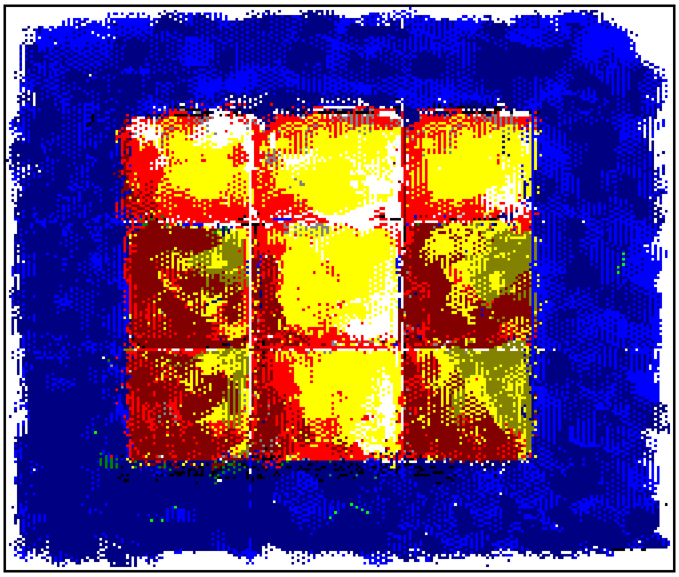
Tix Programming Guide



Ioi K. Lam

0.1 About This Manual

This manual is the programmer's guide to the Tix library. It takes you though a step-by-step tutorial about the different widgets and functions available in the Tix library. It also covers how to write new widgets using the Tix Intrinsics objecr-oriented programming interface. The accompanying *Tix Reference Manual* is a collection of the Tix manual pages. It describes all the options and other details of the Tix widgets and functions.

0.2 Other Formats Of This Document

This document is also available both postscript and HTML format. The postscript format is available at `ftp://ftp.xpi.com/pub/tix-4.0.ps.gz`. The HTML format is available from `http://www.xpi.com/tix/doc/tix-4.0/tix.book.html`

0.3 Organization of This Manual

Chapter 1, *Introduction* gets you started with the Tix widgets by describing their basic options and operations. Chapter 2, *Container Widgets*, describes the Tix widgets that can be used to contain other widgets and maintain their geometry. Chapter 3, *TList Widget and Display Items*, describes the tabular listbox widget and the Tix display items, which are used by many Tix widgets. Chapter 4, *Hierarchical Listbox*, describes how to create a hierarchical list structure using the TixHList widget. Chapter 5, *Selection Files and Directories*, describes how to use the file and directory selection widgets in the Tix library. Finally, Chapter 6, *Tix Object Oriented Programming*, describes how to use the Tix object oriented programming library to create new classes of Tix widgets.

Contents

Chapter 1

Introduction

1.1 What is Tix

1.1.1 Tix for Application Programmers

The acronym Tix stands for Tk Interface Extension. Tix is different things for different people.

If you are a GUI application programmer, that is, if you earn a living by building graphical applications, you will appreciate Tix as a library of *mega-widgets*: widgets made out of other widgets. To use a crude analogy, if the widgets in the standard TK library are bricks and mortars for a builder, the mega-widgets in the Tix library are walls, windows or even pre-build kitchens. Of course, these "bigger components" are themselves made of bricks and mortars, but it will take much less effort to put them together than planting bricks on top of each other.

The Tix widgets not only help you speed up the development of your applications, they also help you in the design process. Since the standard Tk widgets are too primitive, they force you to think of your house as, by using the same analogy, millions of bricks. With the help of the Tix mega-widgets, you can design your application is a more structural and coherent manner.

Moreover, the Tix library provides a rich set of widgets. Figure 1.1 shows all Tix widgets – there are more than 40 of them! Although the standard Tk library has many useful widgets, they are far from complete. The Tix library provides most of the commonly needed widgets that are missing from standard Tk: FileSelectBox, ComboBox, Control (a.k.a. SpinBox) and an assortment of scroll-able widgets. Tix also includes many more widgets that are generally useful in a wide range of applications: NoteBook, FileEntry, PanedWindow, MDIWindow, etc.

With all these new widgets, you can introduce new interaction techniques into applications, creating more useful and more intuitive user interfaces. You can design your application by choosing the most appropriate widgets to match the special needs of your application and users.

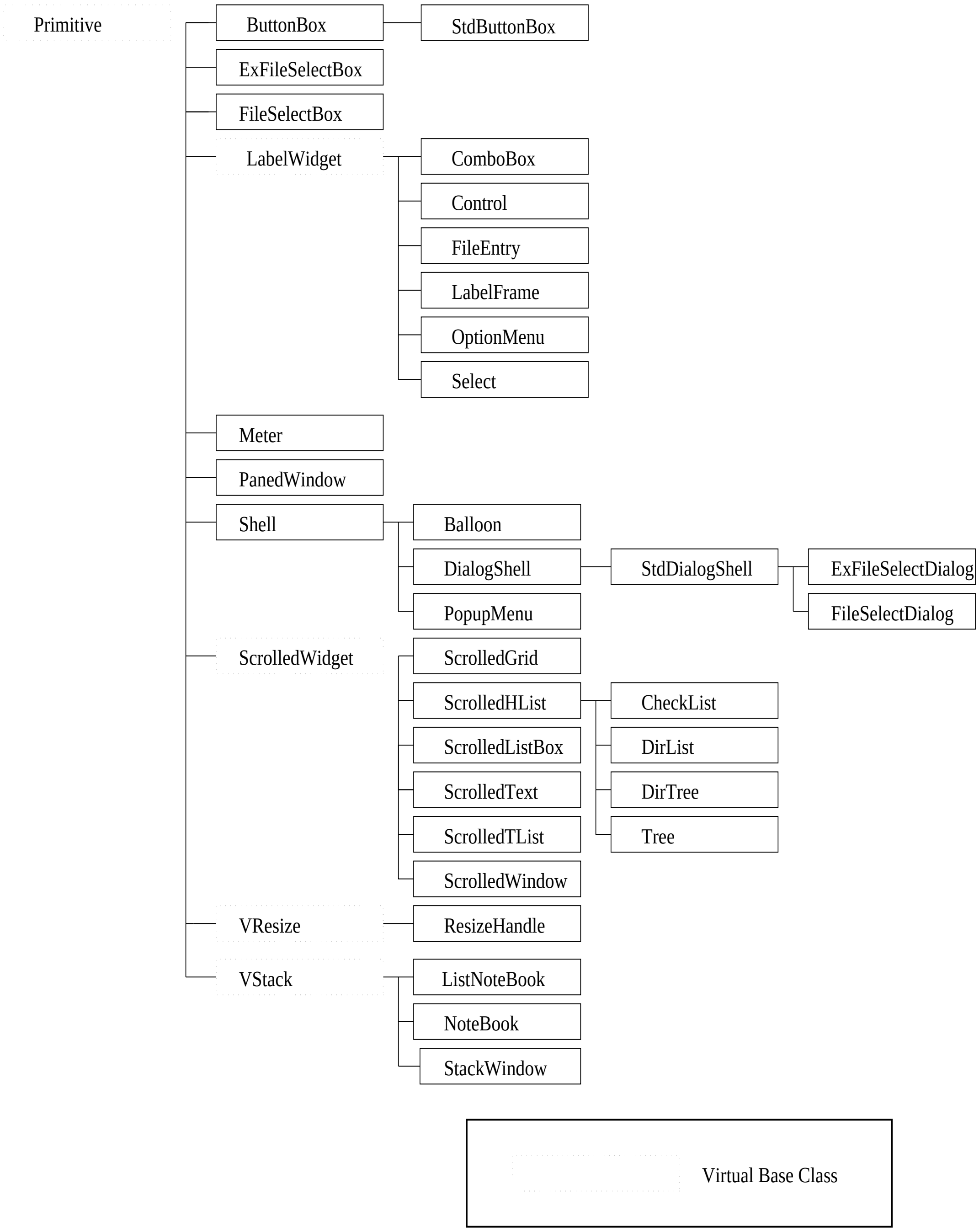


Figure 1.1: The Class Hierarchy of Tix Widgets

1.1.2 Tix for Widget Developers

On the other hand, if you are a widget developer, Tix provides an object oriented programming environment, the Tix Intrinsics, that is carefully designed for the development of mega-widgets. If you have developed widgets in C, you will know how slow and painful such a process would be. In recognition of the difficulties in widget development, the Tix Intrinsics includes many tools that dramatically cuts down the efforts required to develop new widgets. With the Tix Intrinsics, the rapid prototyping/development of widgets is finally feasible: you can write a new widgets in the matter of hours or even minutes.

With the Tix Intrinsics, you widget code can readily become reusable. Tix also provides a set of rules and mechanisms that allow you to develop widgets that are inter-operable with other widgets.

In Part I of this manual, we will talk about using the Tix widgets. The discussion of writing new Tix widgets will be carried out in Part II.

1.2 Getting Started: the TixControl Widget

Pre-requisites: you should be familiar with Tk widgets and programming, or read the Tk book along with this book

Before delving into the deep philosophy of the Tix widgets, let us first have a quick example to demonstrate the usefulness and convenience of an Tix widget: the TixControl is basically an entry widget that displays a value. Next to the entry, there are two up and down arrow buttons for you to adjust the value inside the entry widget.

1.2.1 Creating a TixControl Widget

The following code demonstrates how to create a TixControl widget and specify its options:

```
tixControl .lawyers -label Lawyers: -max 10 -min 0
.lawyers config -integer true -step 2
```

This example creates a TixControl widget that let us to select the numbers of lawyers we wish to be allowed in this country. (Figure 1.2)

Let us examine the options: the `-label` option specifies a caption for this widget. The `-max` option specifies the maximum number of lawyers we can choose. The `-min` option specifies the minimum number of lawyers we can choose: although we would love to enter a negative number, reality dictate that the lower limit must be zero. The `-integer` option indicates that the number of lawyers must be an integer; that is, we respect the lawyers' rights not to be chopped up into decimal points. Finally, since lawyers seem to go in pairs, we set the `-step` option to `2`, which indicates that when we press the up/down arrow buttons, we want the number of lawyers to go up and down by two each time.

As shown in the example, you can create and manipulate a Tix widget in the same manner as the standard Tk widgets. The options of the widget can be specified during

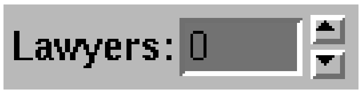


Figure 1.2: The TixControl Widget

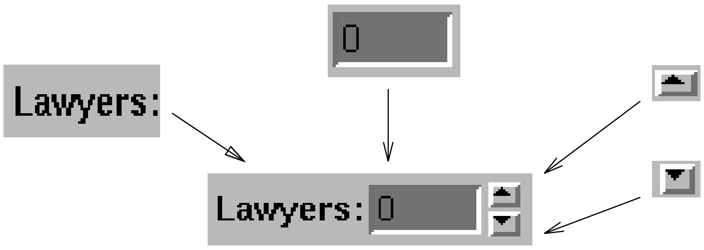


Figure 1.3: The Composition of TixControl

the creation of the widget. Alternatively, they can be changed by the `configure` widget command. In addition, options can also be specified in the option database or as X resources. Here is an example that produces the same result as the previous code fragment:

```
option add *lawyers.max 10
option add *lawyers.min 0
tixControl .lawyers -label Lawyers: -integer true
.lawyers config -step 2
```

In figure 1.3, you can see the composition of TixControl: it is made out of a label widget, an entry widget and two button widgets. Widgets that are composed of other widgets, like TixControl, are called *mega-widgets*. Most widgets in the Tix library are mega-widgets (xx: and as you know this book is about them!).

1.2.2 Accessing The Value of a TixControl Widget

The TixControl widget allows the user to input a value. There are several ways to read this value in your program. First of all, TixControl stores the current value in the `-value` option. You can use query the `-value` option by calling the command

```
.c cget -value
```

this command will return the current value of the tixContro widget `.c`. The following command sets the value of the widget to a new number (100):

```
.c config -value 100
```

The second way to access the value of TixControl is to use the `-variable` option. This options instructs the TixControl widget to store the its value into a global variable so that you can read it at any time. Also, by assigning a new value to this global variable, you can change the value of the TixControl widget. Here is an example:

```
.c config -variable myvar
set myvar 100
```

In some situations, you may want to be informed immediately when the value of the TixControl widget changes. To accomplish this, you can use the `-command` option. The following line causes the TCL procedure `valueChanged` to be called whenever the value of `.c` changes:

```
tixControl .c -command valueChanged
```

Disabling Callbacks Temporarily

Now, if you want to change a value from within the program, you have to disable the callback. The reason is that the callback runs whenever you (as well as the user) makes a change. In particular, if you make a change within the callback procedure and forget to disable the callback, it will recursively call itself and enter an infinite loop. To avoid this problem, you should use the `-disablecallback` option. Here is an example:

```
tixControl .c -command addOne

proc addOne {value} {
    .c config -disablecallback true
    .c config -value [incr value]
    .c config -disablecallback false
}
```

The procedure `addOne` adjusts the value of `.c` by one whenever the user enters a new value into `.c`. Notice that it is necessary to set `-disablecallback` here or otherwise `addOne` will be infinitely recursed! That is because `addOne` is called *every time* the value changes, either by the user or by the program.

