is $w"
=> The name of the widget is .label#1
```

To group characters *and disable* $ substitution, use braces; for example,

```
puts {$x refers to the value of x.}
=> $x refers to the value of x.
```

Tcl Built-in Commands

There are many Tcl built-in commands, and we have already covered a few, such as `set` and `puts`.

Tip – Information about each built-in Tcl command comes with the Tcl release. For platform-specific ways to access it, see “Tcl Command Information” on page 109.

Here are two built-in commands that are used all the time.

proc

When you set the command property of a button, you can include any Tcl command, as explained in “Designing an Application” on page 24. This includes commands you define yourself—with the proc command.

The proc command has the following form:

```
proc proc-name { args } {
    proc-body
}
```

A proc can have zero or more arguments. Once you define a proc, you can use it the way you use any built-in Tcl command.

For example, here is a proc that just prints its arguments (to standard output or the console):

```
proc print {a b c} {
    puts “The values a, b, and c are: $a, $b, and $c”
}
```

To call it, you can place a command like the following in a button’s command property:

```
    print "whatever’s right" 5 7.9
```

And the following line is printed:

```
    The values a, b, and c are: whatever’s right, 5, and 7.9
```

List-related Commands

Here are a few commands that work with lists:

- `list arg1 arg2 ...`

  The `list` command creates a list from its arguments: arg1 becomes element 0, arg2 becomes element 1, and so forth. To use this command, you can embed it in a set command; for example,

  ```
  set fruits [list apples oranges grapefruit]
  ```

- `lindex` *list i*

  The lindex command returns the i-th element of the list; using the list created above:

  ```
  lindex $fruits 0
  => apples
  lindex $fruits end
  => grapefruit
  ```

  For the lindex command, `end` represents the last list value.

- `llength` *list i*

  The llength command returns the length of the list; using the list created above:

  ```
  llength $fruits
  => 3
  ```

Tcl Command Information

The way you access information about Tcl built-in commands depends on which platform you use. However, on all platforms there is an HTML help facility you can view with your network browser at:

```
http://sunscript.sun.com/man/tcl8.0/contents.html
```

This URL contains the Tcl/Tk Manual, including Tcl and Tk commands and keywords.

For MS Windows

When you select `Help=>Help` in Windows, a standard Windows Help facility window appears to describe Tcl and Tk commands. As it first appears, WinHelp displays the table of contents, as follows:

- Tcl Application
- Tcl Built-in Commands
- Tcl Library Procedures
- Tk Applications
- Tk Built-in Commands
- Tk Library Procedures

If you double-click on Tcl Built-in Commands, you can then go through the command definitions one by one. And there's a search facility.

For UNIX

There are UNIX man pages for all Tcl and Tk commands:

- For Tcl, see `man` page entries for: append, array, break, catch, ... (include the whole list). There is also a man entry for Tcl; for example, to learn about the list command, enter:
  ```
  man list
  ```
- For Tk, see `man` page entries for: button, label, ... (include the whole list).

Advanced Topics

9

This chapter describes advanced topics of SpecTcl.

Using Multiple Assemblies

A simple application typically uses a single `.ui.tcl` file. With a more complex application, it's sometime convenient to develop your user interface in parts, which we'll call assemblies, with each assembly having its own `.ui.tcl` file. During execution, your application must explicitly load the required assemblies. For example, you might have a listbox, scrollbars, and a text entry in one assembly, and a group of interacting radiobuttons in another.

Let's demonstrate multiple assemblies with an application that loads the same scale assembly twice. Although this won't happen much in practice, this shows that two assemblies can work together even when the widgets were originally assigned identical names. Figure 9-1 demonstrates this with `exAssemM.ui`, which has two frames and `exAssemS.ui.tcl`, which is loaded in each frame:

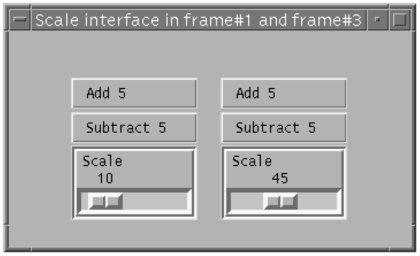


Figure 9-1 Executing `exAssemM.ui` - Subassemblies in Frames

If you select `File=>Open...` to open `exAssemM.ui` and select `Edit=>Edit Code`, you see:

```
source exAssemS.ui.tcl
exAssemS_ui .frame#1
exAssemS_ui .frame#3
```

This code loads the assembly twice: into frame#1 and frame#3, respectively.

(To load the same code into the main window would be:

```
source exAssemS.ui.tcl
exAssemS_ui .
```

)

Here is the command in the Add button.