1.2.3 Validating User Inputs

Sometimes it may be necessary to check the user input against certain criteria. For example, you may want to allow only even numbers in a TixControl widget. To do this, you can use the `-validatecmd` option, which specifies a Tcl command to call whenever the user enters a new value. Here is an example:

```
tixControl .c -value 0 -step 2 -validatecmd evenOnly

proc evenOnly {value} {
    return [expr $value - ($value %2)]
}
```

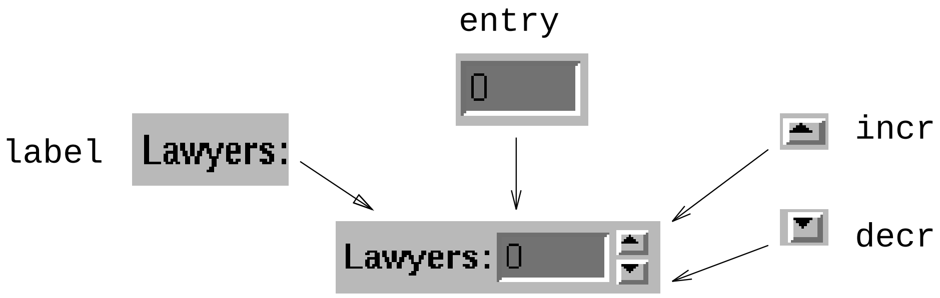


Figure 1.4: Subwidgets inside TixControl Widget

The value parameter to `evenOnly` is the new value entered by the user. The `evenOnly` procedure makes sure that the new value is even by returning a modified, even number. The Tcl command specified by the `-validatecmd` must return a value which it deems valid and this value will be stored in the `-value` option of the TixControl widget.

1.3 Accessing The Components Inside Mega Widgets

1.3.1 Subwidgets

As we have seen in section 1.2.1, the TixControl widget is composed of several widgets: one label widget, one entry widget and two button widgets. These "widgets inside mega-widgets" are called *subwidgets* in the Tix terminology. We will often have the need to access these subwidgets. For example, sometimes we need to change the configuration options of the subwidgets. In other cases, we may need to interact with the subwidgets directly.

1.3.2 Subwidget Names

Each subwidget inside a mega is identified by a *subwidget name*. Naturally, the label and entry subwidgets inside a TixSelect widget are called `label` and `entry`, respectively. The two button widgets are called `incr` and `decr` because they are used to `incr`ement and `decr`ement the value inside the TixControl widget (see figure 1.4).

1.3.3 The `subwidget` Method

All Tix mega-widgets support the `subwidget` method. This method takes at least one argument, the name of a subwidget. When you pass only one argument to this method, it returns the pathname of the subwidget which is identified by that name. For example, if .c is the pathname of a TixControl widget, the command:

```
.c subwidget entry
```

returns the pathname of the `entry` subwidget, which is `.c.frame.entry` in this case.

If you call the `subwidget` method with additional arguments, the widget command of the specified subwidget will be called with these arguments. For example, if `.c` is, again, the pathname of a TixControl widget, the command:

```
.c subwidget entry configure -bg gray
```

will cause the widget command of the `entry` subwidget of `.c` to be called with the arguments `configure -bg gray`. So actually this command will be translated into the following call:

```
.c.frame.entry configure -bg gray
```

which calls the `configure` method of the `entry` subwidget with the arguments `-bg gray` and changes its background color to `gray`.

We can call the `subwidget` method with other types of arguments to access different methods of the specified subwidget. For example, the following call:

```
.c subwidget entry icursor end
```

calls the `icursor` method of the `entry` subwidget with the argument `end` and sets the insert cursor of the `entry` subwidget to the end of its input string.

1.3.4 Chaining the `subwidget` Method

Some Tix mega-widgets may have subwidgets that in turn contain subwidgets. For example, the TixExFileSelectDialog (section 5.1.3) widget contains a TixExFileSelectBox subwidget called `fsbox`, which in turn contains a TixComboBox (section 1.4) subwidget called `dir`. If we want to access the `dir` subwidget, we can just "chain" the `subwidget` method. For example, if we have a TixExFileSelectDialog called `.file`, the following command will return the pathname of the `dir` subwidget of the `fsbox` subwidget of `.file`:

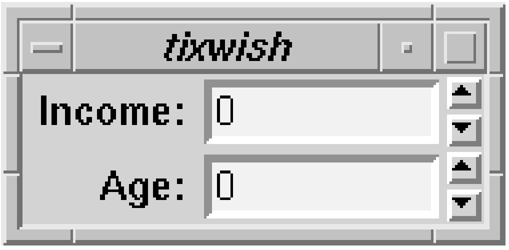
```
.file subwidget fsbox subwidget dir
```

Moreover, the following command configures the `dir` subwidget to have a border of the groove type with a border width of 2 pixels:

```
.file subwidget fsbox subwidget dir configure -bd 2 -relief groove
```

The chaining of the subwidget command can be applied for arbitrarily many levels, depending whether your widget has a subwidget that has a subwidget that has a subwidget that has a subwidget ... and so on.

(a) Unaligned Labels



(b) Aligned Labels

Figure 1.5: Using the `-options` Switch to Align the Labels

1.3.5 Configuring Subwidget Options Using the `-options` Switch

As we have seen above, we can use commands like "**`subwidget`** *name* **`configure ...`**" to set the configuration options of subwidgets. However, this can get quite tedious if we want to configure many options of many subwidgets.

There is a more convenient and terse way to configure the subwidget options without using the **`subwidget`** method: the **`-options`** switch. All Tix mega-widgets support the **`-option`** switch, which can be used during the creation of the mega-widget.

Program 1.1 Using the `-options` switch

```
tixControl .income -label "Income: " -variable income -options {
    label.width       8
    label.anchor      e
    entry.width       10
    entry.borderWidth 3
}
tixControl .age    -label "Age: "    -variable age    -options {
    label.width       8
    label.anchor      e
    entry.width       10
    entry.borderWidth 3
}
pack .income .age -side top
```

The use of the **`-options`** switch is illustrated in program 1.1, which creates two TixControl widgets for the user to enter his income and age. Because of the different sizes of the labels of these two widgets, if we create them haphazardly, the output may look like fig 1.5(a).

To avoid this problem, we set the width of the **`label`** subwidgets of the **`.income`** and **`.age`** widgets to be the same (8 characters wide) and set their **`-anchor`** option to **`e`** (flushed to right), so that the labels appear to be well-aligned. Program 1.1 also does other things such as setting the `entry` subwidgets to have a width of 10 characters and a border-width of 3 pixels so that they appear wider and "deeper". A better result is shown in figure 1.5(b).

As we can see from program 1.1, the value for the `-options` switch is a list of one or more pairs of

subwidget-option-spec value ..

subwidget-option-spec is in the form *subwidget-name*. *option-name*. For example, `label.anchor` identifies the `anchor` option of the `label` subwidget, `entry.width` identifies the `width` option of the entry subwidget, and so on.

Notice we must use the *name* of the option, not the *command-line switch* of the option. For example, the option that specifies the border-width of the `entry` subwidget has the command-line switch `-borderwidth` but its name is `borderWidth` (notice the capitalization on the name but not on the command-line switch). Therefore, we have used the capitalized version of "`entry.borderWidth 3`" in program 1.1 and not "`entry.borderwidth 3`". To find out the names of the options of the respective subwidgets, please refer to their manual pages.

1.3.6 Configuring Subwidget Options Using the Tk Option Database

The `-options` switch is good if you want to specify subwidget options for one or a few mega-widgets. If you want to specify the subwidget for many mega-widgets, it is easier to use the Tk Option Database.

Options in the Tk Option Database can be specified using the `option` command and the pathname of the widget. For all the Tix mega-widgets, it is guaranteed that the pathname of their subwidgets ends with the *name* of the subwidgets. For example, if we have a mega widget called `.a.b.megaw` and it has a subwidget whose name is `subw`, then we can be sure that the pathname of the subwidget will be something like

```
.a.b.megaw.foo.bar.subw
```

Therefore, if we want to specify options for it in the Option Database, we can issue commands like:

```
option add *a.b.megaw*subw.option1 value1
option add *a.b.megaw*subw.option2 value2
```

Notice that it will be wrong to issue the commands as:

```
option add *a.b.megaw.subw.option1 value1
option add *a.b.megaw.subw.option2 value2
```

because in general we will not know whether the subwidget is an immediate child window of `.a.b.megaw` [1].

[1] such a decision is left to the mega-widget implementor and may vary in different versions of the same mega-widget

Program 1.2 demonstrates how the Tk Option Database can be used to achieve the same effect as program 1.1.

Program 1.2 Using the Tk Option Database in Place of the `-options` switch

```
option add *TixControl*label.width       8
option add *TixControl*label.anchor      e
option add *TixControl*entry.width       10
option add *TixControl*entry.borderWidth 3

tixControl .income -label "Income: " -variable income
tixControl .age    -label "Age: "    -variable age

pack .income .age -side top
```

1.3.7 Caution: Restricted Access

In the current implementation of Tix, there is no limits on how you can access the options of the subwidgets. However, many options of the subwidgets may be already used by the mega-widget in special ways. For example, the `-textvariable` option of the `entry` subwidget of TixControl may be used to store some private information for the mega widget. Therefore, you should access the options of the subwidgets with great care. In general you should only access those options that affect the appearance of the subwidgets (such as `-font` or `-foreground`) and leave everything else intact. [2]

1.4 Another Tix Widget: TixComboBox

The *TixComboBox* widget is very similar to the ComboBox widgets available in MS Windows and Motif 2.0. A TixComboBox consists of an entry widget and a listbox widget. Usually, the ComboBox contains a list of possible values for the user to select. The user may also choose an alternative value by typing it in the entry widget. Figure 1.6 shows two ComboBoxes for the user to choose fonts and character sizes. You can see fro the figure that a listbox is popped down from the ComboBox for fonts for the user to choose among a list of possible fonts.

1.4.1 Creating a TixComboBox Widget

In program 1.3, we set up a ComboBox `.c` for the user to select an animal to play with. If the user is just a dull person like you and me, he would just press the arrow button and select a pre-designated animal such as "dog". However, if he wants to try something new, he could type "micky" or "sloth" into the entry widget and he will get to play with his favorite animal.

[2] In future versions of Tix, there will be explicit restrictions on which subwidget options you can access. Errors will be generated if you try to access restricted subwidget options

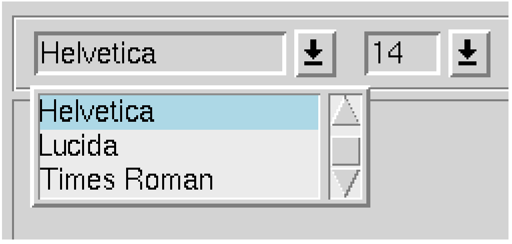


Figure 1.6: The TixComboBox Widget

Program 1.3 Creating a ComboBox

```
tixComboBox .c -label "Animal:" -editable true
.c insert end cat
.c insert end dog
.c insert end pig
```

Of course, sometimes we don't want too many sloths around us and we want to limit the range of the user's selections. In this case we can do one of two things. First, we can set the `-editable` option to `false` so that the user cannot type in the entry widget at all. Alternatively, we can use the `-validatecmd` option (see section 1.4.3) to check input the input.

1.4.2 Controlling the Style of the TixComboBox

The TixComboBox widget can appear in many different styles. If we set the `-dropdown` option to `true` (which is the default), the listbox will only appear when the user presses the arrow button. When `-dropdown` is set to `false`, the listbox is always shown and the arrow button will disappear because it is not needed anymore.

There is also an option called `-fancy`. It is set to `false` by default. When set to `true`, a tick button and a cross button will appear next to the entry widget. The tick button allows you to select again the value that's already in the ComboBox. If you press the cross button, the entry widget will be cleared.

1.4.3 Static Options

The `-dropdown` and `-fancy` options are so-called "static options". They can only be set during the creation of the ComboBox. Hence this code is valid:

```
tixComboBox .c -dropdown true
```

But the following code will generate an error because it attempts to set the `-dropdown` option *after* the ComboBox has already been created.

```
TixComboBox .c
```

```
.c config -dropdown true
```

The restrictions of the static options, although annoying, nevertheless make sense because we don't want our interface to suddenly change its style. If sometimes a button is there and sometimes it disappear all by itself, that will certainly create a lot of confusion and makes the user wonder why he should buy our software. Also, as you will see in chapter 6, having some static options will make the life of widget writers a lot easier.

Accessing the value of the ComboBox is very similar to accessing the value of the TixControl widget. The ComboBox has these four options, which we discussed in section 1.2.2: `-value`, `-variable`, `-command` and `-validatecmd`. You can use these four options to access the user input and respond to user actions in exactly the same way as discussed in section 1.2.2.

1.4.4 Monitoring the User's Browsing Actions

When the user drags the mouse pointer over the listbox, the listbox item under the pointer will be highlighted and a "browse event" will be generated. If you want to keep track of what items the user has browses through, you can use the `-browsecmd` option. Here is an example:

```
tixComboBox .c -browsecmd mybrowse
....

proc mybrowse {item} {
    puts "user has browsed $item"
}
```

When the Tcl command specified by the `-browsecmd` option is called, it will be called with one parameter: the current item that the user has highlighted.

The `-browsecmd` is useful because it gives the user the possibility of temporarily seeing the results of several choices before committing to a final choice.

For example, we can list a set of image files in a ComboBox. When the user single-clicks on an item on the ComboBox, we want to show a simplified view of that image. After the user has browsed through several images, he can finally decide on which image he wants by double-clicking on that item in the listbox.

The following is some pseudo Tcl code that does this. Please notice that the `-browsecmd` procedure is called every time the user single-clicks on an item or drags the mouse pointer in the listbox. The `-command` procedure is only called when the user double-clicks on an item.

```
tixComboBox .c -dropdown false -browsecmd show_simple -command load_fullsize
.c insert end "/pkg/images/flowers.gif"
.c insert end "/pkg/images/jimmy.gif"
.c insert end "/pkg/images/ncsa.gif"
```

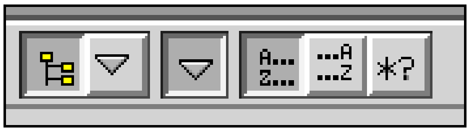


Figure 1.7: The TixSelect Widget

```
proc show_simple {filename} {
    # Load in a simplified version of $filename
}

proc load_fullsize {filename} {
    # Load in the full size image in $filename
}
```

As we shall see, all Tix widgets that let us do some sort of selections have the -browsecmd option. The -browsecmd option allows us to respond to user events in a simple, straightforward manner. Of course, you can do the same thing with the Tk bind command, but you don't want to do that unless you are very fond of things like <Control-Shift-ButtonRelease-1> and "%x %X $w %W %w".

1.5 The TixSelect Widget

The TixSelect widget figure 1.7 provides you the same kind of facility that is available with the Tk radiobutton and checkbutton widgets. That is, TixSelect allows the user to select one or a few values out of many choices. However, TixSelect is superior because it allows you to layout the choices in much less space than what is required by the Tk radiobutton widgets. Also, TixSelect supports complicated selection rules. Because of these reasons, TixSelect is a primary choice for implementing toolbar buttons, which often have strict space requirements and complicated selection rules.

1.5.1 Creating A TixSelect Widget

Program 1.4 shows how to create a TixSelect widget. At line 1 of program 1.4, we create a TixSelect using the the tixSelect command.

Program 1.4 Creating a TixSelect Widget

```
tixSelect .fruits -label "Fruits: " -orientation horizontal
.fruits add apple  -text Apple  -width 6
.fruits add orange -text Orange -width 6
.fruits add banana -text Banana -width 6
pack .fruits
```

(a) Horizontal Orientation

(b) Vertical Orientation

Figure 1.8: The TixSelect Widget

Label and Orientation

As shown in program 1.4, with the `-label` option, we can put a label next to the button subwidgets as the caption of the TixSelect widget. We can also control the layout of the button subwidgets using the `-orientation` option. The `-orientation` option can have two values: `horizontal` (the default value) or `vertical`, and the buttons are lied up accordingly. Figure 1.8(b) shows the output of a TixSelect widget whose `-orientation` is set to `vertical`.

Creating the Button Subwidgets and Configuring Their Appearance

After we have created the TixSelect widget, we can create the button subwidgets inside the TixSelect widget by the `add` widget command (lines 2-4 of program 1.4).

The first argument to the `add` command is the name of the button subwidget. Additional arguments can be given in *option-value* pairs to configure the appearance of the button subwidget. These *option-value* pairs can be any of those accepted by a normal TK button widget. As shown in program 1.4, we use the `-text` option to put appropriate text strings over the three button subwidgets.

Notice that we also set the `-width` option of all the button subwidgets to 6 characters. This way, the three buttons will have the same width. If we didn't set the `-width` option for the button widgets, they will have different widths, depending on their text string, and the result would look less esthetically pleasing than buttons with same widths.

The output of program 1.4 is shown in figure 1.8(a)

Accessing the Button Subwidgets

We have already seen the concept of subwidgets and how they can be accessed in section 1.3.1 — when we create a Tix mega-widget, some subwidgets are created for us automatically. For example, the label and entry subwidgets inside a TixControl widget. We can access these subwidgets in a multitude of ways, including using the subwidget method.

One thing about the subwidgets we saw in section 1.3.1 is that they are "static", meaning they are created when the mega-widget is created and they remain there for the whole lifetime of the mega-widget.

The TixSelect widget takes us to a new concept: *dynamic subwidgets* are subwidgets that can be created on-the-fly. After we add a new button into the TixSelect widget, we get a new subwidget. The name of this new subwidget is given by the first parameter passed to the `add` method. As the following code demonstrates, we can access this new subwidget using the `subwidget` method:

```
tixSelect .s
.s add apple  -text Apple
.s add orange -text Orange
#   Mmmm..., let's make the widget look more educated
#   by using French words
.s subwidget apple  config -text Pomme
.s subwidget orange config -text Orange
```

1.5.2 Specifying Selection Rules

For simple selection rules, you can use the `-allowzero` and `-radio` options. The `-allowzero` option specifies whether the user can select none of the choices inside the TixSelect widget. The `-radio` option controls how many buttons can be selected at once: when set to true, the user can select only one button at a time; when set to false, the user can select as many buttons as he desires.

With these two options, we can write a program using two TixSelect widgets for little Jimmy to fill up his lunch box. On the Sandwich side, we set `-radio` to true and `-allowzero false`. That means Jimmy can select one and only one sandwich among beef, cheese or ham sandwiches. On the Veggie side, we want to encourage Jimmy to consume as much veggie as possible, so we set the `-allowzero` option to `false`. We also set the `-allowzero` option to `false` so that Jimmy cannot get away with eating none of the vegetables (see program 1.5).

Program 1.5 Specifying Simple Selection Rules

```
tixSelect .sandwich -allowzero false -radio true -label "Sandwich :"
.sandwich add beef   -text Beef
.sandwich add cheese -text Cheese
.sandwich add ham    -text Ham

tixSelect .vege -allowzero false -radio false -label "Vegetable :"
.vege add bean    -text Bean
.vege add carrot  -text Carrot
.vege add lettuce -text Lettuce
```

1.5.3 Accessing the Value of a TixSelect Widget

The *value* of a TixSelect widget is a list of the names of the button subwidgets that are currently selected. For example, in program 1.4, if the user has selected the apple button, then the value of the TixSelect widget will be `apple`. If the user has selected both the apple and the orange buttons, then the value will be the list `"apple orange"`.

The TixSelect widget supports same set of options as the TixControl widget for you to access its value: the `-value` option stores the current value, which can be queried and modified using the cget and configure methods. You can also use the `-variable` option to specify a global variable to store the value of the TixSelect widget. The `-command` option specifies a TCL command to be called whenever the user changes the selection inside a TixSelect widget. This command is called with one argument: the new value of the TixSelect widget. There is also the `-disablecallback` option which you can use to control whether the command specified by the `-command` option should be called when the value of the TixSelect changes.

1.5.4 Specifying Complex Selection Rules

If you want to have more complex selection rules for the TixSelect widget, you can use the `-validatecmd` option. This option works the same as the `-validatecmd` option of the TixControl widget which we discusses in section 1.2: it specifies a command to be called every time the user attempts to change the selection inside a TixSelect widget.

In the example program 1.6, the procedure `TwoMax` will be called every time the user tries to change the selection of the `.fruits` widget. `TwoMax` limits the maximum number of fruits that the user to choose to be 2 by always truncating the value of the TixSelect widget to have no more than two items. If you run this program, you will find out that you can never select a third fruit after you have select two fruits.

Program 1.6 Specifying More Complex Selection Rules

```
tixSelect .fruits -label "Fruits: " -radio false -validatecmd TwoMax
.fruits add apple  -text Apple  -width 6
.fruits add orange -text Orange -width 6
.fruits add banana -text Banana -width 6
pack .fruits

proc TwoMax {value} {
   if {[llength $value] > 2} {
      return [lrange $value 0 1]
   } else {
      return $value
   }
}
```

Chapter 2

Container Widgets

In addition to providing some nice-looking interface elements, Tix offers some useful ways to organize the elements that you create. It does this by providing *container widgets*, which are widgets designed to contain whatever you want to put into them.

Different container widgets have different policies as to how they arrange the widgets inside them. In this chapter, we'll talk about TixNoteBook, which arranges its subwidgets using a notebook metaphor, TixPanedWindow, which arranges its subwidgets in non-overlapping horizontal or vertical panes, and a family of "Scrolled Widgets", which attach scrollbars to their subwidgets.

2.1 TixNoteBook

When your need to put a lot of information into your interface, you may find out that your window has to grow intolerably big in order to hold all the information. Having a window that's 10000 pixels wide and 5000 pixels high doesn't seem to be the perfect solution. Of course, you can "chop up" your big window into a set of smaller dialog boxes, but the user will most likely find it impossible to manage 20 different dialog boxes on their desktop.

The TixNoteBook (fig 2.1) widget comes into rescue. It allows you to pack a large interface into manageable "pages" using a notebook metaphor: it contains multiple pages with anything you want on them, displays one at a time, and attaches a tab to each page so the user can bring it forward with a single click on the tab.

2.1.1 Adding Pages to a TixNoteBook

The example program in figure 2.1 creates the TixNoteBook widget shown in figure 2.1. In the first three lines, we create the notebook widget and two pages inside it. While we create the pages, we also set the labels on the tabs associated with each page and use the `-underline` option to indicate the keyboard accelerator for each page.

Each time we create a page in the notebook using the `add` method, a frame subwidget

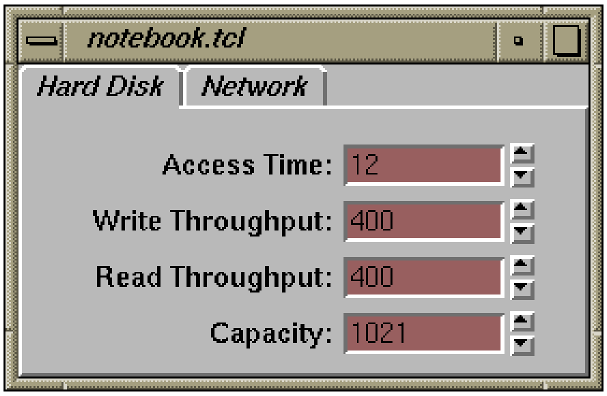


Figure 2.1: The TixNoteBook Widget

is created for us automatically. This frame subwidget has the same name as the page (the first parameter passed to the `add` method). We can use the `subwidget` method to find out the pathname of this frame subwidget and pack everything we want to display on the page into this frame widget. Lines 4-10 of program 2.1 shows how to create the widgets inside the "Hard Disk" page. Creating the widgets inside the "Network" page will be similar.

Program 2.1 Using The TixNoteBook Widget

```
tixNoteBook .n
.n add hd  -label "Hard Disk" -underline 0
.n add net -label "Network"   -underline 0

set frame [.n subwidget hd]
tixControl $frame.access   -label "Access Time:"
tixControl $frame.write    -label "Write Throughput:"
tixControl $frame.read     -label "Read Througput:"
tixControl $frame.capacity -label "Capacity:"
pack $frame.access $frame.write $frame.read $frame.capacity \
    -side top -fill x
```

2.1.2 Keyboard Accelerators

Note that in line 2-3 of program 2.1, we have indicated the keyboard accelerators for the two pages using the `-underline` option. The value of this option is the position of the character to be underlined in the string, where zero represents the first character. When the user presses `<Alt-N>` or `<Meta-N>` the "Network" page will be activated; on the other hand, if he presses `<Alt-H>` or `<Meta-H>` the "Hard Disk" page will be activated. The TixNoteBook widget will automatically create the keyboard bindings for these accelerators for us, in a way similar to what the menu widget does, so there is no need to set the keyboard bindings ourself.

2.1.3 Delaying the Creation of New Pages

If your notebook contains many complicated pages, it may take quite a while to create all widgets inside these pages and your program will probably freezes for a few seconds when it pops up the notebook for the first time. To avoid embarrassing moments like this, we can use the "delayed page creation" feature of the TixNoteBook widget.

When we create a page using the `add` method, we can specify the optional parameter `-createcmd` so that we only need to create the page when the user wants to see it. This is illustrated in program 2.2:

Program 2.2 Delayed Page Creation

```
tixNoteBook .n
.n add hd  -label "Hard Disk" -underline 0 -createcmd CreateHd
.n add net -label "Network"   -underline 0 -createCmd CreateNet

proc CreateHd {frame} {
    tixControl $frame.access   -label "Access Time:"
    tixControl $frame.write    -label "Write Throughput:"
    tixControl $frame.read     -label "Read Througput:"
    tixControl $frame.capacity -label "Capacity:"
    pack $frame.access $frame.write $frame.read $frame.capacity \
        -side top -fill x
}

proc CreateNet {frame} {
    ...
}
```

In line 2 of program 2.2, we use the `-createcmd` option to specify that the procedure `CreateHd` should be called when the "Hard Disk" page needs to be created. `CreateHd` takes one argument, the frame subwidget of the page. As we can see, program program 2.2 is not very different than program 2.1, except now we can issue less commands during the set-up of the NoteBook widget and the interface can be started up more quickly.

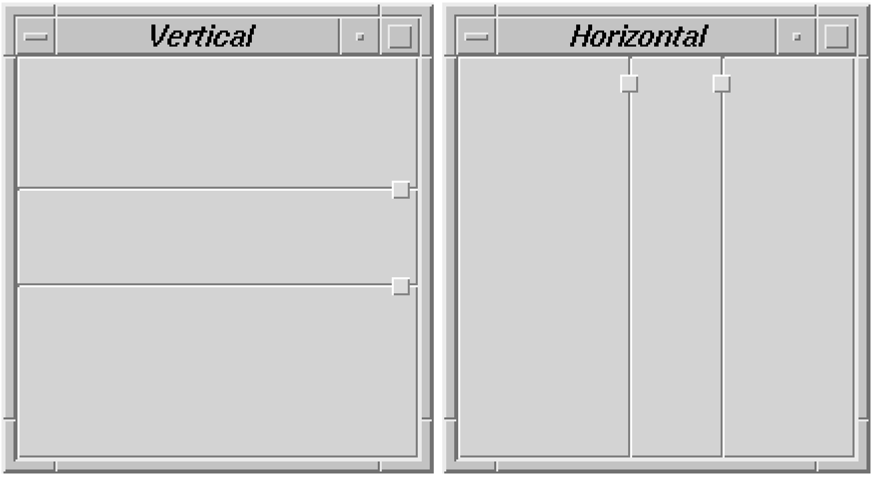
2.1.4 Changing Page Tabs and Deleting Pages

To change the information in the tabs of the pages, we can use the `pageconfigure` method. For example, the following command:

```
.nb pageconfigure hd -label "Fixed Disk"
```

changes the label from "Hard Disk" to "Fixed Disk". To delete a page, we can use the `delete` method.

You should avoid using the `pageconfigure` and `delete`. Your users will just feel annoyed if the interface changes all the time and notebook pages appear and disappear every now and then.



(a) Vertical Panes (b) Horizontal Panes

Figure 2.2: The TixPane Widget

2.2 PanedWindow

The *TixPanedWindow* widget arranges arranges its subwidgets in non-overlapping panes. As we can see in figure 2.2, the PanedWindow widget puts a resize handle between the panes for the user to manipulate the sizes of the panes interactively. The panes can be arranged either vertically (figure 2.2(a)) or horizontally (2.2(b)).

Each individual pane may have upper and lower limits of its size. The user changes the sizes of the panes by dragging the resize handle between two panes.

2.2.1 Adding Panes Inside a TixPanedWindow Widget

You can create a TixPanedWindow widget using the `tixPanedWindow` command. After that, you can add panes into this widget using the `add` method (see program 2.3).