```
set x [%B.scale#1 get]; %B.scale#1 set [expr $x + 5]
```

The command gets the value of the scale and then increments it by 5. As you see, the widget name, .scale#1 is qualified by %B, (base). SpecTcl expands %B to .frame#1 and .frame#3, when the scale assembly is loaded into frame#1 and frame#3, respectively. Widget names, name qualification, and the % abbreviations are explained further in the next section.

Widget Names in SpecTcl Scripts

The next few paragraphs discuss the facilities that enable assemblies to work correctly.

Introduction and Terminology

Suppose you refer to a widget name from within a SpecTcl script or command property. Then the form this widget name takes depends on whether you loaded the `.ui` file into the main window or into a frame.

Let’s start with some terminology. You might skim this now and come back later. **Root** is the window that contains all the other windows. **Base** is a qualifier to use as a prefix to the basic widget name: null for a main window, because no qualification is needed.

Main Window Assembly

Let’s consider the case with the assembly loaded into the main window:

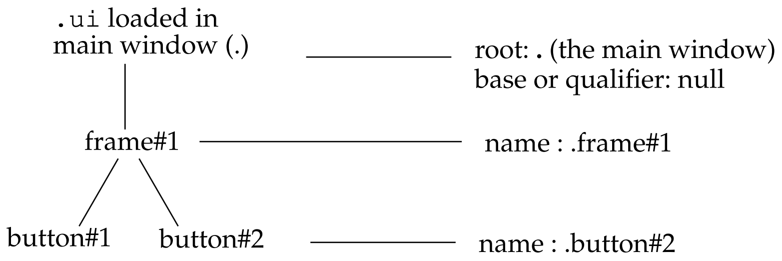


Figure 9-2 An Assembly in the Main Window

In the main-window case, you refer to all widgets in a command property *as if* they were top-level widgets; for example, you refer to button#2 as a `.button#2` even though it’s contained in frame#1.

Assembly in a Frame

Let’s contrast the last case with the `.ui.tcl` assembly loaded into a frame:

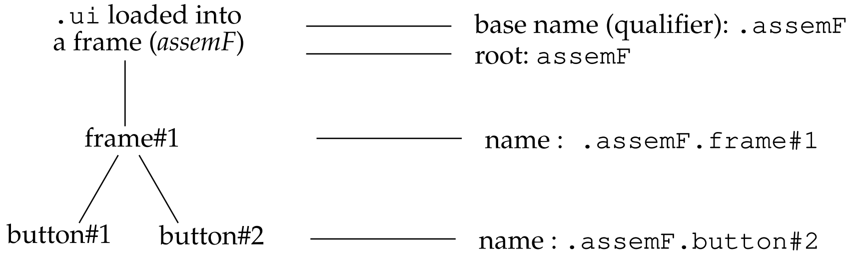


Figure 9-3 An Assembly in a Frame

In this case, you refer to all widgets in the command property *as if* they were directly contained in *assemF*; for example, you refer to button#2 as *assemF*`.button#2` even though it’s contained in frame#1.

Automatic Qualification by Base

When you drag a button onto the palette, the statements that SpecTcl generates in the `.ui.tcl` file are automatically qualified by the base. If the `.ui.tcl` file is loaded into a frame f3 the base is `.f3`; if it's loaded into the main window, the base is null, because no qualification is necessary.

Explicit Qualification by Base

When you select `Edit=>Edit Code` and enter a script you do not have to qualify widget names if you know the `.ui` file you're creating will always be loaded into the main window. However, if the `.ui` file might be loaded into a frame, by all means qualify any widget names by the base. To make this easier, SpecTcl provides some % substitutions, as explained next.

Substitutions in Commands

If you want to allow for the possibility that a `.ui.tcl` file might be loaded into a frame, you should use fully-qualified widget names.

Here are some per cent sign (%) substitutions that are convenient, but which you can only use in a widget command property. The names used as examples are widgets in Figure 9-3.

%B	Base name of the widget. This is the qualifier we've been discussing: null for the main window and `.assemF` for the example in the figure.
%M	Name of the geometry master—the direct container of the widget; `frame#1` for button#2 in the figure.