When you use the `add` method, there are several optional parameters which you can use to control the size of each of the pane. The `-min` parameter controls the minimum size of the pane and the `-max` parameter controls its maximum size. These two parameters controls how much the user can expand or shrink a pane. If neither is specified, then the pane can be expanded or shrunk without restrictions.

In addition, the -size parameter specifies the initial size of the pane. If it is not specified, then the initial size of the pane will be its natural size.

In program 2.3, we set the initial size of `pane1` to 100 pixels using the -size parameter. We don't set the `-size` parameter for `pane2` so it will appear in its natural size. However, we use the `-max` option for `pane2` so that the user can never expand the size of `pane2` to

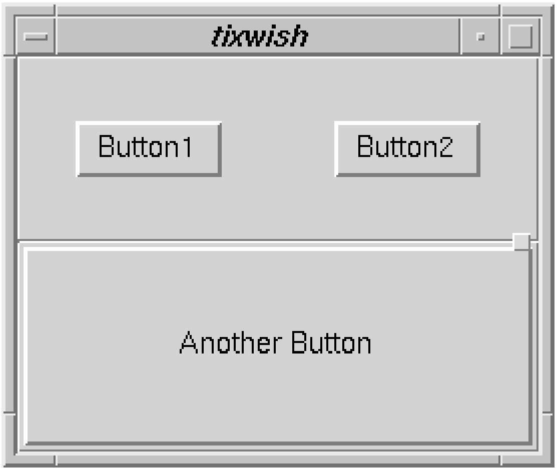


Figure 2.3: Output of Program 2.3

more than 300 pixels.

Program 2.3 Adding Panes into a TixPanedWindow Widget

```
tixPanedWindow .p
.p add pane1 -size 100
.p add pane2 -max 300

set p1 [.p subwidget pane1]
button $p1.b1 -text Button1
button $p1.b2 -text Button2
pack $p1.b1 $p1.b2 -side left -expand yes

set p2 [.p subwidget pane2]
button $p2.b -text "Another Button"
pack $p2.b -side left -expand yes -fill both

pack .p -expand yes -fill both
```

2.2.2 Putting Widgets Inside the Panes

Each pane we have created using the `add` method is essentially a frame widget. After we have created the panes, we can put widgets inside them. As shown inside program 2.3, we can use the `subwidget` method to find out the name of the pane subwidgets. Then we can just create new widgets as their children and pack these new widgets inside the panes. The output of program 2.3 is shown in figure 2.3

2.2.3 Setting the Order of the Panes

Usually, when you create a new pane, it is always added to the bottom or right of the list of panes. If you want to control the order in which the panes appear inside the TixPanedWindow widget, you can use the two optional parameters, `-before` and `-after`, for the `add` method. For example, the call:

```
.p add pane2 -after pane1
```

will place the new pane immediately after `pane1`. The call:

```
.p add pane2 -before pane1
```

will place the new pane immediately in front of `pane1`.

2.2.4 Changing the Sizes of the Panes

If you want to change the sizes of the existing panes or change their maximum/minimum size constraints, you can use the `paneconfigure` method. For example, the following code changes the size of `pane2` to 100 pixels and adjusts its minimum size constraint to no less than 10 pixels:

```
.p paneconfigure pane2 -size 100 -min 10
```

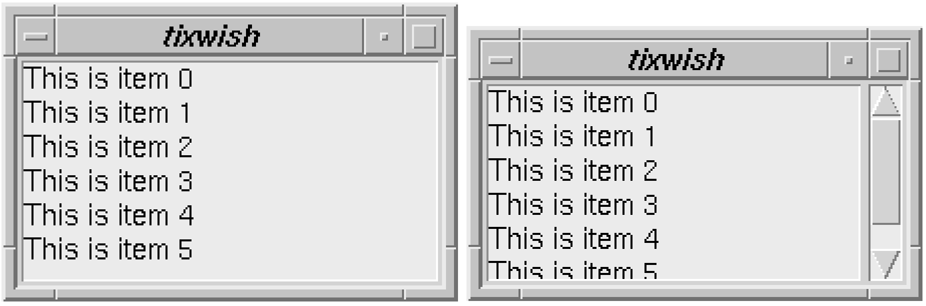
Notice that after you call the `paneconfigure` method, the PanedWindow may jitter and that may annoy the user. Therefore, use this method only when it is necessary.

2.3 The Family of Scrolled Widgets

With plain Tcl/Tk, the widgets do not automatically come with scrollbars. If you want to use scrollbars with the text, canvas or listbox widgets, you will need to create scrollbars separately and attach them to the widgets. This can be a lot of hassle because you would almost always need scrollbars for these widgets. Sometimes you will wonder why you need to write the same boring code again and again just to get the scrollbars to working.

The Tix scrolled widgets are here to make your life easier. With a single command such as `tixScrolledListBox` or `tixScrolledText`, you can create a listbox or text widget that comes automatically with scrollbars attached.

Another advantage of the Tix scrolled widgets is that you can specify their scrolling policy so that the scrollbars appear only when they are needed. This feature is especially useful if you are displaying a lot of widgets and running out of screen real estate.



(a) Scrollbars not displayed

(b) Scrollbars displayed only when needed

Figure 2.4: Scrolled ListBox with Automatic Scrollbars

2.3.1 The Scrolled Listbox Widget

You can create a scrolled listbox widget using the `tixScrolledListBox` command. Notice that the widget created by the `tixScrolledListBox` command is not itself a listbox widget. Rather, it is a frame widget which contains two scrollbar subwidgets: one is called `hsb` (the horizontal scrollbar) and the other is called `vsb` (the vertical scrollbar). Similarly, the listbox being scrolled is also a subwidget which is appropriately called `listbox`. Therefore, if we need to put things into the listbox (as we always do!), we can use the `subwidget` method. As shown in program 2.4, we first find the pathname of the `listbox` subwidget by calling "`.sl subwidget listbox`". Then, we insert some items into the `listbox` subwidget.

Program 2.4 Scrolled Listbox Widget

```
tixScrolledListBox .sl -scrollbar auto
set listbox [.sl subwidget listbox]

for {set x 0} {$x < 6} {incr x} {
    $listbox insert end "This is item $x"
}

pack .sl -side left -expand yes -fill both
```

Also, as seen in the first line of program 2.4, we use the `-scrollbar` option to control the scrolling policy of the TixScrolledListBox widget. Usually, we'll set it to "`auto`": the scrollbars are displayed only if they are needed. Other possible values are "`both`": the two scrollbars are always displayed; "`x`": the horizontal scrollbar is always displayed, while the vertical scrollbar is always hidden; "`y`": the opposite of "`x`"; "`none`": the two scrollbars are always hidden. The result of program 2.4 is shown in figure 2.4.

2.3.2 Other Scrolled Widgets

The **TixScrolledText** widget is very similar to the TixScrolledListBox widget, except it scrolls a text subwidget, which is called `text`. One problem with the TixScrolledText widget, though, is its `-scrollbar` option doesn't work in the **auto** mode. This is due to a bug in Tk which doesn't report the width of the **text** subwidget correctly. Until this bug is fixed in TK, the **auto** mode will behave the same way as the both mode for the TixScrolledText widget.

Another scrolled-widget is **TixScrolledWindow**. Sometimes you have a large number of widgets that can't possibly be shown in the screen all at once and your application doesn't allow you to divide the widgets into several pages of a TixNoteBook. In this case you can use TixScrolledWindow. It contains a frame subwidget called `window`. You can just create as many widgets as you need as children of the `window` subwidget. An example is shown in program 2.5, which uses the TixScrolledWindow widget to implement a "cheap" spreadsheet application. The boxes of the spreadsheet are just entry widgets and they are packed inside the `window` subwidget. The user will be able to scroll to different parts of the spreadsheet if it is too large to fit in one screen.

Program 2.5 Cheap Spreadsheet Application with TixScrolledWindow

```
tixScrolledWindow .sw -scrollbar auto
set f [.sw subwidget window]

for {set x 0} {$x < 10} {incr x} {
    frame $f.f$x
    pack  $f.f$x -side top -expand yes -fill both
    for {set y 0} {$y < 10} {incr y} {
        entry $f.f$x.e$y -width 10
        pack $f.f$x.e$y -side left -fill x
    }
}

pack .sw -side left -expand yes -fill both
```

There are two more scrolled-widgets in the Tix library: **TixScrolledTList** scrolls a TixTList widget and **TixScrolledHList** scrolls a TixHList widget. The subwidgets that they scroll are called `tlist` and `hlist`, respectively. The use of the TList and HList widgets will be described in the next chapters.

Chapter 3

Tabular Listbox and Display Items

3.1 tixTList – The Tix Tabular Listbox Widget

TixTList is the Tabular Listbox Widget. It displays a list of items in a tabular format. For example the TixTList widget in figure 3.1 displays files in a directory in rows and columns.

TixTList does all what the standard Tk listbox widget can do, i.e, it displays a list of items. However, TixTList is superior to the listbox widget is many respects. First, TixTList allows you to display the items in a two dimensional format. This way, you can display more items at a time. Usually, the user can locate the desired items much faster in a two dimensional list than the one dimensional list displayed by the Tk listbox widget.

In addition, while the Tk listbox widget can only display text items, the TixTList widget can display a multitude of types of items: text, images and widgets. Also, while you can use only one font and one color in a listbox widget, you can use many different fonts and colors

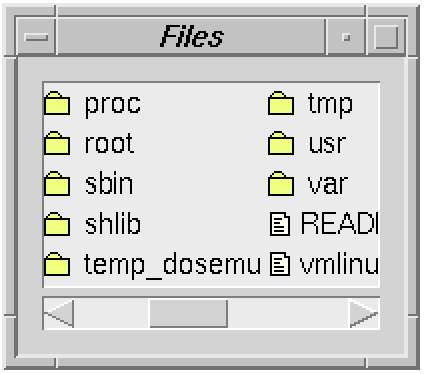


Figure 3.1: Files Displayed in a TixTList Widget in a Tabular Format

Figure 3.2: Employee Names Displayed in a TixTList Widget

in a TixTList widget. In figure 3.1, we use graphical images inside a tixTList widget to represent file objects. In figure 3.2, we display the names of all employees of a hypothetical company. Notice the use of a bold font to highlight all employees whose first name is Joe.

3.2 Display Items

Before we rush to discuss how to create the items inside a TixTList widget, let's first spend some time on a very important topic about the Tix library: the repationship between the display items and their host widgets.

We can better define the terms by taking a quick preview of the TixHList widget, which will be covered in details in the next chapter. Let's compare the items displayed on the two widgets in figure 3.3. If we take a close look at the item that shows the `usr` directory in the TixTList widget on the left versus the TixHList widget on the right, we can see that this item appears exactly the same on both widgets.

If fact, all the items in these two widgets are of the *same* type: they all display an image next to a textual name. The only difference between these two widgets is how these items are arranged. The TixTList widget arranges the items in rows and columns, while the TixHList widget arranges the items in a hierachical format.

With this observation in mind, we can see a separation of tasks berween the widgets and the items they display. We call the TixHList and TixTList widgets in figure 3.3 *host widgets*: their task is to arrange the items according to their particular rules. However, they don't really care what these items display; they just treat the items as rectangle boxes. In contrast, these items, which are called *display items* in Tix terminology, controls the visual information they display, such as the images, text strings, colors, fonts, etc. However, they don't really care where on the host widget they will appear.

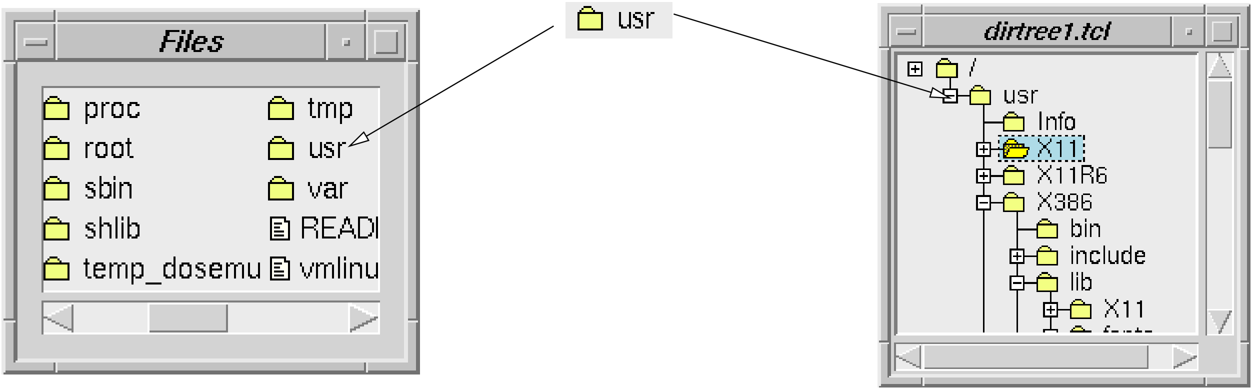


Figure 3.3: The Same Type of Items Displayed in a TixTList (left) and a TixHList(right)

3.2.1 Advantages of Display Items

It is easy to see the advantages of seperating the display items from their host widgets. First, the display items are easy to learn. Since they are the same across different types of widgets. Once you learn about a type of display items, you will know how to use them in all Tix widgets that support display items (currently these include TixHList, TixTList and the spreadsheet widget TixGrid, but the number is growing). In contrast, if you want to create a text item for the Tk widgets, you will find out that the listbox, text, canvas and entry widget each have a different method of creating and manipulating text items, and it is quite annoying to learn each of them individually.

Second, the hosts widgets that use display items are extensible. Because of the separation of task, the host widgets are not involved in the implementation details of the display items. Therefore, if you add a new type of display items, such as a `animation` type that displays live video, the host widgets will gladly take them in and display them. You don't need to modify the existing host widgets at all. In contrast, if you want to display graphical images in the existing Tk listbox widgets, you'd better set aside 100 hours to rewrite it completely!