%W	Fullly-qualified name of the widget; for example, `.assemF.button#2 or .assemF.frame#1.`
%R	Name of the widget's root (parent of all widgets), which is `.` for the main window and `.assemF` for the figure.

To show you these % substitutions in working commands, here are the commands in the Add and Subtract buttons, respectively in the application in Figure 9-1 on page 111:

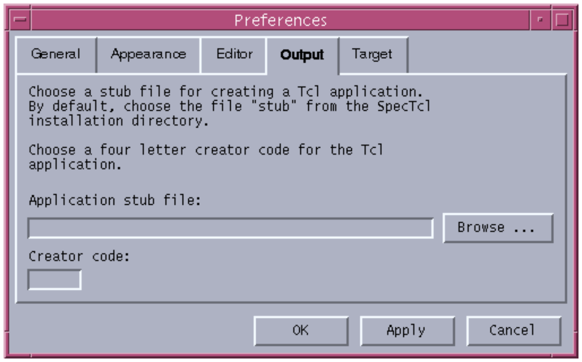
```
set x [%B.scale#1 get]; %B.scale#1 set [expr $x + 5]
set x [%B.scale#1 get]; %B.scale#1 set [expr $x - 5]
```

In the first application, %B is empty, and the references become simply `.scale#1`. In the second application, %B expands to either `.frame#1` or `.frame#3`, depending on the frame.

Building a Macintosh Application

On the Macintosh, a normal Build command creates a `ui.tcl` file. (If you double-click on this `ui.tcl` file, it runs SpecTcl, which isn't very useful.) To create a double-clickable version of your application, use this command: `Commands=>Build Application`... .

To bring an application into execution a "stub" file is used. To view (or edit) the choice of stub file or creator code, select `Preferences=>Options`... When the dialog box appears, click on the Output tab, and you will see a display like the following:



Choose a stub file for creating a Tcl application.
By default, choose the file "stub" from the SpecTcl
installation directory.

Choose a four letter creator code for the Tcl
application.

Application stub file:

Creator code:

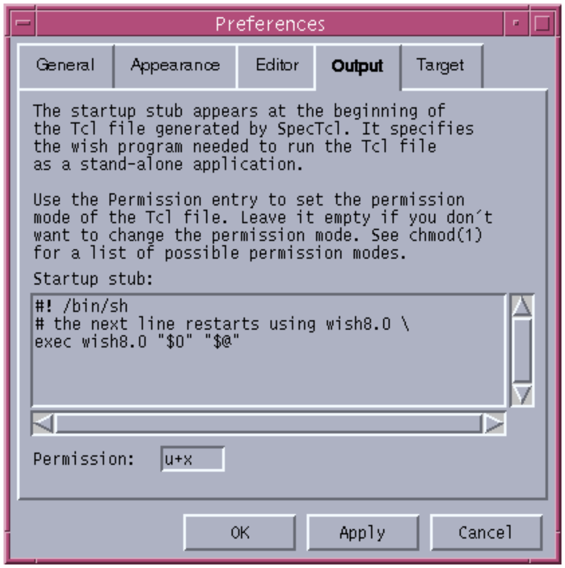
Figure 9-4 Macintosh Output Preferences

We ship a version of wish 8.0 that is used as a "stub" to create double-clickable applications. You can, however, override this stub file with your own modified or enhanced copy of wish. You can also change the creator code so that your generated application can use its own icons and so forth.

Execution Options in UNIX

In UNIX, you can execute the application’s `.ui.tcl` file the way you would any executable file. And, unless you would like to alter the execution defaults, you can skip the rest of this section.

Before changing the execution defaults, you need to understand the way the `ui.tcl` file works. SpecTcl begins each `ui.tcl` file with a stub that causes `wish` first to execute and then to interpret the Tcl statements in the file. To view or edit the stub, select `Preferences=>Options`... . When the dialog box appears, click on the Output tab, and you will see a display like the following:



The startup stub appears at the beginning of the Tcl file generated by SpecTcl. It specifies the wish program needed to run the Tcl file as a stand-alone application.

Use the Permission entry to set the permission mode of the Tcl file. Leave it empty if you don’t want to change the permission mode. See chmod(1) for a list of possible permission modes.

Startup stub:

```
#! /bin/sh
# the next line restarts using wish8.0 \
exec wish8.0 "$0" "$@"
```

Permission: u+x

Figure 9-5 Unix Output Preferences

This enables you to view and edit the stub or the file permissions that SpecTcl uses with the file.

Index