Third, display items are good for writers of host widgets. Because now they just need to implement the arrangement policy of the host widgets. They don't need to worry about drawing at all because it is all handled by the display items. This is a significant saving in code because a widget that does not use display items has to spend 30% of its C code to do the drawing.

3.2.2 Display Items and Display Styles

The appearance of a display item is controlled by a set of attributes. For example, the `text` attribute controls the text string displayed on the item and the `font` attribute specifies what font should be used.

Usually, each of the attributes falls into one of two categroies: "*individual*" or "*collective*". For example, each of the items inside a TixTList widget may display a different text string; therefore we call the text string an *individual attribute*. However, in most cases, the items

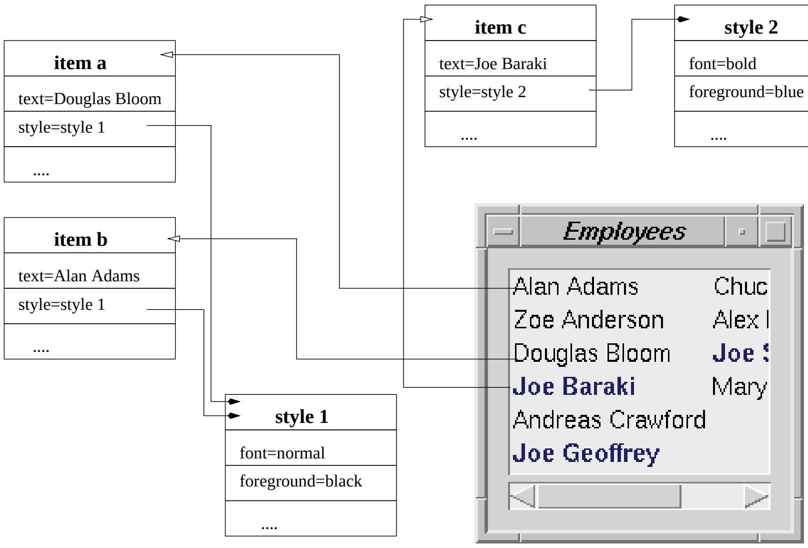


Figure 3.4: Relationship Between Display Items and Display Styles

share the same color, font and spacing and we call these *collective attributes*.

One question concerns where we keep the collective attribute for the display items. Certainly, we can keep a **font** attribute for each item, but this is not really an efficient solution. In fact, if all the items have the same font, we would be keeping a duplicated copy of the same font for each of the items we create. Since a host widget may have many thousands of items, keeping thousands of dupilcated copys of the same font, or any other collective attributes, would be very wasteful.

To advoid the unnecessary duplication of resources, Tix stores the collective attributes in special objects called *display styles*. The relationship between display items and their styles is depicted in figure 3.4. Each item holds its own copy of the individual attributes, such as **text**. However, the collective attributes are stored in the style objects. Each item has a special **style** attribute that tells it which style it should use. In figure 3.4, since items *a* and *b* are assigned the same style, therefore, they share the same font and color. Item *c* is assigned a different style, thus, it uses a different font than *a* and *b*.

3.3 Creating Display Items in the TixTList Widget

3.3.1 Creating Display Items

Now it's time to put our knowledge about host widgets, display items and display styles into practice. The following example code creates two items in a TixTList widget using the **insert** method:

```
tixTList .t
pack .t
```

```
.t insert end -itemtype text -text "First Item"  -underline 0
.t insert end -itemtype text -text "Second Item" -underline 0

set picture [image create bitmap -file picture.xbm]
.t insert end -itemtype image -image $picture
```

As we can see, the `insert` method of TixTList is very similar to the `insert` method of the standard Tk listbox widget: it inserts a new item into the TixTList widget. The first argument it takes is the location of the new item. For example `0` indicates the first location in the list, `1` indicates the second location, and so on. Also the special keyword `end` indicates the end of the list.

Then, we can use the `-itemtype` switch to specify the type of display item we want to create. There are currently four types of items to choose from: `text`, `image`, `imagetext` and `window`. In the above example, we create two items of the type `text` and one item of the type `image`. The subsequent arguments to the `insert` method set the configuration options of the individual attributes of the new item. The available options for these items are listed in figures 3.7 through 3.10.

3.3.2 Setting the Styles of the Display Items

Note that in the above example, if we want to control the foreground color of the text items, we cannot issue commands such as:

```
.t insert end -itemtype text -text "First Item" -foreground black
```

because `-foreground` is not an individual attribute of the text item. Instead, it is a collective attribute and must be accessed using a display style object. To do that we can use the command `tixItemStyle` to create display styles, as shown in the following example:

```
set style1 [tixDisplayStyle text -font 8x13]
set style2 [tixDisplayStyle text -font 8x13bold]

tixTList .t; pack .t

.t insert end -itemtype text -text "First Item"  -underline 0 \
    -style $style1
.t insert end -itemtype text -text "Second Item" -underline 0  \
    -style $style2
.t insert end -itemtype text -text "Third Item"  -underline 0  \
    -style $style1
```

The first argument of `tixDisplayStyle` specify the type of style we want to create. Each type of display item needs its own type of display styles. Therefore, for example, we cannot create a style of type `text` and assign it to an item of type `image`. The subsequent arguments to `tixDisplayStyle` set the configuration options of the collective attributes defined by this

(a) Three `text` items in a TixTList

(b) The `text` items with fonts switched

Figure 3.5: Two Display Styles With Different Fonts

style. A complete list of the configuration options of each type of the display style is in figures 3.11 through 3.13.

The `tixDisplayStyle` command returns the names of the newly created styles to us and we use the variables `style1` and `style2` to store these names. We can then assign the styles to the display items by using the names of the styles. As shown in figure 3.5(a), by assing these two styles to the `-style` option of the display items, we assigned a medium-weight font to the first and third item and a bold font to the second item.

The name of the style returned by `tixDisplayStyle` is also the name of a command which we can use to control the style. For example, we can use the following commands to switch the fonts in the two styles we created in the above example:

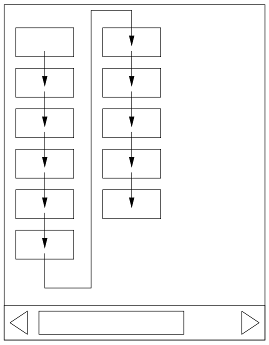
```
$style1 configure -font 8x13bold
$style2 configure -font 8x13
```

After the execution of the above command, the font in the second item in the TixTList widget becomes medium-weight and the font in the first and third items becomes bold, as shown in figure 3.5(b).

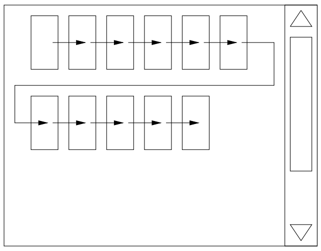
3.3.3 Configuring and Deleting the Items

You can configure the individual attributes of the items using the `entryconfigure` method. There is also the entrycget method for querying the attributes of the items. To `delete` the items, you can use the delete method. In the following example, we use these two methods to change the first and third items to display the text strings `One` and `Two` and change the third item to use the style `$style2`. Then we delete the second item using the `delete` command.

```
.t entryconfigure 0 -text One
.t entryconfigure 2 -text Two
.t delete 1
```



(a) Vertical Orientation



(b) Horizontal Orientation

Figure 3.6: The `-orientation` option of the TixSelect Widget

3.3.4 Choosing the Orientation and Number of Rows or Columns

There are three options that controls the layout of the items in the TixTList widget. The `-orientation` option can be set to either vertical or horizontal. When `-orientation` is set to `vertical`, the items are laid out vertically from top down and wrapped to the next column when the bottom is reached (see figure 3.6(a)). The opposite layout policy is chosen if `-orientation` is set to `horizontal` (see figure 3.6(b)).

When the `-orientation` option is set to `vertical`, normally the number of columns displayed depends on the number of items in the TixTList widget: the more items there are, the more columns will there be. However, we can use the `-columns` option to control the number of columns: the items will be wrapped in a way so that the number of columns produced will be exactly as dicated by the `-columns` option.

One use of the `-columns` option is to specify the same layout policy as that of the standard Tk listbox widget. We can do this by setting `-orientation` to vertical and `-columns` to `1`. This way we can get a replacement listbox widget that can display multiple fonts and colors and graphics!

The counterpart of the `-columns` option is the `-rows` option, which is used for the same purpose when the `-orientation` option is set to `horizontal`.

3.3.5 Event Handling

You can handle the events in a TList widget using the `-browsecmd` and `-command` options. The meanings of these two options are silimar to their meanings in other Tix widgets such as the ComboBox. Usually, the command specified by `-browsecmd` is called when the user clicks or drags the mouse over the items or presses the arrow keys. The command specified by `-command` is called when the user double-clicks or presses the Return key. These commands are called with one extra argument — the index of the currently “active” item, which is usually the item under the mouse cursor.

3.3.6 Selection

The `-selectmode` option controls how many items the user can select at one time. In the `single` and `browse` mode, the user can select only one item at a time. In the `multiple` and `extended` mode, the user can select multiple items; the `extended` mode allows disjoint selections while the `multiple` mode does not.

Normally, the user selects the items using the mouse or the keyboard. You can find out which items the user has selected with the `info selection` method, which returns a list of the currently selected items. You can also set the selection using the `selection set` method. For example, the command `.tlist selection set 3` selects the item whose index is `3`. The command `.tlist selection set 2 10` selects all the items at index `2` through `10`. The method `selection clear` empties the selection.

Option	Meaning
`-bitmap`	Specifies the bitmap to display in the item.
`-image`	Specifies the image to display in the item. When both the `-bitmap` and `-image` options are specified, only the image will be displayed.
`-style`	Specifies the display style to use for this item.
`-showimage`	A Boolean value that specifies whether the image/bitmap should be displayed.
`-showtext`	A Boolean value that specifies whether the text string should be displayed.
`-text`	Specifies the text string to display in the item.
`-underline`	Specifies the integer index of a character to underline in the text string in the item. 0 corresponds to the first character of the text displayed in the widget, 1 to the next character, and so on.

Figure 3.7: Individual Attributes for the `imagetext` Display Item

Option	Meaning
`-style`	Specifies the display style to use for this item.
`-text`	Specifies the text string to display in the item.
`-underline`	Specifies the integer index of a character to underline in the text string in the item. 0 corresponds to the first character of the text displayed in the widget, 1 to the next character, and so on.

Figure 3.8: Individual Attributes for the `text` Display Item

Option	Meaning
`-style`	Specifies the display style to use for this item.
`-image`	Specifies the image to display in the item.

Figure 3.9: Individual Attributes for the `image` Display Item

Option	Meaning
`-style`	Specifies the display style to use for this item.
`-window`	Specifies the widget to display in the item.

Figure 3.10: Individual Attributes for the `window` Display Item

`-activebackground`	`-activeforeground`	`-anchor`
`-background`	`-disabledbackground`	`-disabledforeground`
`-foreground`	`-font`	`-justify`
`-padx`	`-pady`	`-selectbackground`
`-selectforeground`	`-wraplength`	

Figure 3.11: Collective Attributes for the `imagetext` and `text` Display Items

`-anchor`	`-padx`	`-pady`

Figure 3.12: Collective Attributes for the `window` Display Item

`-activebackground`	`-activeforeground`	`-anchor`
`-background`	`-disabledbackground`	`-disabledforeground`
`-foreground`	`-padx`	`-pady`
`-selectbackground`	`-selectforeground`	

Figure 3.13: Collective Attributes for the `image` Display Item

Chapter 4

Hierarchical Listbox

4.1 TixHList – The Tix Hierarchical Listbox Widget

TixHList is the Tix Hierarchical Listbox Widget. You can use it to display any data that have a hierarchical structure. For example, the HList widget in figure 4.1(a) displays a Unix file system directory tree; the HList widget in figure 4.1(b) displays the corporate hierarchy of a hypothetical company. As shown in these two figures, the entries inside the TixHList widget are indented can be optionally connected by branch lines according to their positions in the hierarchy.

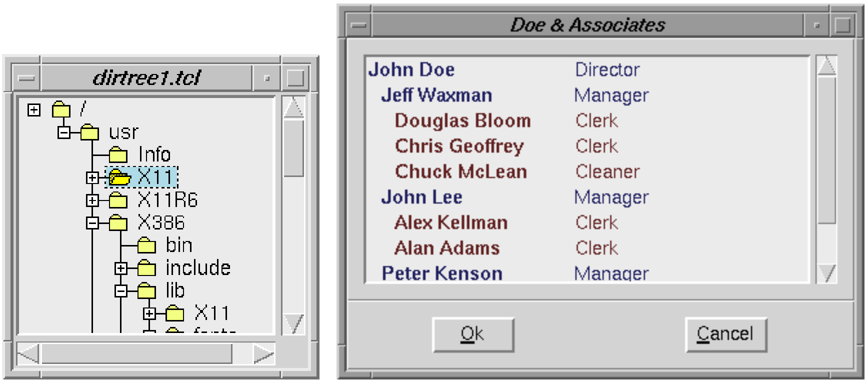
4.1.1 Creating a Hierarchical List

A TixHList widget can be created by the command `tixHList`. However, most likely, you would want to create a TixHList with scrollbars attached. Therefore, usually you will use the `tixScrolledHList` command to create a scrolled hierarchical listbox (line 1 in program 4.1). The `tixScrolledHList` command is very similar to the `TixScrolledListBox` command we saw in section 2.3.1. It creates a TixHList subwidget of the name `hlist` and attaches two scrollbars to it.

As shown in the first five lines in program 4.1, we create a scrolled TixHList widget, using the `-options` switch (see section 1.3.5) to set several options for the `hlist` subwidget (we'll talk about these options shortly). Then, we can access the HList subwidget widget using the `subwidget hlist` method (line 7 in program 4.1).

4.1.2 Creating Entries in a HList Widget

Each entry in an HList widget has a unique name, called its *entry-path*, which determines each entry's position in the HList widget. The entry-paths of the HList entries are very similar to the pathnames of Unix files. Each entry-path is a list of string names separated by a *separator character*. By default, the separator character is the period character (`.`), but it can be configured using the `-separator` option of the HList widget.



(a) Directory Tree Display

(b) A Corporate Hierarchy

Figure 4.1: Examples of the TixHList Widget

Program 4.1 Creating Entries in a HList Widget

```
tixScrolledHList .sh -options {
    hlist.itemType text
    hlist.drawBranch false
    hlist.indent     8
}
pack .sh -expand yes -fill both
set hlist [.sh subwidget hlist]

$hlist add foo         -text "foo"
$hlist add foo.bar     -text "foo's 1st son"
$hlist add foo.bor     -text "foo's 2nd son"
$hlist add foo.bar.bao -text "foo's 1st son's 1st son"
$hlist add foo.bar.kao -text "foo's 1st son's 2nd son"
$hlist add dor         -text "dor, who has no son"
```

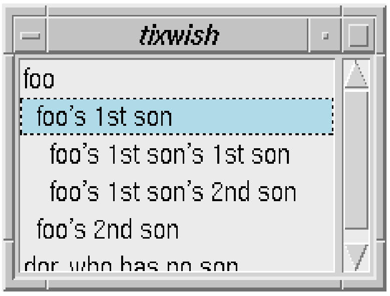


Figure 4.2: Output of Program 4.1

In program 4.2, we add several new entries `foo`, `foo.bar`, `foo.bor`, `foo.bar.bao`, .. etc, into the HList widget using the `add` method. The relationship between the entries is signified by their names, in a way similar to how Unix denotes directories and subdirectories. For example, `foo` is the *parent* of `foo.bar` and `foo.bor`; `foo.bar` is the parent of `foo.bar.bao`, and so on. As far as the terminology goes, we also say that `foo.bar` a *child* of `foo`; `foo` is an *ancestor* of `foo.bar.bao` and `foo.bar.bao` is a *descendant* of `foo`.

The output of program 4.1 is shown in figure 4.2. As we can see, the entries are displayed under their parents with the amount of indentation control by the `-indent` option of the HList widget: `foo.bar.bao` and `foo.bar.kao` are display under `foo.bar`, which is in turn displayed under `foo`.

Entries with no parents, for example, `foo` and `dor` in program 4.1, are called *top-level entries*. Top-level entries are usually entries with no immediate superiors in a hierarchical. For example, the owner of a company, the principle of a school or the root directory of a Unix file system. Toplevel entries are displayed with no indentation.

As evident from program 4.1, all entries who entry-path does not contain a separator character are top-level entries. The only exception is the separator character itself is also a toplevel entry. This makes it easy to display Unix file and directory names inside the HList widget, as shown in program 4.2.

Program 4.2 Displaying Directories in a HList Widget

```
set folder [tix getimage folder]
tixScrolledHList .sh -options {
    hlist.separator     /
    hlist.itemType      imagetext
    hlist.drawBranch    true
    hlist.indent        14
    hlist.wideSelection false
}
pack .sh -expand yes -fill both
set hlist [.sh subwidget hlist]

foreach directory {/ /usr /usr/bin /usr/local /etc /etc/rc.d} {
    $hlist add $directory -image $folder -text $directory
}
```

Each entry is associated with a display item (see section 3.2 about display items). We can use the `-itemtype` option of the HList widget to specify the default type of display item to be created by the the `add` method, as shown in program 4.1 and 4.2. Alternatively, we can also specify the type of display item using the `-itemtype` option for the `add` method.

4.1.3 Controlling the Layout of the Entries

There are two options to control the layout of the entries: the `-showbranch` option specifies whether branch lines should be drawn between parent entries and their children. The `-indent` option controls the amount of relative indentation between parent and child entries.

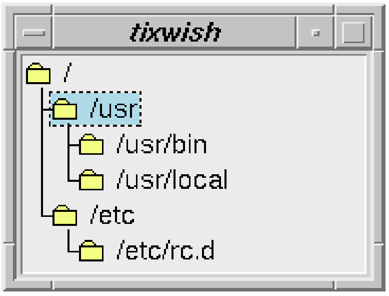


Figure 4.3: Output of Program 4.2

Notice the `-drawbranch` option is turned on in figure 4.3 but turned off in figure 4.2. Usually, you need to set a bigger indentation when the branches are shown — we used an indentation of 14 pixels in 4.3 compared to 8 pixels in 4.2.

4.1.4 Handling the Selection and User Event

The handling of the selection and user events for the HList widget is very similar to the TList widget (see section 3.3.5), except that for the HList widget all the operations are based on entry-paths, not list indices. The methods `info selection`, `selection set` and `selection clear` can be used to query, set or clear the selection; the option `-selectmode` controls how many entries can be selected at a time; the options `-browsecmd` and `-command` can be used to specify a command to be called to handle user events.

There is one more option worth mentioning: the `-wideselection` option. When set to `true`, the selection highlight will be drawn across the whole HList widget (see figure 4.2). When set to false, selection highlight will be drawn as wide as the selected entry (see figure 4.3). Normally, you would set `-wideselection` to `false` when you use `imagetext` items inside (as we did in program 4.2).

4.2 Creating Collapsible Tree Structures with TixTree

The TixTree widget is based on the TixScrolledHList widget; you can use it to create a collapsible hierarchical structure so that the user can conveniently navigate through a large number of list entries. As shown in figure 4.4, the TixTree puts the little "`+`" and "`-`" icons next to the branches of an HList entry that has descendants. These two icons are knows as the open and close icons, respectively. When the user presses the open icon next to an entry, its immediate children of an entry will be displayed. Conversely, when the user presses the close icon, the entry's children will become hidden.

Program 4.3 shows how to create a collapsible tree. We first create a TixTree widget. Then we add the entries in your hierarchical structure into its `hlist` subwidget using the add method of this subwidget. When we are finished with adding the entries, we just call the `autosetmode` method of the TixTree widget, which will automatically adds the open and close icons next to the entries who have children.

Program 4.3 Creating a Collapsible Hierarchy

```
set folder [tix getimage folder]
tixTree .tree -command Command -options {
    hlist.separator  /
    hlist.itemType   imagetext
    hlist.drawBranch true
    hlist.indent     18
}
pack .tree -expand yes -fill both
set hlist [.tree subwidget hlist]

foreach directory {/ /usr /usr/bin /usr/local /etc /etc/rc.d} {
    $hlist add $directory -image $folder -text $directory
}
.tree autosetmode

proc Command {entry} {
    puts "you have selected $entry"
}
```

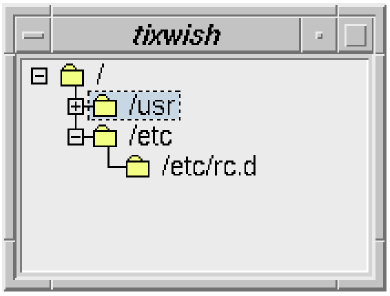


Figure 4.4: Output of Program 4.3

Note that in program 4.3 we use the `-command` option of the TixTree widget, not the `-command` option of its `hlist` subwidget. This is because the TixTree actually used the `-command` option of its `hlist` subwidget to process some low-level events. In general, if both a mega-widget and its subwidget have the options of the same name, you would always use the option that belongs to the mega-widget.

Chapter 5

Selecting Files and Directories

One task that an application has to perform frequently is to ask the user to select files or directories. To select files, you can use the Tix *File Selection Widgets*: TixFileSelectDialog and TixExFileSelectDialog. To select directories, you can use the Tix *Directory Selection Widgets*: TixDirList and TixDirTree.

5.1 File Selection Dialog Widgets

There are two file dialog widgets inside Tix: the TixFileSelectDialog (figure 5.1) is similar to the FileSelectionDialog widget in Motif; TixExFileSelectDialog (figure 5.2) looks like its conunterpart in MS Windows. Both widgets let the user navigate through the file system directories and select a file.

One advanced feature of both types of file selection boxes is they use ComboBoxes to store the files, directories and patterns the user has selected in the past. If the user wants to select the same files again, he can simply open the ComboBoxes and click on his past inputs. This saves a lot of keystrokes and is especially useful when the user needs to switch among several files or directories.

5.1.1 Using the TixFileSelectDialog Widget

An example of using the TixFileSelectDialog widget is in figure 5.1. At line 1, we create a TixFileSelectDialog widget and set the title of the dialog to "Select A File" using the `-title` option. We also use the `-command` option to specify that the procedure `selectCmd` should be called when the user has selected a file. `selectCmd` will be called with one parameter, the filename selected by the user. When the TixFileSelectDialog widget is created, it is initially not shown on the screen. Therefore, at line 3, we call its `popup` widget command to place the widget on the screen.

Program 5.1 Using the TixFileSelectDialog

```
tixFileSelectDialog .file -title "Select A File" -command selectCmd
.file subwidget fsbox config -pattern "*.txt" -directory /usr/info
.file popup

proc selectCmd {filename} {
    puts "You have selected $filename"
}
```

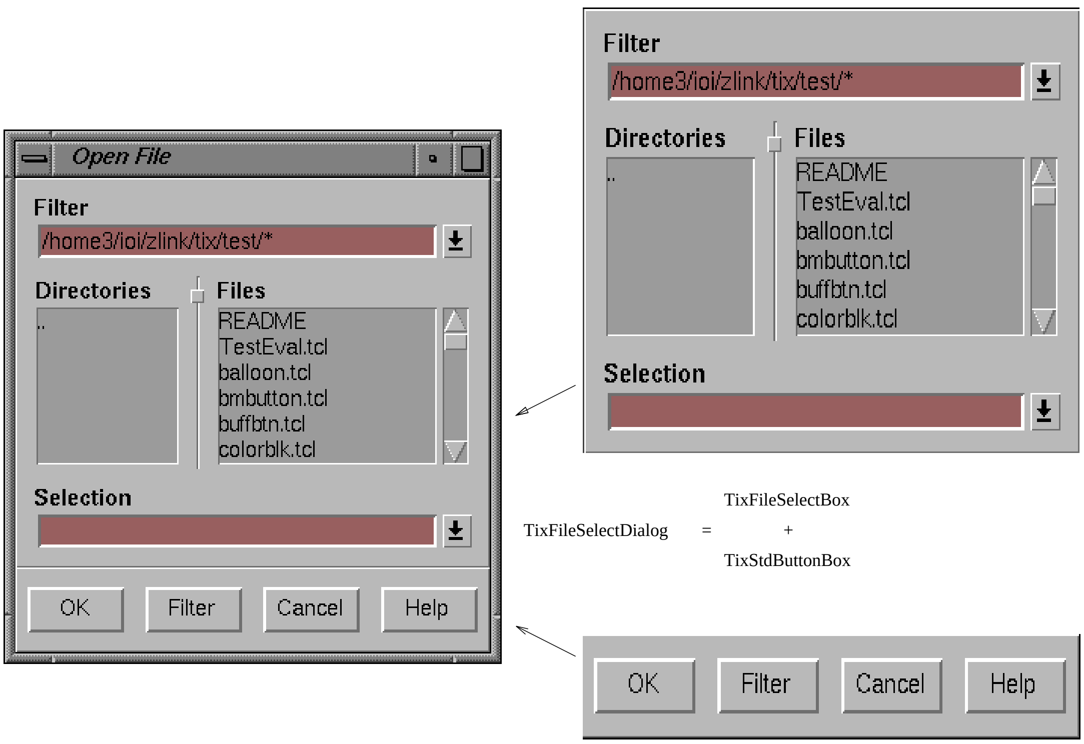


Figure 5.1: The Composition of a TixFileSelectDialog Widget

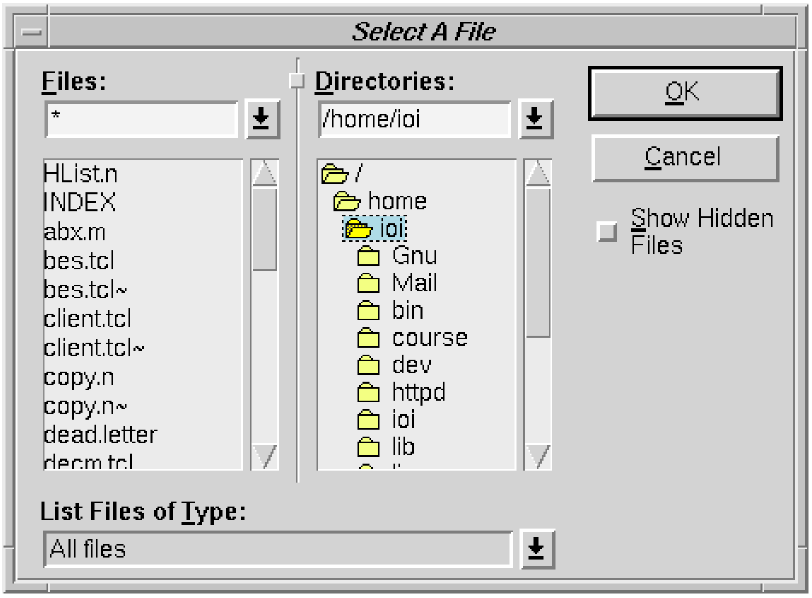


Figure 5.2: The ExFileSelectDialog Widget

5.1.2 The Subwidget in the TixFileSelectDialog

We may also want to set other options for the file dialog such as its file filter and working directory. To do this, we must know the composition of the TixFileSelectDialog widget. As shown in figure 5.1, the TixFileSelectDialog contains a subwidget `fsbox` of the type TixFileSelectBox and a subwidget `bbox` of the type TixStdButtonBox.

The `fsbox` subwidget supports the `-pattern` and `-directory` options. At line 2 of figure 5.1, we use the `-directory` option to tell the `fsbox` subwidget to display files in the directory `/usr/info`; we also use the `-pattern` option to specify we only want the filenames that has the `txt` extension.

The `fsbox` subwidget also supports the `-selection` option, which stores the filename currently selected by the user. We can query this value by the `cget` widget command of the `fsbox` subwidget.

Remember that the `-pattern`, `-directory` and `-selection` options do not belong to the TixFileSelectDialog widget. A common mistake that people make is to try to configure the non-existent `-pattern` option of the TixFileSelectDialog, which causes much despair, long error messages and great loss of self-confidence. *Always remember:*, when you want to configure an option, find out whether it belongs to the widget or its subwidgets.

5.1.3 The TixExFileSelectDialog Widget

The TixExFileSelectDialog widget is very similar to the TixFileSelectDialog widget. It supports all the options and widget commands of the latter, so essentially we can just take the program 5.1 and replace the command `tixFileSelectDialog` in the first line to `tixExFileSelectDialog`.

The composition of the TixExFileSelectDialog widget is a bit different: it contains one contains one subwidget, which is also called `fsbox`, of the type TixExFileSelectBox widget (figure 5.2). Again this `fsbox` widgets supports all widget options and commands of the `fsbox` subwidget in TixFileSelectDialog, so the line 2 of program 5.1 can work for TixExFileSelectDialog widgets without any change.

5.1.4 Specifying File Types for TixExFileSelectDialog

The TixExFileSelectBox widget has a ComboBox subwidget marked as "Select Files of Type:" (see figure 5.2). This widget contains some pre-set types of files for the user to choose from. For example, a word processor program can include choices such as "Microsoft Word Documents" and "WordPerfect Documents".

The TixExFileSelectBox widget has a `-filetypes` option for this purpose. As shown in line 3 through 7 in program 5.2, the value for the `-filetypes` option is a list. Each item in the list should contain two parts. The first part is a list of file patterns and the second part is the textual name for this type of files.

5.1.5 The `tix filedialog` Command

TixExFileSelectDialog and TixFileSelectDialog are very similar to each other. So which one should we use? That is just a matter of taste. However, since we know that programmers usually have bad taste, clever programmers would rather step aside and let the users exercise their own taste. To do this, we can use the `tix filedialog` command.

For any programs based on Tix, the user can choose his preferred type of file dialog by setting the X resource `FileDialog` to either `tixFileSelectDialog` or `tixExFileSelectDialog`. This can usually be done by inserting a line similar to the following into the user's `.Xdefaults` file:

```
*myapp*FileDialog: tixExFileSelectDialog
```

When we call the command `tix filedialog`, it will return a file dialog widget of the user's preferred type.

The advantage of using `tix filedialog` is it makes coding flexible. If the management suddenly mandates that we dump the Motif look-and-feel in favor of the MS Windows look-and-feel, we don't need to dig up every line of `tixFileSelectDialog` calls and replace it with `tixExFileSelectDialog`. Also, `tix filedialog` creates only one copy of the file dialog, which can be shared by different parts of the program. Therefore, we can avoid creating a separate file dialog widget for each of the "Open", "Save" and "Save As" commands in our application. This way, we can save resource since a file dialog is a large widget and it takes up quite a bit of space.

The use of the `tix filedialog` command is shown in program 5.2. This program is very similar to what we saw in program 5.1, except now we aren't really sure which type of file dialog the user have chosen. Therefore, if we want to do something allowed for only

Program 5.2 Using the `tix dialog` command

```
set dialog [tix filedialog]
$dialog -title "Select A File" -command selectCmd
$dialog subwidget fsbox config -pattern "*.txt" -directory /usr/info
if {[winfo class $dialog] == "TixExFileSelectDialog"} {
    $dialog subwidget fsbox config -filetypes {
        {{*}          {*     -- All files}}
        {{*.txt}      {*.txt -- Text files}}
        {{*.c}        {*.c   -- C source files}}
    }
}
$dialog popup

proc selectCmd {filename} {
    puts "You have selected $filename"
}
```

one type of file dialogs, we have to be careful. At line 4 of program 5.2, we use the `winfo` command to see whether the type of the file dialog is TixExFileSelectDialog. If so, we set the value for the `-filetypes` option of its `fsbox` subwidget.

5.2 Selecting Directories with the TixDirTree and TixDirList Widgets

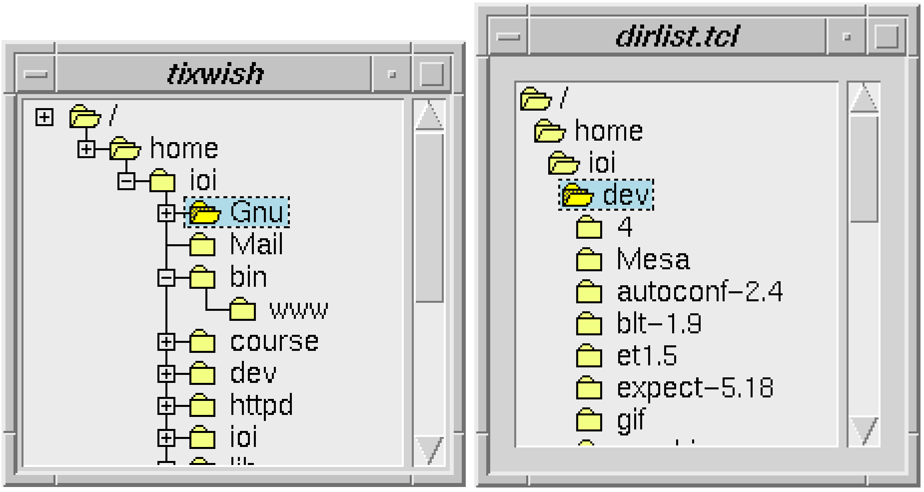
There are two Tix widgets for selecting a directory: TixDirList (figure 5.3(a)) and TixDirTree (figure 5.3(b)). Both of them display the directories in a hierarchical format. The display in the TixDirList widget is more compact: it shows only the parent- and child-directories of a particular directory. The TixDirTree widget, on the other hand, can display the whole tree structure of the file system.

The programming interface of these two widgets are the same and you can choose the which one to use depending on your application. As shown in the following example, you can use the `-directory` option of the TixDirList widget to specify a directory to display. In the example, we set `-directory` to be `/home/ioi/dev`. As a result, the TixDirList widget displays all the subdirectories and all the ancestor directories of `/home/ioi/dev`. You can use the `-command` and `-browsecmd` options to handle the user events: a double click or Return key-stroke will trigger the `-command` option and a single click or space bar key stroke will trigger the `-browsecmd` option. Normally, you would handle both type of events in the same manner, as we have done in program 5.3

Program 5.3 Using the TixDirList widget

```
tixDirList .d -value /home/ioi/dev \
    -command "selectDir" -browsecmd "selectDir"
pack .d

proc selectDir {dir} {
    puts "now you select $dir"
}
```



(a) DirTree (b) DirList

Figure 5.3: The DirTree and DirList Widgets

Chapter 6

Tix Object Oriented Programming

This chapter is intended for experienced programmers who want to create new Tix widgets. If you just want use the Tix widgets in your applications, you can skip this chapter.

6.1 Introduction to Tix Object Oriented Programming

Tix comes with a simple object oriented programming (OOP) framework, the *Tix Intrinsics*, for writing mega-widgets. The Tix Intrinsics is not a general purpose OOP system and it does not support some features found in general purpose OOP systems such as `[incr Tcl]`. However, the Tix Intrinsics is specially designed for writing mega-widgets. It provides a simple and efficient interface for creating mega-widgets so that you can avoid the complexity and overheads of the general purpose OOP extensions to Tcl.

The hard thing about programming with mega-widgets is to make sure that each instance you create can handle its own activities. Events must be directed to the right widget, procedures must act on data that is internal to that widget, and users should be able to change the options associated with the widget. For instance, we'll show an arrow widget that needs to know what direction it's pointing; this requires each instance of the widget to have its own variable.

Furthermore, each widget should respond properly to changes requested by the application programmer during the program's run. The whole reason people use Tcl/Tk is because they can alter things on the fly.

The advantage of an object-oriented programming system is that you can easily associate a widget with its own data and procedures (methods). This chapter shows how to do that, and how to configure data both at the time the widget is initialized and later during the program.

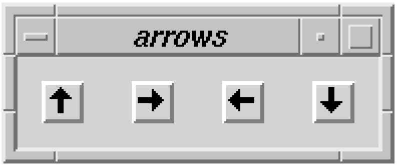


Figure 6.1: Arrow Buttons

6.1.1 Widget Classes and Widget Instances

All the mega-widget classes in Tix, such as TixComboBox and TixControl, are implemented in the Tix Intrinsics framework. Also, you can write new *widget classes* with the Tix Intrinsics. In the next section, I'll go through all the steps of creating a new widget class in Tix. I'll illustrate the idea using a new class "TixArrowButton" as an example. TixArrowButton is essentially a button that can display an arrow in one of the for directions (see figure 6.1).

Once you have defined your classes, you can create *widget instances* of these classes. For example, the following code will create four instances of your new TixArrowButton class:

```
tixArrowButton .up    -direction n
tixArrowButton .left  -direction e
tixArrowButton .right -direction w
tixArrowButton .down  -direction s
```

6.1.2 What is in a Widget Instance

Each widget instance is composed of three integral parts: variables, methods and component widgets

Variables

Each widget instance is associated with a set of variables. In the example of an instance of the TixArrowButton class, we may use a variable to store the direction to which the arrow is pointing to. We may also use a variable to count how many times the user has pressed the button.

Each variable can be public or private. Public variables may be accessed by the application programmer (usually via `configure` or cget `methods`) and their names usually start with a dash (`-`). They usually are used to represent some user-configurable options of the widget instance. Private variables, on the other hand, cannot be accessed by the application programmer. They are usually used to store information about the widget instance that are of interests only to the widget writer.

All the variables of an instance are stored in a global array that has the same name as the instance. For example, the variables of the instance `.up` are stored in the global array `.up:`. The public variable `-direction`, which records the direction to which the arrow is pointing to, is stored in `.up(-direction)`. The private variable `count`, which counts how many times the user has pressed the button, is stored in `.up(count)`. In comparison, the same variables of the `.down` instance are stored in `.down(-direction)` and `.down(count)`.

Methods

To carry out operations on the widget, you define a set of procedures called *methods* (to use common object-oriented terminology). Each method can be declared as public or private. *Public methods* can be called by the application programmer. For example, if the TixArrowButton class supports the public methods `invoke` and `invert`, the application programmer can issue the commands to call these method for the widget instance `.up`.

```
.up invert
.up invoke
```

In contrast, *Private methods* are of interests only to widget writers and cannot be called by application programmers.

Component Widgets

A Tix mega-widget is composed of one or more component widgets. The main part of a mega-widget is called the *root widget*, which is usually a frame widget that encompasses all other component widgets. The other component widgets are called *subwidgets*.

The root widget has the same name as the the mega-widget itself. In the above example, we have a mega-widget called `.up`. It has a root widget which is a frame widget and is also called `.up`. Inside `.up` we have a button subwidget called `.up.button`.

Similar to variables and methods, component widgets are also classified into public and private component widgets. Only public widgets may be accessed by the application programmer, via the `subwidget` method (see section 1.3.1) of each widget instance.

6.2 Widget Class Declaration

The first step of writing a new widget class is to decide the base class from which the new class. Usually, if the new class does not share any common features with other classes, it should be derived from the TixPrimitive class. If it does share common features with other classes, then it should be derived from the appropriate base class. For example, if the new class support scrollbars, it should be derived from TixScrolledWidget; if it displays a label next to its "main area", then it should be derived from TixLabelWidget.

In the case of our new TixArrowButton class, it doesn't really share any common features with other classes, so we decide to use the base class TixPrimitive as its superclass.

6.2.1 Using the tixWidgetClass Command

We can use the `tixWidgetClass` command to declare a new class. The syntax is:

```
tixWidgetClass classCommandName {
    -switch value
    -switch value
    ....
}
```

For example, the following is the declaration section of TixArrowButton:

Program 6.1 declaration of the TixArrowButton Class

```
tixWidgetClass tixArrowButton {
    -classname  TixArrowButton
    -superclass tixPrimitive
    -method {
        flash invoke invert
    }
    -flag {
        -direction -state
    }
    -configspec {
        {-direction direction Direction e}
        {-state state State normal}
    }
    -alias {
        {-dir -direction}
    }
    -default {
        {*Button.anchor         c}
        {*Button.padX           5}
    }
}
```

We'll look at what each option means as I describe the command in the following sections.

The first argument for `tixWidgetClass` is the *command name* for the widget class (`tixArrowButton`). Command names are used to create widgets of this class. For example, the code

```
tixArrowButton .arrow
```

creates a widget instance `.arrow` of the class TixArrowButton. Also, the command name is used as a prefix of all the methods of this class. For example, the `Foo` and `Bar` methods of the class TixArrowButton will be written as `tixArrowButton::Foo` and `tixArrowButton::Bar`.

The *class name* of the class (`TixArrowButton`)is specified by the `-classname` switch inside the main body of the declaration. The class name is used only to specify options in the TK option database. For example, the following commands specifies the TixArrowButton widget instances should have the default value `up` for their `-direction` option and the default value `normal` for their `-state` option.

```
option add *TixArrowButton.direction up
option add *TixArrowButton.state     normal
```

Notice the difference in the capitalization of the class name and the command name of the TixArrowButton class: both of them has the individual words capitalized, but the command name (`tixArrowButton`)starts with a lower case letter while the class name (`TixArrowButton`) starts with an upper case letter. When you create your own classes, you should follow this naming convention.

The `-superclass` switch specifies the superclass of the new widget. In our example, we have set it to `tixPrimitive`. Again, pay attention to the capitalization: we should use the command name of the superclass, not its class name.

6.3 Writing Methods

After we have declared the new widget class, we can write methods for this class to define its behavior. Methods are just a special type of TCL procedures and they are created by the `proc` command. There are, however, three requirements for methods. First, their names must be prefixed by the command name of their class. Second, they must accept at least one argument and the first argument that they accept must be called `w`. Third, the first command executed inside each method must be

```
upvar #0 $w data
```

For example, the following is an implementation of the invert method for the class TixArrowButton:

```
proc tixArrowButton::invert {w} {
    upvar #0 $w data

    set curDirection $data(-direction)
    case $curDirection {
        n {
            set newDirection s
        }
        s {
            set newDirection n
        }
        # ....
    }
}
```

Notice that the name of the method is prefixed by the command name of the class (`tixArrowButton`). Also, the first and only argument that it accepts is `w` and the first line it executes is "`upvar #0 $w data`".

The argument `w` specifies which widget instance this method should act upon. For example, if the user has issued the command

```
.up invert
```

on an instance `.up` of the class tixArrowButton, the method `tixArrowButton::invert` will be called and the argument `w` will have the value `.up`.

The `invert` method is used to invert the direction of the arrow. Therefore, it should examine the variable `.up(-direction)`, which stores the current direction of the instance `.up`, and modify it appropriately. It turns out that in TCL, the only clean way to access an array whose name is stored in a variable is the "`upvar #0 $w data`" technique: essentially it tells the intepreter that the array data should be an alias for the global array whose name is stored in `$w`. We will soon see how the widget's methods use the data array.

Once the mysterious "`upvar #0 $w data`" line is explained, it becomes clear what the rest of the `tixArrowButton::invert` method does: it examines the current direction of the arrow, which is stored in `$data(-direction)` and inverts it.

6.3.1 Declaring Public Methods

All the methods of a class are by default private methods and cannot be accessed by the application programmer. If you want to make a method public, you can include its name in the `-method` section of the class declaration. In our TixArrowButton example, we have declared that the methods `flash`, `invert` and `invoke` are public methods and they can be accessed by the application programmer. All other methods of the TixArrowButton class will be private.

Usually, the names of private methods start with a capital letter with individual words capitalized. The names of public methods start with a lowercase letter.

6.4 Standard Initialization Methods

Each new mega-widget class must supply three standard initialization methods. When an instance of a Tix widget is created, three three methods will be called to initialize this instance. The methods are `InitWidgetRec`, `ConstructWidget` and `SetBindings` and they will be called in that order. The following sections show how these methods can be implemented.

6.4.1 The InitWidgetRec Method

The purpose of the `InitWidgetRec` method is to initialize the variables of the widget instance. For example, the following implementation of `tixArrowButton::InitWidgetRec` sets the `count` variable of each newly created instance to zero.

```
proc tixArrowButton::InitWidgetRec {w} {
    upvar #0 $w data

    set data(count) 0
```

```
}
```

Earlier, we showed how each widget you create is associated with an array of the same name. Within the methods, you always refer to this array through the name `data` –the method then works properly in each instance of the widget.

Chaining Methods

The above implementation is not sufficient because our TixArrowButton class is derived from TixPrimitive. The class derivation in Tix is basically an *is-a* relationship: TixArrowButton *is a* TixPrimitive. TixPrimitive defines the method `tixPrimitive::InitWidgetRec` which sets up the instance variables of every instance of TixPrimitive. Since an instance of TixArrowButton is also an instance of TixPrimitive, we need to make sure that the instance variables defined by TixPrimitive are also properly initialized. The technique of calling a method defined in a superclass is called the *chaining* of a method. The following implementation does this correctly:

```
proc tixArrowButton::InitWidgetRec {w} {
    upvar #0 $w data

    tixPrimitive::InitWidgetRec $w
    set data(count) 0
}
```

Notice that `tixPrimitive::InitWidgetRec` is called before anything else is done. This way, we can define new classes by means of successive refinement: we can first ask the superclass to set up the instance variables, then we can modify some of those variables when necessary and also define new variables.

The tixChainMethod call

The above implementation of `tixArrowButton::InitWidgetRec` is correct but it may be cumbersome if we want to switch superclasses. For example, suppose we want to create a new base class TixArrowWidget, which presumably defines common attributes of any class that have arrows in them. Then, instead of deriving TixArrowButton directly from TixPrimitive, we decide to derive TixArrowButton from TixArrowWidget, which is in turn derived from TixPrimitive:

```
tixWidgetClass tixArrowWidget {
    -superclass tixPrimitive
    ...
}
tixWidgetClass tixArrowButton {
    -superclass tixArrowWidget
    ...
}
```

Now we would need to change all the method chaining calls in TixArrowButton from:

```
tixPrimitive::SomeMethod
```

to:

```
tixArrowWidget::SomeMethod
```

This may be a lot of work because you may have chained methods in many places in the original implementation of TixArrowButton.

The `tixChainMethod` command solves this problem. It will automatically find a superclass that defines the method we want to chain and calls this method for us. For example, the following is a better implementation of `tixArrowButton::InitWidgetRec` that uses `tixChainMethod` to avoid calling `tixPrimitive::InitWidgetRec` directly:

```
proc tixArrowButton::InitWidgetRec {w} {
    upvar #0 $w data

    tixChainMethod $w InitWidgetRec
    set data(count) 0
}
```

Notice the order of the arguments for tixChainMethod: the name of the instance, `$w`, is passed before the method we want to chain, `InitWidgetRec`. In general, if the method we want to chain has $1 + n$ arguments:

```
proc tixPrimitive::MethodToChain {w arg1 arg2 ... argn} {
    ...
}
```

We call it with the arguments in the following order

```
tixChainMethod $w MethodToChain $arg1 $arg2 ... $argn
```

We'll come back to more detailed discussion of `tixChainMethod` shortly. For the time being, let's take it for granted that `tixChainMethod` must be used in the three standard initialization methods: `InitWidgetRec`, `ConstructWidget` and `SetBindings`

6.4.2 The ConstructWidget Method

The `ConstructWidget` method is used to creates the components of a widget instance. In the case of TixArrowButton, we want to create a new button subwidget, whose name is `button`, and use a bitmap to display an arrow on this button. Assuming the bitmap files are stored in the files `up.xbm`, `down.xbm`, `left.xbm` and `right.xbm`, the string substitution `@$data(-direction).xbm` will give us the appropriate bitmap depending on the current direction option of the widget instance.

```
proc tixArrowButton::ConstructWidget {w} {
    upvar #0 $w data

    tixChainMethod $w ConstructWidget

    set data(w:button) [button $w.button -bitmap @$data(-direction).xbm]
    pack $data(w:button) -expand yes -fill both
}
```

The `tixArrowButton::ConstructWidget` method shown above sets the variable `data(w:button)` to be the pathname of the `button` subwidget. As a convention of the Tix Intrinsics, we must declare a public subwidget *swid* by storing its pathname in the variable `data(w:`*swid*`)`.

6.4.3 The SetBindings Method

In your interface, you want to handle a lot of events in the subwidgets that make up your mega-widget. For instance, when somebody presses the button in a TixArrowButton widget, you want the button to handle the event. The `SetBindings` method is used to creates event bindings for the components inside the mega-widget. In our TixArrowButton example, we use the bind command to specify that the method `tixArrowButton::IncrCount` should be called each time when the user presses the first mouse button. As a result, we can count the number of times the user has pressed on the button (obviously for no better reasons than using it as a dumb example).

```
proc tixArrowButton::SetBindings {w} {
    upvar #0 $w data

    tixChainMethod $w SetBindings

    bind $data(w:button) <1> "tixArrowButton::IncrCount $w"
}

proc tixArrowButton::IncrCount {w} {
    upvar #0 $w data

    incr data(count)
}
```

6.5 Declaring and Using Variables

The private variables of a widget class do not need to be declared. In fact they can be initialized and used anywhere by any method. Usually, however, general purpose private variables are initialized by the `InitWidgetRec` method and subwidget variables are initialized in the `ConstructWidget` method.

We have seen in the `tixArrowButton::InitWidgetRec` example that the private variable `data(count)` was initialized there. Also, the private variable `data(w:button)` was initialized in `tixArrowButton::ConstructWidget` and subsequently used in `tixArrowButton::SetBindings`.

In contrast, public variables must be declared inside the class declaration. The following arguments are used to declare the public variables and specify various options for them:

- `-flag`: As shown in the class declaration in figure 6.1, the `-flag` argument declares all the public variables of the TixArrowButton class, `-direction` and `-state`
- `-configspec`: We can use the `-configspec` argument to specify the details of each public variable. For example, the following declaration

  ```
  -configspec {
      {-direction direction Direction e}
      {-state state State normal}
  }
  ```

  specifies that the `-direction` variable has the resource name `direction` and resource class `Direction`; its default value is `e`. The application programmer can assign value to this variable by using the `-direction` option in the command line or by specifying resources in the Tk option database with its resource name or class. The declaration of `-state` installs similar definitions for that variable.
- `-alias`: The `-alias` argument is used to specify alternative names for public variables. In our example, the setting

  ```
  -alias {
      {-dir -direction}
  }
  ```

  specifies that `-dir` is the same variable as `-direction`. Therefore, when the application issue the command

  ```
  .up config -dir w
  ```

  it is the same as issuing

  ```
  .up config -direction w
  ```

  The `-alias` option provides only an alternative name for the application programmer. Inside the widget's implementation code, the variable is still accessed as `data(-direction)`, *not* `data(-dir)`.

6.5.1 Initialization of Public Variables

When a widget instance is created, all of its public variables are initialized by the Tix Intrinsics before the `InitWidgetRec` method is called. Therefore, `InitWidgetRec` and any other method of this widgte instance are free to assume that all the public variables have been properly initialized and use them as such.

The public variables are initialized by the following criteria.

- Step 1: If the value of the variable is specified by the creation command, this value is used. For example, if the application programmer has created an instance in the following way:

  ```
  tixArrowButton .arr -direction n
  ```

  The value `n` will be used for the -direction variable.

- Step 2: if step 1 fails but the value of the variable is specified in the options database, that value is used. For example, if the user has created an instance in the following way:

  ```
  option add *TixArrowButton.direction w
  tixArrowButton .arr
  ```

  The value `w` will be used for the `-direction` variable.

- Step3: if step 2 also fails, the default value specified in the `-configspec` secton of the class declaration will be used.

Type Checker

You can use a *type ckecker procedure* to check whether the user has supplied a value of the correct type for a public variable. The type checker is specified in the `-configspec` section of the class declaration after the default value. The following code specifies the type checker procedure `CheckDirection` for the `-direction` variable:

```
    -configspec {
        {-direction direction Direction e CheckDirection}
        {-state state State normal}
    }
    ...
}

proc CheckDirection {dir} {
    if {[lsearch {n s w e} $dir] != -1} {
        return $dir
    } else {
        error "wrong direction value \"$dir\""
}
```

Notice that no type checker has been specified for the `-state` variable and thus its value will not be checked.

If a type checker procedure is specified for a public variable, this procedure will be called once the value of a public variable is determined by the three steps mentioned above.

6.5.2 Public Variable Configuration Methods

After a widget instance is created, the user can assign new values to the public variables using the configure method. For example, the following code changes the `-direction` variable of the `.arr` instance to `n`.

```
.arr configure -direction n
```

In order for configuration to work, you have to define a configuration method that does what the programmer expects. The configuration method of a public variable is invoked whenever the user calls the configure method to change the value of this variable. The name of a configuration method must be the name of the public variable prefixed by the creation command of the class and `::config`. For example, the name configuration method for the `-direction` variable of the TixArrowButton class is `tixArrowButton::config-direction`. The following code implements this method:

```
proc tixArrowButton::config-direction {w value} {
    upvar #0 $w data

    $data(w:button) config -bitmap @$value.xbm
}
```

Notice that when `tixArrowButton::config-direction` is called, the `value` parameter contains the new value of the `-direction` variable but `data(-direction)` contains the `old` value. This is useful when the configuration method needs to check the previous value of the variable before taking in the new value.

If a type checker is defined for a variable, it will be called before the configuration method is called. Therefore, the configuration method can assume that the type of the `value` parameter is got is always correct.

Sometimes it is necessary to override the value supplied by the user. The following code illustrates this idea:

```
proc tixArrowButton::config-direction {w value} {
    upvar #0 $w data

    if {$value == "n"} {
        set value s
        set data(-direction) $value
    }

    $data(w:button) config -bitmap @$value.xbm
    return $data(-direction)
}
```

Notice the above code always overrides values of `n` to `s`. If you need to override the value, you must do the following two things:

- Explicitly set the instance variable inside the configuration method (the `set data(-direction) $value` line).

- Return the modified value from the configuration method.

If you do not need to override the value, you don't need to return anything from the configuration method. In this case, the Tix Intrinsics will assign the new value to the instance variable for you.

Configuration Methods and Public Variable Initialization

For efficiency reasons, the configuration methods are not called during the intialization of the public variables. If you want to force the configuration method to be called for a particular public variable, you can specify it in the `-forcecall` section of the class declaration. In the following example, we force the configuration method of the `-direction` variable to be called during intialization:

```
-forcecall {
    -direction
}
```

6.6 Summary of Widget Instance Initialization

The creation of a widget instance is a complex process. You must understand how it works in order to write your widget classes. The following is the steps taken by the Tix Intrinsics when a widget instance is created:

- When the user creates an instance, the public variables are intilized as discussed in section 6.5.1. Type checkers are always called if they are specified. Configuration methods are called only if they are specified in the `-forcecall` section.

- The `InitWidgetRec` method is called. It should initialize private variable, possibly according to the values the public variables.

- The `ConstructWidget` method is called. It should create the component widgets. It should also store the names of public subwidgets into the subwidget variables.

- The `SetBinding` method is called. It should create bindings for the component widgets.

After the above steps, the creation of the instance is complete and the user can iterate with it using its widget command.

6.7 Loading the New Classes

Usually, you can use a separate script file to store the implementaion of each new widget class. If you have several of those files, it will be a good idea to group the files into a single directory and create a `tclIndex` file for them so that the new classes can be auto-loaded.

Suppose you have put the class files into the directory `/usr/my/tix/classes`. You can create the `tclIndex` file using the `tools/tixindex` program that comes with Tix:

```
cd /usr/my/tix/classes
/usr/my/Tix4.0/tools/tixindex *.tcl
```

The `tclIndex` file must be created by the `tixindex` program. You cannot use the standard `auto_mkindex` command that comes with Tcl.

Once you have created the `tclIndex` file, you can use your new widget classes by auto-loading. Here is a small demo program that uses the new TixArrowButton class:

```
#!/usr/local/bin/tixwish
lappend auto_path /usr/my/tix/classes

# Now I can use my TixArrowButton class!
#
tixArrowButton .arr -direction n
pack .arr
```